INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Access Manager

A DBMS framework for extensible access methods

Ralph Acker

TECHNISCHE UNIVERSITÄT MÜNCHEN
Institut für Informatik

Access Manager:
A DBMS framework for extensible access methods

Ralph Acker

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzende: Univ.-Prof. G. J. Klinker, Ph.D.
Prüfer der Dissertation:
1. Univ.-Prof. R. Bayer, Ph.D. (em.)
2. Univ.-Prof. Dr. M. Bichler

Die Dissertation wurde am 16.02.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 09.06.2011 angenommen.

In memory of my father, Martin Acker.

Abstract

The presented *Access Manager framework* provides modular extensibility to a standard database management system (DBMS), for bridging the functional gap between a data-independent DBMS implementation and the specific requirements of a particular application domain for specialized access methods, permitting efficient data retrieval, maintenance, and storage. Therefore it opens the architecture of an operational standard DBMS in selected areas, by introducing a concise yet flexible and powerful interface to the DBMS's components.

The DBMS will function as a host system for accommodating application domain-specific plug-ins, allowing defined adaptation and customization of the host DBMS by introduction of *access modules* for alternative primary and secondary access methods, supported by custom *algorithmic units* implementing auxiliary transformations, and a potent *data integration layer*. This interface particularly emphasizes thorough integration of extension modules into the host system's intrinsic query optimization process, by devising a fully-automated, negotiation-based technique for constructing, transforming, and assessing efficient query evaluation plans containing external modules.

Both interface and its corresponding protocol are designed for actively facilitating the development of extension modules by encouraging modularization and reuse. As a consequence, coding complexity of access modules depends closely on the complexity of the new access structure and its distinctiveness from existing implementations. We prove the framework's ability to provide true extensibility by augmenting an existing host DBMS with several additional access methods. To this end, a prototype implementation of the Access Manager framework was successfully integrated into the relational DBMS Transbase of Transaction Software GmbH.

Zusammenfassung

Das vorgestellte *Access Manager Framework* erlaubt die modulare Erweiterung eines herkömmlichen Datenbank Management Systems (DBMS) zur Überwindung der funktionalen Diskrepanz zwischen einem datenunabhängigen Datenbanksystem und den besonderen Anforderungen eines bestimmten Anwendungsgebiets. Dafür sollen speziell zugeschnittene Zugriffsmethoden für effiziente Datenhaltung, Suche und Manipulation hinzugefügt werden. Zu diesem Zweck wird die Architektur eines voll einsatzfähigen DBMSs an ausgewählten Stellen über eine kompakte, flexible und zugleich mächtige Schnittstelle geöffnet, um Zugriff auf interne Komponenten des Systems zu erlauben.

Das DBMS übernimmt dabei Rolle eines Wirtssystems, das die Implementierung anwendungsspezifischer Plug-ins zur wohldefinierten Erweiterung und Anpassung an eine Datenbank-Applikation gestattet. Bei diesen Erweiterungen handelt es sich erstens um alternative primäre oder sekundäre *Zugriffsmethoden*, die durch maßgeschneiderte *Algorithmen* für relationale Transformationen ergänzt werden können und zweitens um eine mächtige *Integrationsschicht* für den Zugriff auf externe Daten. Die Schnittstelle gewährleistet im Besonderen eine grundlegende Integration solcher Erweiterungen in den Anfrageoptimierer des Wirtssystems durch einen vollautomatisierten, verhandlungsbasierten Prozess für Konstruktion, Transformation und Kostenabschätzung effizienter Anfragepläne, welche externe Module beinhalten.

Sowohl die Schnittstelle als auch das zugehörige Protokoll wurden so entworfen, dass sie den Entwicklungsprozess externer Module durch Modularisierbarkeit und Wiederverwendbarkeit von Komponenten erleichtern. Dadurch wird eine starke Korrelation zwischen dem Entwicklungsaufwand neuer Module und deren Komplexität bzw. deren Unterschied zu existierenden Modulen bewirkt. Als Beleg für die Eignung dieses Frameworks für echte Erweiterbarkeit eines bestehenden DBMS wurde ein Prototyp in das relationale Datenbanksystem Transbase der Firma Transaction Software GmbH integriert.

Acknowledgements

First of all, I thank my supervisor Prof. Rudolf Bayer, Ph.D. for his support and invaluable feedback during the different phases of this work, especially as this external thesis was characterized by a ‘variable’ pace, induced by schedules of pending projects for my employer, such that progress came in abrupt leaps rather than as a constant flow. This necessitated him to re-familiarize with the topic on short notice, which he always did with untiring interest, while providing confidence and inspiring advice.

I am particularly grateful to Dr. Christian Roth, CEO of Transaction Software, for providing the opportunity for setting out on this extensive research project and actively supporting it in many ways, thereby effectively establishing the preconditions that made this thesis possible. Also thanks to Dr. Klaus Elhardt, CTO of Transaction Software, who tolerated the numerous intrusions of the Access Manager framework into critical Transbase system components, and for the seeing past the initial irritations of its introduction.

All Transaction staff members helped greatly improving the outcome of this thesis, by providing constructive criticism in numerous discussions on questionable design decisions, but also for acknowledging, encouraging, and supporting all achievements. Special thanks for active support to my colleague Simon Valentini, for adopting the Flat Table in a very early state, when it was no more than a mere case study, and perfecting it to the industrial strength access module to which it eventually evolved. Moreover, I want to thank Dr. Roland Pieringer, who provided invaluable support and guidance during the early phase of this work.

Very special thanks to my love Tanja, for moral support, motivation, and for diverting me when I was hopelessly entangled. Also for proofreading this thesis on a subject outside her own discipline - not so many thanks for laughing out loud at my less fortunate formulations. More thanks to Ulrike Herrmann, for helping me with a highly intricate administrative procedure of TUM.

Finally, I thank my family and friends for their patience and constancy, although I was only able to spend very little time and energy for sustaining the relationships. This is going to change – I promise.

Contents

1. Introduction

The demand for modeling structured data coming from a designated application domain introduced user-defined data types into standard DBMSs. To satisfy the need for support of natural operations on these types, user-defined functions were incorporated. Finally, these operations had to be integrated via an extensible indexing framework into the core system's access methods to supplement efficient storage and retrieval functionality. The features of Informix DataBlades, Oracle Data Cartridges, and IBM Extenders, to name the most established ones, are widely known throughout the scientific and industrial community, each using a different approach to open the host system architecture to a certain degree. Yet these frameworks are often found to be either too complex or not flexible enough to cope with the wide range of requirements in domain-specific access methods. Moreover, the available extensible indexing frameworks are clearly not suitable for rapid development and evaluation of research prototypes. As a consequence, implementations of such prototypes usually integrate a loosely coupled DBMS (via its API) or no DBMS at all. Hence, a broad variety of prototype implementations for closely related research topics exist, but result comparison or transferability remains difficult or impossible.

The implementation of new access methods for DBMSs, native integration of new data types, alternative data models (e.g. unstructured data, XML), or other core extensions usually result in major modifications of the DBMS kernel. Namely, the integration of a new index structure requires changes in the SQL compiler, query plan optimizer, access path layer, cache manager, lock manager, and the physical storage layer. The latter is also likely to affect logging and recovery facilities.

Such changes of the database core system make it very expensive, time consuming, and error prone to implement and test new access methods, user-defined data types, alternative physical data layout models, contiguous data flow from streaming data sources, and external storage approaches such as heterogeneous federated DBMS, external data files, or data on the net. With the existing approaches, comparison of technologies for applicability in a project or for scientific purposes is only possible with an isolated environment, usually outside a DBMS. For example, Generalized Search Trees (GiST [Hel95]) offer a generic template for tree-based index implementation. But there is still no industry-standard DBMS that thoroughly integrates this framework. On the other hand, there exist frameworks in commercial DBMSs that support integration of new access structures. For example, Oracle Data Cartridges provide DBMS

extensibility but are restricted to secondary index integration. Custom implementations of clustering primary indexes cannot be integrated with this module. Open source DBMSs can be modified in the source code, but they do not explicitly provide interfaces for integrating new structures without modifying major parts of the kernel code in an error prone venture. We will introduce the *Access Manager* specification as a new programming interface to several layers of a DBMS kernel. It enables a programmer to add new data structures and auxiliary operators to a DBMS with a minimum of effort. Therefore, it is very suitable for rapid development of new functionality and allows comparison against other techniques, having all features of a complete DBMS at hand for representative benchmarking. A prototype of the Access Manager interface was implemented into Transbase [Tra10], a fully-fledged relational DBMS with a highly modularized architecture.

1.1. Objective

An extensible indexing architecture is a powerful framework that is adaptable to domain-specific requirements of a DBMS application. Its basic purpose is to support natural operations on domain-specific data for efficient storage and retrieval. Examples for such domain-specific data are multimedia objects, documents (structured and unstructured), temporal and spatial data, scientific and engineering data. Date storage in a primary access path and additional indexes as secondary access paths are both available. The query plan optimizer autonomously chooses the best available access path for a query, so access path selection remains completely transparent to the user (e.g. in SQL). Additionally, the framework supports and enforces all necessary operations on all involved access structures throughout all DBMS tasks. That is, all modifications (insert/ update/ delete) on primary and secondary access paths are carried out consistently within their transactional context. Data integrity is enforced independently, and data as well as system security is provided through logging and recovery facilities. Access privileges of database users are maintained in a centralized data dictionary. Multiple operations of concurrent users are processed in parallel, offering locking technology and concurrency control on a reasonably fine granular basis. Intra-query parallelism is available for performance improvements. Access methods have direct influence on performance characteristics of the system through their ability of holding required pages in the DBMS data cache. The implementation itself allows for rapid prototyping and provides sophisticated testing and debugging facilities. It encourages common software engineering concepts, namely modularization and reuse. As a consequence, coding complexity depends closely on the complexity of the new access structure and its distinctiveness from existing implementa-

tions. Built-in access methods of the database system (i.e. B-tree [Bay72], UB-tree [Bay96], and Full-text) are available for reuse as modular components in a resource pool. Moreover, every implementation of a new extension also becomes immediately available as a reusable component in this resource pool. Additionally, the host framework provides a rich set of utility functionality for common tasks. System stability and data security is not to be compromised. Portability of existing access methods to other DBMSs implementing the Access Manager framework is desirable. The most important task of all is to devise a compact yet adaptive interface for integrating data access structures into the host DBMS that is capable of exploiting access method characteristics thoroughly and efficiently. Finally, the framework should provide enough flexibility for incorporating future requirements.

1.2. Structure

This thesis is divided into two major parts. The first part will provide the theoretical basis for an extensible relational query evaluation model by deriving its essential principles from Relational Algebra in *Chapter 2: Theory*. This chapter provides an abstract conception of relational algorithms consisting of classical relational operators and an instrumentation to handle these constructs. Finally, we will establish the main features of the *scan operator*, an abstract relational operator that provides all functionality required to encapsulate an arbitrary access method. After outlining the theoretical scope of this work, *Chapter 3: Related Work* will survey other existing approaches towards extending database systems and compare them to our own approach. The second part, beginning with *Chapter 4: Architecture*, will provide in-depth descriptions of the proposed framework and its operation inside the host DBMS Transbase. This will be followed by a detailed description of various existing implementations based on the Access Manager concept in *Chapter 5: Proof of Concept*. The functionality of the prototypes will be additionally endorsed by examination and comparison of important performance aspects. The final chapter will summarize the results of this thesis and indicate possible directions of future work.

2. Theory

In 1969, the original concept of relational database systems was established by introduction of the relational model by E. F. Codd [Cod70]. The motivation of his contribution was to protect database users and database application programs from dealing with internal data representations of the DBMS. At that time, storage structures were an integral part of the data and in-depth understanding of these structures was required for data retrieval and navigation. Codd's ultimate goal was to provide an abstraction of data representation that permits a database user to exploit relational data by knowing no more than names and semantics of relations and attributes. He summarized his mathematical approach in the formulation of the *relational algebra* (RA). Although never fully implemented in its original claim, this algebra established itself as the common basis of relational DBMSs. In the following years, SQL (Structured Query Language) evolved as *equivalent, declarative* counterpart to the *procedural* RA, providing a comprehensive, descriptive approach for relational database queries. In this process the strict set-theoretical features of RA were slightly softened for increased usability and improved performance characteristics. Eventually, SQL has become the standard relational query language. SQL and the Extended Relational Algebra (ERA) on which it is based, are the axiomatic features that define today's relational database concept.

On this foundation of the relational world, we will construct a novel theoretical model for query evaluation. Our model is extensible as it applies to arbitrary external relational algorithms. It will allow for these algorithms to be plugged into a host DBMS where they are employed automatically and efficiently. Algorithms can be added, refined, replaced, or removed at any time without affecting operability or consistency of the overall system.

Our focus is to use such algorithms for implementing supplementary data access paths. The goal is to provide more flexibility to commodity RDBMS technology by enabling it to cope with application-specific demands for data storage and retrieval. For this, we utilize the primary purpose of RA, its ability to provide an abstraction from internal data representations. But in this case the abstraction shall relieve the DBMS itself from any dispensable details of particular access methods, providing a generalized model for an extensible query evaluation engine based on common ERA, but capable of incorporating arbitrary user-defined access methods and auxiliary relational operators.

2.1. Relational Algebra

ERA terms are generated by a RDBMS when an SQL query, formulated by a user, is translated (compiled) for evaluation. Compilation exploits the equivalence of SQL and ERA for making the transition from the high-level descriptive query language to a first procedural representation in ERA that is iteratively computable by an abstract query evaluation engine. ERA terms represent an intermediate level query language where global algorithmic considerations are involved. This again is the abstract equivalent of a fully-fledged Query Evaluation Plan (QEP) where every detail of query evaluation is decided. We start by briefly discussing the important characteristics of Codd's original RA and some of the differences between RA and ERA.

RA operates on relations. If $\mathcal{D}_1, \dots, \mathcal{D}_n$ are sets, then a relation R of degree n is a finite subset of the Cartesian product $\Omega_R = \mathcal{D}_1 \times \dots \times \mathcal{D}_n$. R consists of n-tuples of the form $(a_1, \dots, a_n)$ such that $a_i \in \mathcal{D}_i$. $\mathcal{D}_i$ is called the domain of attribute a_i and Ω_R is the domain of R. The cardinality $|R|$ is defined as the number of elements in R. Let A be the set of all possible finite relations over an arbitrary but finite set of domains. We define:

Definition.1: $a \in A$ is a RA term.

The *primitive* operators of RA are projection (π), selection (σ), rename (ρ), Cartesian product ($\times$), set union ($\cup$), and set difference ($\setminus$). All other RA operators can be constructed from this set of primitive operators. RA operations are closed on A in the sense that every n-ary RA operator f operates on $n \geq 1$ relations to yield one relation, or formally:

Definition.2: $f : A^n \to A$ is a RA term.

Definitions 1 and 2 allow the recursive definition of all well-formed RA terms. It also follows directly that operators $f: A^n \to A$ with $n \geq 1$ and

$g_i: A^{m_i} \to A$ with $1 \leq i \leq n$ and $m_i \geq 1$

can be composed. For their composition holds the following

Corollary.1: $f \circ (g_1, \dots, g_n): A^{m_1} \times \dots \times A^{m_n} \to A$ is a RA term.

Composition of RA operators also introduces the concept of *intermediate results*, i.e. the outcome of operator $g_i: A^{m_i} \to A$ before f is applied.

The alert reader will have noticed that we permit n-ary operators, while classic RA considers only unary and binary operators. Our conception of a *generic n-ary relational operator* is capable of accepting a *variable* number of n≥2 input relations. This guarantees that any n-ary operation f can always be decomposed into a cascade of binary f operations. Hence, the addition of n-ary operations to classical RA does not compromise RA universality in any way. We henceforth adopt generic n-way operations for our purposes, as we expect them to offer additional performance relevant opportunities through their increased compactness. For such generic n-ary operators we define:

Definition.3: Generic n-ary Operator: $\forall f, g_i$:
$f: A^n \to A$ with $n \geq 2$ and $g_i: A^{m_i} \to A$ with $1 \leq i \leq n$ and $m_i \geq 1$:

$$f \circ (g_1, \dots, g_\mathrm{n}) \equiv f \circ (f \circ (g_1, g_2), \dots, g_\mathrm{n})$$

For the sake of a more intuitive presentation, RA terms are often depicted in tree representation. In this representation, all leaves are input relations (following Definition.1) and all internal nodes of the tree are operators (Definition.2). The precedence of operators corresponds to the parent-child relationships in the operator tree.

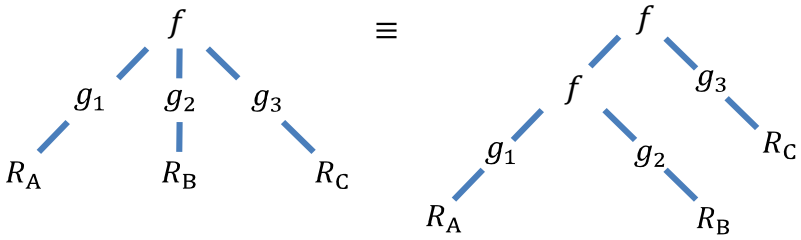


Figure.1 Two equivalent sample QEPs. Both plans apply the unary operators g_i after accessing relations $R_\mathrm{A}, R_\mathrm{B}, R_\mathrm{C}$. On the left, these *intermediate results* are joined using the generic n-way operator f, operating on three input streams, while on the right side the same operation is conducted using a cascade of binary f operations.

In addition to composition, there exists a set of transformation rules for converting a given RA term into an equivalent RA term. The query plan optimizer of a DBMS applies such transformations in its effort to derive the most efficient plan for a query. The details of these transformations are beyond the scope of this work. We exemplary name two well-known techniques: projection pushdown and selection pushdown. The goal of these transformations is to reduce the amount of data to be processed as early as possible, i.e. as low as possible in the QEP tree. They translate figuratively into the tree representation of query plans where these operators are pushed towards the leaf nodes. Besides this, the optimizer is in charge of selecting access paths, i.e. deciding whether to use the primary access path or one or more of

the secondary indexes during evaluation. Finally, the optimizer chooses the appropriate evaluation algorithms, e.g. whether to process a join using the Sort-Merge, Nested-Loop or Hash-Join algorithm. After optimization is completed, the final QEP is ready for evaluation by the DBMS query processor.

2.1.1. Relations

Relations are the leaf nodes of operator trees. Similar to all other RA operators, relations have an equivalent representation in SQL. While the SQL subclass DML (data manipulation language) only references them by name, they are defined in DDL (data definition language). Data definition comprises all information on a particular relation that is visible to the DBMS. Properties appointed at data definition time are stored in the data dictionary of the DBMS and are re-evaluated when a relation is referenced from an SQL DML query. The data dictionary is the query optimizer's primary guidance for accessing a relation. It offers the possibilities to resolve relation names, column names, and column domains (data types). This information is sufficient for accessing relations in the way assigned by classical RA, namely by linearly traversing the relation and presenting all of its unaltered tuples to the parent operator for further processing.

DDL also provides various mechanisms for expressing additional interesting properties of relations such as *key* constraints (primary key), *unique* constraints (key candidates), *reference* and *check* constraints. Although these constraints often have no obvious impact on the way the DBMS plans its accesses to relations, the DBMSs usually maintain various auxiliary storage structures to efficiently enforce such constraints. These structures are, for example, implemented as B-trees allowing rapid duplicate recognition for supporting UNIQUE constraints. As these auxiliary structures are registered in the data dictionary, they can also be exploited by the DBMS optimizer for other purposes, i.e. as additional access path to a relation. As these mechanisms are concealed from the SQL user, they are not preferential for access path modeling. Still we have to note that access paths are possibly involved as a side-effect of constraint definition.

This leaves the definition of secondary indexes in DDL as the only true measure provided to SQL users for modeling access paths. With these, it is possible to define in a comprehensible way which attributes of a relation are to be stored redundantly in a secondary access path. Many DBMSs provide different index types (e.g. B-trees, multidimensional/ spatial indexes, bitmaps, etc.) allowing modeling access paths tailored to a particular purpose. But the differ-

ent DBMSs' DDL dialects tend to be cluttered with all sorts of physical storage parameters, comprising access path independent information, such as data partitioning and data allocation directives, but also access path specific information. All this information has to be reflected by the data dictionary, so the optimizer can put it to good use. This complex mixture of information makes modeling an efficient data dictionary a challenging problem, even for a limited number of access path types. Clearly, this is not a feasible approach for a DBMS with user-defined access structures. We therefore propose to separate access path specific information from general access path independent information. General information remains publicly available in the data dictionary, while the access path relevant data becomes private to a particular access path implementation, i.e. it is not visible to the DBMS optimizer. Only the type of an access path must be stored in the data dictionary as a discriminator.

This abstraction offers the possibility of a well-structured data dictionary and greatly relieves the optimizer from having to deal with different access path types. On the other hand, it introduces the requirement for a new approach to efficiently exploit access paths with properties that are not described in the data dictionary and hence are unknown to the query optimizer. In the following, we will develop our solution to this problem by examining the interactions of relations with other relational operators.

2.1.2. Operators

Without a formal definition, we now establish our notation of RA operators, assuming that the reader is familiar with these concepts. Let *R* be an *n*-ary relation with attributes $(a_1, \dots, a_n)$. The RA unary operators are:

(1) Projection: $\pi_{a_i,\dots,a_j}(R)$ where $\{a_i, \dots, a_j\} \subseteq \{a_1, \dots, a_n\}$. Projection allows permutation and omission of attributes in R.

(2) Selection: $\sigma_\varphi(R)$ where φ is a propositional function consisting of atoms of the form $a_i \Theta a_j$ or $a_i \Theta c$. Θ is a binary comparison operation that is compatible with the attributes a_i, a_j and a constant value c. The atoms are combined using logical operators $\wedge, \vee, \neg$. The selection $\sigma_\varphi(R)$ extracts all tuples in R for which φ holds.

For the sake of completeness we must also mention the *rename* operator: $\rho_{new \leftarrow old}(R)$. This construct is supplied for substitution of names of attributes and relations, preventing name collision and ambiguity when joining relations (in particular for self-joins). After SQL compilation is completed and the RA operator tree was constructed, all ambiguity is eliminated and

the original names become irrelevant. Moreover, in our notion, RA operators access attributes of their input *n*-tuples by referring to the ordinal position of that attribute rather than to its name. Consequently, the rename operator can be neglected in the following.

The RA set operators are the basic n-ary set operations as known from set theory. Namely they are:

(3) Set Union: $\bigcup_{1 \le i \le n} (R_i)$

(4) Set Difference: $\setminus_{1 \le i \le n} (R_i)$

(5) Cartesian product: $\times_{1 \le i \le n} (R_i)$

Codd [Cod70] proved that this collection of operators is sufficient to achieve an expressiveness that he defined as *relational completeness*, i.e. RA's expressive power is equivalent to the domain-independent *relational calculus*, and thus *first-order logic*. Although relational completeness clearly does not imply that its expressiveness is sufficient to formulate every 'interesting' database query, Codd identified it to be an important milestone towards this target.

In order to add more functionality, RA operators were modified and additional operators were introduced to form the Extended RA (ERA), which is equivalent to the expressiveness of SQL, the de-facto standard of today's relational DBMSs. In contrast to the original RA, ERA operates on multi-sets (bags) instead of strict sets. This implies that ERA operators, unlike their RA counterparts, do not implicitly eliminate duplicates after every operation. The functional reason of this discrepancy between RA and ERA is that SQL offers explicit control over duplicates and their elimination and ERA must reflect this. The other aspect is that duplicate elimination is an expensive operation, because sorting is an algorithmic prerequisite for an efficient implementation. Thus, it may be cheaper to apply several relational operations before eliminating duplicates. In this work, when talking about relational operators, we always refer to ERA operators without implicit duplicate elimination for all previously introduced operators.

To preserve an expressiveness equivalent to RA, ERA introduces a new unary operator:

(6) Distinction (Bag-to-Set Operator): $\delta(R)$

It is clear that ERA operators with their multi-set semantic combined with a subsequent distinction operator are equivalent to the RA operators. As an immediate consequence, the algebra formed by ERA operators (1) - (6) obviously satisfies the constraints of relational completeness.

RA treats relations as unordered sets, i.e. the ordering of sets is not significant. As SQL offers explicit control over the ordering of sets, we also introduce sorting as unary operator:

> *(7) Sort*: $\tau_{a_i,\dots,a_j}(R)$ where $\{a_i, \dots, a_j\} \subseteq \{a_1, \dots, a_n\}$. τ_{a_i} indicates ascending sort order on attribute a_i, while $\tau_{\overline{a_i}}$ denotes descending order.

The sort operator can produce any *lexicographical* order on the attributes of an input set, as required for equivalence to the SQL ORDER BY expression. The addition of the sort operator has no relevance for relational completeness, but it is crucial for equivalence of ERA and SQL. Moreover, the order of intermediate results is significant for the applicability of efficient algorithms when it comes to the implementation of relational operators, a topic to be addressed later in greater detail.

With these seven operators, the set of ERA operators is certainly not complete. There are still numerous SQL constructs left that cannot be expressed with this algebra, e.g. *arithmetic* operators, *grouping*, and *aggregation* are fundamental concepts of SQL. As a matter of fact, a general ERA cannot be completed, as SQL is permanently evolving. Even SQL standardization cannot keep pace with SQL extensions of the 'three large' commercial RDBMSs and often selects one of three solutions to become the SQL standard. This implies also that in addition to standard SQL there exist at least two more SQL dialects (with subtly distinctive ERA operator sets) of practical relevance.

For the moment, we will therefore assess that our subset of seven ERA operators achieves relational completeness, i.e. the algebra defined by these operators is sufficiently comprehensive for expressing a relevant subset of SQL functionality. Therefore, we define a name for this set:

Definition.4: { π, σ, $\cup$, $\setminus$, $\times$, δ, τ } are basic ERA operators.

Our next step is to combine these operators to form expressions of higher complexity.

2.1.3. Composition

With the first six primitive ERA operators { π, σ, $\cup$, $\setminus$, $\times$, δ } it is possible to construct the set of remaining RA operators *Intersection* ($\cap$), *Natural Join* ($\bowtie$), *Theta Join* (Θ_{φ}), *Semi Join* ($\ltimes$), *Antijoin* ($\triangleright$), *(Full) Outer Joins* ($\bowtie$, $\bowtie$), *Division* ($\div$), etc. These *compound* operators do not add expressiveness to ERA. But they are highly valuable in terms of algorithmic complexity, because often more efficient implementations exist for such compound operators than the mere combination of primitive operators allows. Therefore RDBMSs implement both primitive and compound operators as building blocks for evaluation of relational queries. Additionally a class of *complex* ERA operators exists, which cannot be constructed using primitive operators. Complex ERA operators bridge the functional gap between the expressiveness of pure RA and rich SQL concepts, for example *arithmetic* operators, *grouping*, and *aggregation*.

2.2. Query Planning

In the phase of query planning, the DBMS optimizer is aiming to deduce the optimal sequence of ERA transformations for answering a query within the present system parameters. Usually the goal is minimization of calculation time or maximization of system throughput by maximizing utilization of limited resources such as CPUs, memory and I/Os.

Besides arranging the sequence of transformations, the optimizer often has several alternative algorithms available for representing one particular ERA operator, e.g. *Sort-Merge, Nested-Loop* and *Hash-Joins* are well-known examples for join algorithms. The optimizer also decides which particular algorithm to apply under the given preconditions of a query plan.

2.2.1. Optimization

All RDBMSs are typically equipped with cost-based (as originally proposed in [Sel79]) or rule-based query plan optimizers. Without going into detail, all optimization steps are conducted by gradually constructing an operator tree from ERA terms and subsequently refining it by substituting term partitions with equivalent terms. The goal of these transformations is to find the optimal sequence of relational operations that were specified in the declarative SQL query and employment of efficient algorithms for inexpensive query evaluation. An algorithm is one particular implementation of one ERA term. The optimizer can therefore replace a term using one of several matching algorithms. For example, an ERA join operator can be implemented by the sort-merge or nested-loop join algorithms.

For supporting ERA transformations and for justification of algorithmic decisions, the DBMS optimizer must possess at least two minimal instruments for term comparison.

(1) *Qualitative Equivalence* allows recognition that the ERA term f and an algorithm $[g]$ implementing the class of equivalent ERA terms $\bar{g}$ are interchangeable.

$$f \equiv [g] \leftrightarrow f \in \bar{g}$$

(2) *Quantitative Efficiency* allows comparison of costs for evaluating two equivalent algorithms $[f]_1$ and $[f]_2$ representing different implementations of the class ERA terms $\bar{f}$, and choosing the more efficient one.

$$[f]_1(R) \succcurlyeq [f]_2(R) \leftrightarrow cost\big([f]_1(R)\big) > cost\big([f]_2(R)\big)$$

The introduction of external algorithms into a given DBMS must therefore provide concepts for mapping external algorithms that are unknown to the DBMS to equivalent ERA terms known to the DBMS. With these instruments, the DBMS optimizer is enabled to conduct its work as usual, even when handling new algorithms implementing arbitrary ERA expressions. This is done without knowledge of the actual implementation of the algorithm, only by applying ERA substitution rules. This approach eliminates the need for any modifications of the optimizer when injecting novel algorithmic implementations into an existing system.

2.2.2. Costs

In essence, query planning is an iterative process of constructing several equivalent query plans. Then cost estimation is applied and the plans are ranked relative to their costs. Finally the least expensive plan is chosen for the next iteration and ultimately for query execution. Cost estimation is therefore the unquestioned key criterion to modern query plan optimization.

Query execution can induce a multitude of cost factors. An abstract conception of such costs is the occupation of limited resources during query execution. These resources range from computational costs on one or more CPUs, I/O costs, and memory consumption on the several levels of memory hierarchy. The impact of these costs is weighted according to a cost model implemented by the DBMS. The actual costs for occupying a limited resource may also vary depending on volatile conditions, like current system load, and on fixed configuration settings of the DBMS, such as preferred optimizations towards maximum throughput or minimal query latency.

Among the most important properties of a useful cost estimation facility are low computational complexity and high accuracy. Cost estimation is performed for a potentially high number of plan candidates. Therefore cost estimation's own costs have to be relatively low in comparison to the costs of actually executing a query. The costs of smaller plan fragments are used as basis for calculating the costs of more complex plans, thus even small relative errors may multiply and eventually cause substantially wrong estimations.

Cost estimation is usually based on statistical information that is maintained on relations and indexes. Such statistics typically comprise characteristics such as size of relations, e.g. total number of pages, and information on data distribution, such as cardinality, number of distinct values, minimum, maximum, etc. Additionally they provide selectivity estimations for predicates applied to relations or indexes, allowing estimation of a selection's cardinality. These cardinalities are then propagated bottom-up though the query plan. Cardinality estimation is useful in join optimization, when searching for a join-sequence with minimal intermediate result cardinalities. But its foremost purpose is assessment of a relation operator's CPU, memory, and I/O related costs, which are deducible from the operator's estimated input cardinality.

2.3. Interoperability

Until now, we used the concept of ERA term equivalence in optimization for enabling term substitution and term implementation with relational algorithms. An algorithm can be applied if and only if it replaces an ERA term without affecting the soundness of the complete operator tree, i.e. the correctness of the result must be preserved. In a data-centered approach, this means that alternative algorithms must yield equivalent output from identical input. In the following, we will elaborate on equivalence and its impact on interoperability of algorithmic implementations.

2.3.1. Equivalence

For strict RA, i.e. if we are operating on sets, the equivalence of two expressions is guaranteed if their output sets R and S are set equivalent:

$$\text{Set Equivalence: } R \equiv S \leftrightarrow R \setminus S = \emptyset \ \wedge \ S \setminus R = \emptyset$$

Equivalence of input and output sets is a necessary precondition for interoperability. In fact it is also a sufficient condition on the abstraction level of strict RA with its implicit duplicate

elimination. In ERA however, the intermediate results might be multi-sets (R, m_R), where $m_R: R \to \mathbb{N}_{>0}$ is a *multiplicity* function that denotes the number of occurrences of any element of the set R in (R, m_R). Hence set equivalence has to be extended to cover bag equivalence for these cases:

Bag Equivalence: $(R, m_R) \equiv_{BAG} (S, m_s)$

$$\leftrightarrow R \equiv S \ \wedge \ \forall r \in R, s \in S: r = s \rightarrow m_R(r) = m_S(s)$$

A weaker, alternative definition of bag equivalence as $R \equiv_{BAG} S \ \leftrightarrow \ \delta(R) \equiv \delta(S)$ is not sufficient for our purposes, since it ignores the quantity of duplicates, leading to a notion of equivalence that does not match with the semantics of ERA and SQL. The relevance of set or bag equivalence obviously depends on the existence of duplicates in intermediate results.

The three primitive unary ERA operators $\{ \pi, \sigma, \delta \}$ are able to establish set-equivalence for different inputs to a certain degree. They allow an algorithm's *input* set to be projected, pre-filtered, and purged of duplicates in order to bring it into a suitable form for applying that particular algorithm. On the other hand, these operations may be used for projection, post-filtering, and duplicate elimination on the algorithm's *output*, if this is required for meeting equivalence and the algorithm does not meet these requirements. Likewise bag-equivalence can be established by employing the two primitive RA operators $\{ \pi, \sigma \}$. Finally, the operator δ alone serves as universal bag-to-set operator.

2.3.2. Compatibility

For allowing the composition of operations $f \circ (g_1, \dots, g_n)$, f and g_i must meet certain requirements. In a data-centered approach this means that the output relations of g_i must be an adequate input for f. If, for example, *n* input relations are participating in an *n*-way union operation, they must be *domain compatible.*

Domain compatibility: $g_i, 1 \leq i \leq n$ are domain compatible, iff $\forall i, j: \ \Omega_{g_i} = \Omega_{g_j}$

From all primitive RA operators, domain compatibility, also often called union-compatibility, is explicitly required only for set-union $\cup$ and set-difference $\setminus$. But the composition of complex relational operators is responsible that operators, like intersection, inherit this requirement from its inherent primitive operations.

The domain concept of attributes entails the provision of data types for the attributes of relations. In RA and SQL, the comparisons of attributes of different data types are conducted

using highly generic comparison operators. Technically this is realized using type adaptation mechanisms, based on SQL's well-defined type hierarchy, to convert the arguments into compatible types. Domain compatibility is satisfied if adequate type adaptations are available. It is a prerequisite for comparison routines as used explicitly in selections, but also for implicit tuple comparison in many operators, e.g. union, difference and composed operators. Type adaptations are provided as a service by the DBMS, as the optimizer will provide all necessary type adaptations and thereby guarantees that operators always work with compatible data.

For our purposes, we will extend compatibility from mere domain compatibility to the more algorithmic notion of *representation compatibility*. Data representation is an issue whenever data is exchanged between operators. Strict RA only requires that data has to be exchanged in form of sets, without defining precisely how this is conducted. For our definition, we extend domain compatibility to comprise not only the arity but also a partitioning of exchanged sets, i.e. a set can be transferred as a whole, as in RA, or iteratively as horizontal partitions of the original set, either as subsets or tuple-wise. Another approach is vertical partitioning of data, i.e. transfer attribute by attribute, and finally combination of horizontal and vertical partitioning.

We further extend the notion of compatibility to allow arbitrary representations of data. Representation describes how the data to be exchanged is physically modeled and how the exchange is technically conducted. Representation ranges over a wide area of passing data by value, by reference, in structures (e.g. tuples), in buffers, compressed etc., allowing the use of the best representation for a particular purpose. The functional prerequisites of representation compatibility are achieved when two operators agree on one common data exchange format, while the ultimate choice of representation is a cost-based decision made by the query optimizer. With several operators supporting one particular representation, it is possible to compose a family of related algorithms based on that representation, allowing rapid data exchange, tailored to a particular purpose. Later we will discuss one use-case where bitmap indexes, unions and intersection, all based on the bitmap representation, form the components of a tightly coupled, highly efficient query evaluation system. For assuring interoperability under all circumstances, we also presume the existence of a *standard data representation* that has to be supported by all operators. For the moment, we only define transformation of data representation formally as a virtual ERA operator.

Representation: $\chi_k(R)$ denotes the transformation of the input R from any representation to a non-standard representation k. χ_{std} indicates transformation from any representation to standard data representation.

Representation is a virtual operator in the sense that it does not induce any relational transformation, thus it is not a relational operator as such, and it has no analogon in SQL. It serves only as a marker during query optimization to illustrate where and how the representation of the data stream changes. Representation will change as a side-effect when an algorithm implementing a relational operator is applied. More details on data representation will be defined later in the course of applying algorithms.

2.3.3. Data Flow

In strict RA, all 'children' of a relational operator have to be completely evaluated, yielding their complete result set, before the parent may begin processing. As a consequence, the DBMS would have to store intermediate results while they are processed by the subsequent operator. This is expensive if large amounts of data are involved. In the procedural perspective of query evaluation, input sets are not necessarily processed 'as a whole', as proposed by the relational model, they can also be processed in partitions, mostly one tuple at a time.

A common conception of query evaluation is the so-called *Iterator Model*, which describes this procedure abstractly and independent from the actual implementation. Its flow of control is a top-down recursion through the operator tree. First, the root operator is called, whose first task is to acquire input data for processing. Therefore it calls one of its children - which one is called depends on the operator's internal implementation. This call for input data propagates depth-first until it finally reaches a relation at a QEP leaf. This relation then produces its first tuple and returns it to its caller. Hence, data begins to 'flow' from the leaf nodes of the operator tree towards the root. Every operator acquires just enough input data to produce the first output tuple. As a consequence, we can classify ERA operators into two general groups:

Blocking Operators process all input tuples before any output tuple is generated.

Streaming Operators derive output before having seen the complete input.

This classification is very coarse, since it equalizes strict tuplewise streaming with operations that block for several tuples, e.g. grouping/ aggregations. The *blocking* property is depending on the actual implementation of an operator, not only on its function. Functionally streaming and blocking operators are equivalent. From the performance perspective however, streaming

is generally to be favored over blocking, because of potentially high requirements for temporal storage. Also query plans consisting only of streaming operators have more attractive performance characteristics, as the first result tuple can be delivered before the complete result set is calculated. Thus, the time to first results is often significantly shorter and the calculation time is (usually) evenly distributed over the result set while a blocking plan apparently uses all calculation time for the first result tuple and the remaining tuples are instantly available.

This query evaluation technique resembles the Iterator Model, where every operator exposes only one single interface routine, a *next()* method that is visible only to its parent. The parent repeatedly calls this method, causing the operator to *iterate* over its input, yielding its output successively as one tuple at a time. After the complete result is evaluated, the operator eventually terminates by returning the END-OF-DATA marker. Termination happens at leaf nodes if the input relation is exhausted. Internal nodes terminate if all input streams are exhausted (especially blocking nodes, like sort operations), or when a streaming operator cancels evaluation by announcing premature END-OF-DATA to its parent, e.g. existence quantification.

2.3.4. Sort Order

The transition to procedural operation and the Iterator Model exhibits new aspects of our operators. In contrast to the relational model, relations and intermediate multi-sets are processed tuplewise. This induces the concepts of set and multi-set traversal into ERA. Set traversal is formally equivalent to a linearization of the set along a binary relation $x < y$, i.e. x comes before y. In other words, iteratively traversing a set R establishes a *strict total order*. We describe such a linearization formally as

Strict total order: $\forall x, y \in R: \; x < y \vee y < x$

For base relations in a DBMS this order is usually, but not necessarily, predetermined by their storage order (*clustering*). For intermediate results, the output order is depending on the internal implementation of the particular operator that generates the output.

Obviously, the output order of an operator is at the same time the input order of its parent. The focus now lies on how to exploit such orders. The number of relevant orderings in a QEP is limited by the sort operator τ, as this is the prime operator for order generation. Consequently, if one particular operator requires a certain input order, it *must* be an order that can be generated by τ, otherwise the operator is not generally applicable. Thus we have to distinguish

between the general total order that is established through bag linearization and the set of orderings that can be generally exploited in QEPs. For the latter we will use the term *sortedness*. We say a bag is *sorted*, if its traversal corresponds to a *lexicographical* order on its attributes, i.e. its order is equivalent to one established by τ. Exploitable sort orders in QEPs are always *total* orders, but in the general case they are *non-strict*, because τ is not stringently covering all attributes and because of the existence of duplicates in bags. Nevertheless, many algorithms are either relying on non-strict input orders, or they are capable to benefit from such orders.

Sortedness: $\forall x, y \in R\colon\ x \leq_\tau y \ \vee\ y \leq_\tau x$

Strict sortedness: $\forall x, y \in R\colon\ x <_\tau y \ \vee\ y <_\tau x$

Sortedness of tuples is to be distinguished from equality of duplicates, as the sort criterion of τ may refer only to a subset of attributes, while equality always refers to the complete tuple.

$$\forall x, y \in R\colon x = y \rightarrow\ x =_\tau y \quad \text{but} \quad \forall x, y \in R\colon x =_\tau y \nrightarrow\ x = y$$

In summary the *sortedness* properties for any operator f inside a QEP are classified as:

Sort Order Establishing: $\forall x, y \in R\colon f(x) \leq_\tau f(y) \ \vee\ f(y) \leq_\tau f(x)$

Sort Order Preserving: $\forall x, y \in R\colon x \leq_\tau y \ \rightarrow f(x) \leq_\tau f(y)$

Sort Order Homomorphism: $\forall x, y \in R\colon x \leq_{\tau_1} y \ \rightarrow f(x) \leq_{\tau_2} f(y)$

Sort Order Disrupting: $\forall x, y \in R\colon x \leq_\tau y \ \nrightarrow f(x) \leq_\tau f(y)$

Sort order is established exclusively by the primitive sort operator τ, but it is possible to construct compound operators including sort operations and therefore inheriting the order establishing property. Many streaming operators are also order preserving, if they do not alter any order relevant fields like for example σ_φ and δ. One example for a sort order homomorphism is a projection π that removes some order-relevant attributes, while preserving the attribute of highest lexicographical significance. Sort order disruption can be provoked by applying a non-monotonic function to at least one of the sort relevant fields.

Order preservation is important when sortedness is not exploited by the immediate parent operator, but propagated to one of its ancestors to be used there. One can also imagine several consecutive order preserving operators that are all exploiting a given sort order.

Order disruption means that the output order does not satisfy any lexicographical sort criteria. One particular operator (including base relations) may be capable of accepting or producing one of several different sort orders or of preserving or disrupting an input order. Such output order is controlled by a configuration parameter of the encapsulated algorithm, and this parameter will also influence performance characteristics of the operator. Since the internal structure and functionality of the algorithm is unknown to the DBMS optimizer, it will employ a cost-based sort order negotiation protocol for determining the optimal configuration.

2.4. Substitution

Relational algorithms replace an equivalent ERA term in a query plan and thereby implement the replaced operations in one encapsulated, indivisible algorithmic unit. Similarly to one single query having many different equivalent algebraic representations, each of these algebraic representations may have many different algorithmic implementations. These implementations originate from a pool of available algorithmic building blocks provided by the DBMS. Finding the single optimal query plan for a given query in this huge search space of combinatory possibilities makes query optimization a challenging problem.

To cope with this complexity, DBMSs provide sophisticated term replacement mechanisms, employing rule-based and cost-based iterative decision processes for assembling and transforming alternative query evaluation plans. During this process, alternative plans are repeatedly assessed using cost estimation and unpromising candidates are pruned in order to limit the search space complexity. Extending such a complex mechanism, in order to deal with arbitrary algorithmic units in a consistent way, is a daunting task. Therefore our proposal is to circumvent this problem not by extending the substitution mechanism, but rather by making the pool of alternative algorithmic building blocks extensible. In doing so, we must provide the optimizer with the required leverage to deal with arbitrary external algorithmic units. The key to this approach is to sustain and exploit the equivalence between algebraic and algorithmic representation of a query plan candidate, and to offer the possibility to switch between these two representations as needed.

In the following, we consider a DBMS as host system that exhibits an extensible pool of relational algorithms. All built-in algorithms of the DBMS represent the initial content of this pool. We use the limited expressiveness of the hosts system’s ERA as a basis for defining the space of possible ERA expressions that may be implemented by custom algorithmic units.

The Access Manager framework allows the addition of such custom algorithmic implementations to the host system's pool of algorithms. As equivalence of different expressions within ERA is prevalent, one algorithmic unit corresponds automatically to an equivalence class of ERA expressions (cf. *Qualitative Equivalence* on page 13). The optimizer handles built-in and subsequently added algorithms equally, i.e. on a high level of abstraction as pure algebraic expressions. During query planning, the optimizer constructs a query plan by combining arbitrary expressions from its ERA basis. Intermittently, the optimizer implements parts of the plan, by applying appropriate algorithmic replacements for individual ERA sub-expressions, using relational algorithms as building blocks from its pool of implementations. All elements in that pool correspond to the language that is defined and limited by the host DBMS's ERA. This is apparently a restriction to the extensibility of the host DBMS, because only algorithms can exist in its pool that are expressible in the DBMS's own ERA, and hence can be formulated in the DBMS's SQL dialect (a consequence of the formerly mentioned equivalence between ERA and SQL expressiveness). In other words, the host DBMS's algorithmic base is fully extensible within the scope of its query language. This is a sensible and comparatively small limitation to the extensibility of DBMSs. And since modern DBMSs actually provide extensible query languages, in form of user-defined data types, user-defined functions, procedures, aggregates, and suchlike, even this limitation becomes quite insignificant.

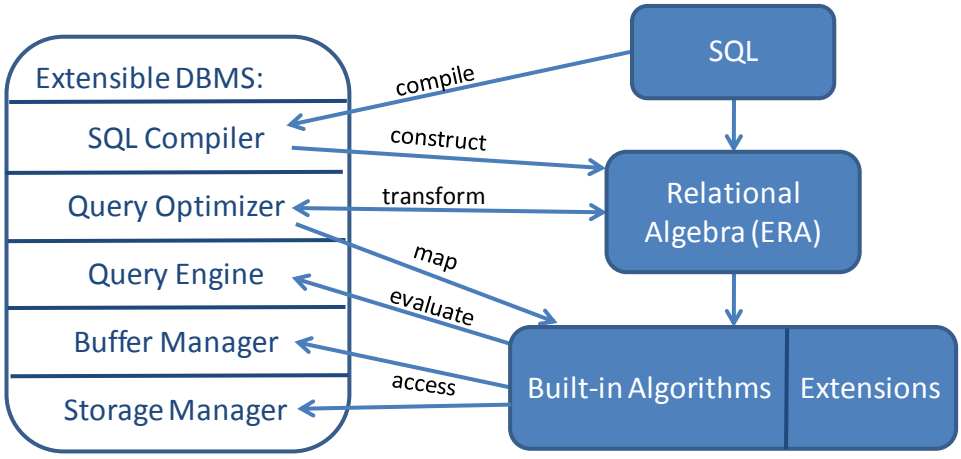


Figure.2 Processing an SQL request in an extensible DBMS. A query formulated in the host DBMS's query language is compiled into the DBMS's internal representation of a query execution plan. This plan is constructed from the host DBMS's limited set of ERA terms. Subsequent transformations of the initial plan strive towards obtaining an equivalent but more efficient QEP. During this process, the optimizer also maps ERA terms onto executable algorithmic building-blocks from its pool of available algorithms. This pool is extensible and may contain customized algorithms that are specialized for peculiarities of the database's application domain. These extensions include relational transformations and data access methods. The extensible system provides identical interfaces for optimization and evaluation of built-in and supplementary algorithms.

Our approach is based on the principle of an iterative, cost-driven selection of implementations for ERA sub-expressions, in order to formulate the final query evaluation plan. Therefore it is not limited to any particular optimization algorithm. We only provide mechanisms to support decisions of a query optimizer, dealing with arbitrary algorithms, without any commitment to when and why the optimizer should employ these mechanisms. In the following we will exemplarily demonstrate how join-optimization, as the central problem of query optimization, can employ arbitrary algorithmic entities for generating optimal plans.

To this end, we make only minimal assumptions concerning the actual optimization mechanisms of the host system. We presume that we deal with the well-known classical cost-driven optimizer model [Sel79], based on dynamic programming and early pruning. The optimizer iteratively constructs partial query evaluation plans in a bottom-up fashion. In construction phase *p* it tries to construct the optimal plan for joining any *p+1* relations. Therefore it is extending its plans for joining *p* relations from the previous iteration by joining one additional relation. To reduce the complexity of the search space for the following iteration, the optimizer will then assess the costs of its plans for joining any *p+1* relations, in order to identify plans with minimal costs. All equivalent but more expensive plans joining the same set of *p+1* relations are pruned from the search space. Pruning is based on the assumption that for finding the optimal plan joining *p+1* relations, it suffices to consider only optimal plans for joining *p* relations and extend them with one additional join (*principle of optimality*). Thus at the end of iteration *p*, the optimizer retains $\binom{n}{p+1}$ partial plans with minimal costs, each joining a different set of *p+1* relations. Due to the iterative construction of these plans, the resulting join graphs are always left-deep trees, as opposed to bushy join trees. Neglecting bushy trees is a risk often accepted by optimizers. Only in rare cases a bushy plan represents the most efficient plan, since joining the results of two joins is usually a blocking operation, incurring complete calculation and expensive temporary storage of intermediate results. Still bushy tree joins are interesting for non-standard query evaluation techniques, in particular in distributed or parallel query evaluation.

While the optimizer proceeds, it will also substitute initial algebraic expressions and map them to algorithmic entities. Often this concept is extended to comprise the idea of *interesting orders*, which was also introduced in [Sel79] for the System R optimizer. For interesting orders, two plans joining the same *p* relations are only equivalent in the pruning step, if they also deliver their results in identical output order. This concept prevents that plans offering an *interesting order*, which could be exploited by subsequent joins or other relational operators,

are prematurely purged in favor of less expensive equivalent plans, exhibiting no such exploitable orders.

The dualistic conception of query plans as algebraic and algorithmic perspectives opens the possibility to switch between these views as necessary. The two views complement each other, while each one is dedicated to its own purpose. The algebraic view defines the domain of the DBMS optimizer, whose competence is to transform ERA terms and eventually provide algorithmic replacements for algebraic terms. This view offers fine-grained interpretation of all involved relational operations and thereby allows maximum flexibility in term optimizations on the limited set of ERA operators. The complete algebraic plan provides an exact specification of the query result, whereas its sub-expressions provide exact definitions of intermediate results. Thereby this representation is perfectly suitable for statistical estimates on such intermediate results, as these statistics represent the crucial input for reliable cost estimation.

The algorithmic view is orthogonal to the algebraic conception. It groups one or more ERA sub-expressions of the query plan into an opaque algorithmic entity. For example, the algebraic representation of a Cartesian product with a following selection will be combined into an algorithmically equivalent nested-loop join. The optimizer may not make any assumptions on operations within an algorithmic entity beyond those that are deducible directly from the algorithm's algebraic counterpart. The most important property of this dualistic conception of query plans is the fact that boundaries of algorithmic units always coincide with complete algebraic sub-expressions. All statistically relevant properties of an intermediate result exchanged between two algorithmic units is specified in detail by its defining algebraic term. Thus, it is always possible to switch back to the algebraic view, in order to determine required statistical information describing input and output relations of algorithmic units. This brings us finally into the position to estimate costs of algorithmic entities based on statistics derived from the algebraic query representation.

We will use the following simple query as running example for illustrating these transformations. It selects tuples from a relation S, having corresponding tuples in R, such that the join predicate $S.s_1 = R.r_1$ is satisfied. The result is projected to $S.s_1$ and finally sorted.

```
SELECT S.s1 FROM S, R WHERE S.s1 = R.r1 ORDER BY S.s1
```

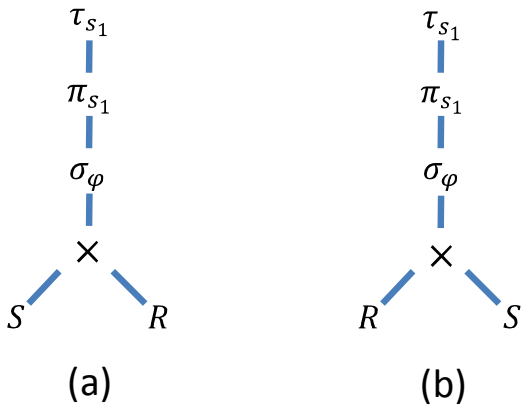


Figure.3 Exemplary decision process during DBMS query optimization. For simplicity, $S.s_1$ and $R.r_1$ shall be the input relations' only attributes. The two plans show the theta-join (equi-join) between S and R over join predicate $\varphi = \{t: t \in S \times R \ \wedge \ S.s_1 = R.r_1\}$ in its primitive algebraic representation. The join result is adjusted to contain only the desired result attribute and is finally sorted according to query specifications. Although the two presented plans are equivalent in ERA representation, as $S \times R \equiv R \times S$, the query optimizer must momentarily consider both alternatives, because depending on the chosen algorithms, available access paths, and effective data distribution, each plan may incur different costs.

2.4.1. Granularity

Grouping ERA operators into algebraic sub-expressions (cf. *2.1.3 Composition* on page 12), which are then implemented by opaque algorithmic entities, raises the issue of granularity: what is the appropriate number of ERA operators to be implemented as one algorithmic entity? In its initial pool of algorithms, the host DBMS will provide standard implementations for every *primitive* ERA operator. The existence of these implementations ensures that the system can always fill the gaps between more bulky algorithmic entities, in order to generate completely computable plans. Additionally, one implementation for every *complex* ERA sub-expression, which cannot be expressed using primitive ERA, is required for sustaining the expressiveness of the system's query language. With this, computability of all expressible queries is guaranteed by the initial algorithmic instrumentation of the host DBMS. Naturally this guarantee holds, even if the algorithmic pool is extended with new algorithms, as it is independent from the characteristics and granularity of additional algorithmic entities. Additionally to these minimum algorithmic requirements, the initial pool contains a substantial amount of efficient implementations of *compound* ERA terms, like specialized join algorithms. These additional algorithms guarantee that the system, even in its initial state, can handle most queries with respectable efficiency. Hence, even with the initial pool of algorithms exists at least one, but generally several ways, to generate a computable query plan from available algorithmic building blocks.

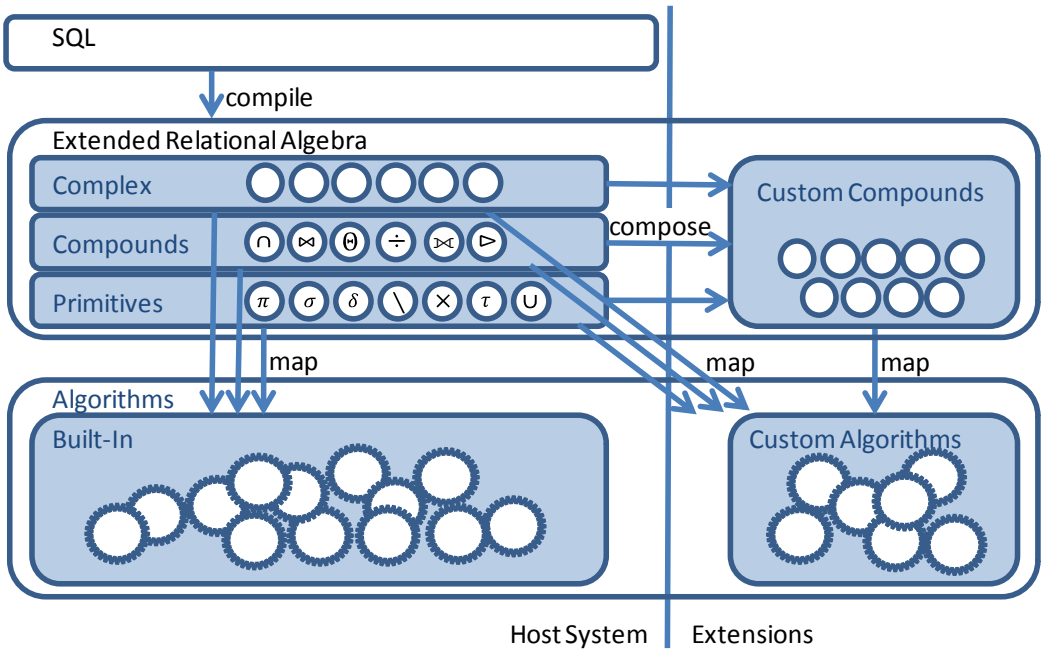


Figure.4 Relationships between algebraic expressions and algorithmic entities. The system provides a fixed set of primitive, compound, and complex ERA expressions. These three sets define the limited expressiveness of the overall system. They also define the algorithmic capabilities of the initial system, as every primitive, compound, and complex ERA term can be mapped to at least one equivalent algorithmic entity in the pool of built-in algorithms. The system is extended by adding alternative implementations for elements of the initial set of algebraic expressions. Alternatively, terms from the initial ERA can be composed into custom compound expressions, to be implemented as custom algorithms.

As a general rule, highly integrated algorithms implementing extensive compound ERA terms are often also highly efficient. On the other hand, overly complex algorithmic units are unlikely to be applied frequently, because the appearance of matching algebraic patterns in plans of ad-hoc queries is less probable for extensive expressions. Only if the probability of specific complex constructs is increased, maybe because a database application makes recurrent use of one particular query pattern, such a sophisticated solution will become interesting. In addition, the composition of several complex algorithmic building blocks into one query evaluation plan is bound to require interspersing smaller operations. These operations fill the remaining gaps between larger building blocks and provide customization of the exchanged data. The necessity of such additional small adaptation operators is likely to partially foil the efficiency benefit intended by using a small number of complex units. The complexity of query optimization is also expected to be influenced by the granularity of algorithms. The search space grows quickly with the number of possible alternatives for replacement of a particular ERA expression. Therefore, an algorithm implementing an extensive replacement patterns is likely to have a multitude of alternatives, each consisting of a patchwork of smaller algorithmic units. From this perspective, the ideal algorithmic pool contains elements of

similar granularity, such that the optimizer has to consider only a small number of algorithmic alternatives for any ERA expression.

Therefore, small algorithmic units with higher potential of reuse and manageable complexity in query optimization are to be preferred over the pushing performance of one single unit to the limit. Small algorithms are also easier to maintain in terms of configuration, as an algorithm may expose various 'knobs' for adjusting it for optimal operation in a given query context. In particular interoperability properties, such as representation compatibility, data flow and order preservation represent such configuration parameters. An extensive algorithmic unit will always make configuration an intricate task, because a higher number of involved relational operators and their potentially complex interactions will certainly hinder selective configurability. In such cases, diversity in algorithmic implementations is better achieved by providing complete algorithmic alternatives for meeting the required flexibility, instead of using configuration.

For substituting an algebraic block with a valid algorithmic implementation, the query optimizer must be able to match these two entities. Whenever a new algorithm is added to the system, it is therefore necessary to provide an expression describing the algorithm's algebraic equivalent. Typically grammar-like production rules are used in extensible DBMSs for accomplishing this sort of mapping. But a commitment to this solution incurs the definition of a complete description language for this single purpose. Instead, we capitalize on the already mentioned equivalence of the fixed set of SQL functionality provided by the host system and the expressiveness of its relational algebra. We therefore propose to use SQL itself as description language for algebraic expressions (cf. Figure.5). The advantage of this solution is that SQL is already in common use by all involved parties. The implementer of an algorithm is certainly familiar with the SQL provided by the host system, and the host system possessed a readily available compiler for translating SQL into an algebraic expression. The latter reveals one aspect of particular elegance provided by this approach: both query and replacement pattern are translated by the same SQL compiler. Hence, two equivalent algebraic expressions originating from compilation are also bound to be highly similar, if not even identical. This property helps alleviating the substitution process.

We outline our proposal of formulating algebraic equivalents in SQL in the following table. Depending on the complexity of its defining SQL expression, one algorithm may correspond to one single relational operator or to a combination of relational operators. The number of input relations in an SQL expression corresponds to the algorithm's arity, i.e. one input

relation is required, and we allow at most two input relations for modeling binary operators. N-ary expressions are equivalent to a cascade of uniform binary patterns. Matching and unification with n-ary algorithmic units is conducted by the query optimizer.

We are using variables for denoting input relations, attribute references, constants, comparison operators, expressions, and predicates. With this, an algebraic expression may be highly generic, expressing that an algorithm implements a class of similar expressions, but they may also be precise for formulating the necessity of exact matches for substitution.

`SELECT * FROM R`	sequential relational access (full table scan)
`SELECT * FROM R UNION SELECT * FROM S`	set UNION with duplicate elimination

`SELECT * FROM R`
`  UNION ALL`
`SELECT * FROM S`
	set UNION without duplicate elimination

`SELECT * FROM R,S WHERE R.a=S.b`	equi-join on one single attribute
`SELECT * FROM R,S WHERE and(R.a = S.a)`	natural Join
`SELECT R.* FROM R,S` `  WHERE and(R.a cmp R.b)`	semi-join on n-ary conjunction of comparisons
`SELECT * FROM A,B WHERE pred(A.a,B.b)`	theta-Join on arbitrary n-ary predicate
`SELECT R.a FROM R` `  GROUP BY R.a HAVING pred(R.b)`	grouping
`SELECT R.a,aggr(R.b) FROM R` `  GROUP BY R.a HAVING pred(R.c)`	grouping with aggregation

Figure.5 Algebraic equivalent of algorithmic entities formulated in SQL. The table shows a rough sketch of an SQL-related syntax for defining algebraic replacement patterns. Variables representing attribute references (e.g. **`R.a`**) shall refer to an arbitrary n-ary projection of actual field references from input stream **`R`**. Normally an SQL compiler would lookup table and field references in the system catalog, for validating the query, determining data-types, and checking privileges, etc. For specification of algebraic equivalents using variables, all these checks are disabled.

The table above demonstrates that substitution of relational accesses, which are the main objective of this work, is trivial, since its defining SQL fragment consists only of the input relation itself, which is not even a relational operation. Hence, we will dispense with the requirement of explicit specification of an algebraic equivalent for accessing relations, and we will devise alternative conventions in due time.

For all non-trivial relational operators, the replacement pattern possesses another important functionality. It decomposes the input of an ERA expression into blocks of attribute refer-

ences, e.g. $R.a \cup R.b \cup R.c = R$ in the *'grouping with aggregation'* example of Figure.5. These blocks may also overlap, e.g. we do not explicitly claim $R.a \cap R.c = \emptyset$, but in this example ERA implicitly demands $R.a \cap R.b = \emptyset$ for sound grouping. Hence, the replacement pattern defines a fixed number of attribute reference blocks on an input stream, such that each block serves for a specific purpose, e.g. $R.a$ attributes are used for grouping, $R.b$ for aggregation and $R.c$ for filtering in this example. These blocks are well-ordered through their order of appearance in the defining SQL fragment. This allows the definition of projection directives for each attribute reference block, such that every attribute from the input stream is unambiguously assigned to well-defined positions in one or more blocks, e.g. attributes from input stream R are mapped to attribute references $(\pi_{R.a}, \pi_{R.b}, \pi_{R.c})$. In general, the input streams of an n-ary relational operator are mapped to input attributes using tions $\left(\left(\pi_j\right)_i\right)$, where each input stream $i = 1, \dots, n$ is decomposed into $j = 1, \dots, m_j$ attribute blocks. This mapping will be important for maintaining attribute references when substituting algebraic expressions with their algorithmic replacements. We formally define input projections:

Definition.5: Input Projection $\boldsymbol{\pi^{in}}$. We call a projection π_i^{in} the i-th input projection of an n-ary algorithmic entity, if it decomposes the entities i-th input stream such that all input attributes are assigned to at least one of m reference blocks within the algorithmic entity's algebraic equivalent, i.e. $\pi_i^{in} = \left(\pi_j\right)_{j=1,\dots,m}$. The entire projection for all input streams is defined as $\pi^{in} = \left(\pi_i^{in}\right)_{i=1,\dots,n}$.

To honor the permanent demand for efficiency so common for DBMSs, we devise an alternative approach for achieving the efficiency of highly integrated algorithmic entities. We will employ representation compatibility as a method to achieve tight coupling of autonomous algorithmic entities by allowing them to exchange information in whatever form seems best for one particular purpose. The goal of this approach is to overcome potential communication bottlenecks between related operators, while conserving the autonomy and general applicability of individual algorithms. The requirement of efficient adaptations of data between algorithmic entities will be addressed by devising a standardized protocol for data exchange.

An algorithm from the DBMS's pool of available algorithmic building blocks is known to implement an expression f. It may be applied in a QEP as implementation for an ERA ex-

pression g only if f and g are *equivalent*. The following Figure.6 shows possible replacements of various granularities based on term equivalence in the exemplary query plan.

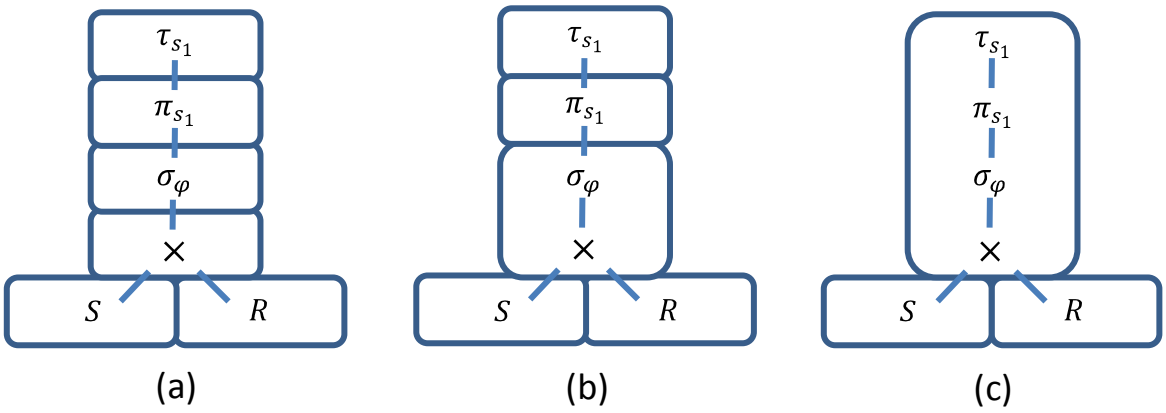


Figure.6 Examples for various granularities of algorithmic replacements. The alternative join sequence $R \times S$ is temporarily omitted. Algorithmic entities are depicted as boxes around ERA sub-terms consisting of one or more ERA operators. For plan (a) only implementations for primitive ERA terms are selected from the initial pool of algorithms. This is the most fine-grained possibility of implementing the QEP. Plan (c) represents a specialized join algorithm, capable of directly producing the final result of this query in one single step. Plan (b) is a balanced implementation that uses a theta-join algorithm for conducting the join. The result is then finalized by projection and sorting according to query specifications, using two primitive algorithmic replacements. Functionally all three substitutions are equivalent, as they implement identical ERA expressions. The optimizer may therefore estimate the costs for each alternative and eventually choose the least expensive implementation for evaluation.

Formally we define mappings between algorithmic entities and ERA expressions as

Definition.6: Algorithmic implementation. We call an algorithmic entity $[f]$ the *algorithmic implementation* of all representatives of equivalence class $\bar{f} \subset ERA$. Correspondingly, we define $[ERA]$ as the set of algorithmic implementations.

Definition.7: Algorithmic equivalent. Algorithmic units $[f]_i$ are *algorithmic equivalents*, providing different implementations for representatives of equivalence class $\bar{f} \subset ERA$.

We extend our ERA notation, allowing intermixture of algorithmic entities and algebraic expressions, in order to describe query plans that have been partially implemented by actual algorithmic entities, e.g. $f \circ g \circ [h]$, or $[f \circ g] \circ h$, etc.

Algorithmic implementation is still rather restrictive, since it requires strict equivalence for permitting substitutions. In particular it prevents the application of an algorithm $[f]$ as replacement for some $g \in ERA$, even if f shows only a minor deviation from g. With query

optimization being a cost-driven procedure, we must take into consideration that an algorithm might be efficient enough to compensate for minor adaptation. For example, let $[f]$ and $[g]$ be two algorithmic candidates to be applied to an input R in replacement for the term f. $[f]$ can be applied directly, while $[g]$ has to be adapted using auxiliary ERA terms p and q, so that $p \circ g \circ q \equiv f$. If cost estimation yields $[f](R) \succ [p] \circ [g] \circ [q](R)$, then $[g]$ is clearly the better replacement, since it outperforms $[f]$ in spite of necessary adaptations. A related problem occurs when replacing an extensive term $f \equiv f_2 \circ f_1$. If it is possible to find one single replacement $[f]$, its efficiency is potentially high, as it offers tight integration of f's complete functionality. On the other hand, the probability of finding an exact match for f declines as f's extensiveness increases, and the optimizer is forced to apply an adapted pattern $[p] \circ [g] \circ [q]$. So the query optimizer will often be confronted with efficiency comparison of extensive but adapted algorithms $[p] \circ [g] \circ [q]$ against exact substitutes of the form $[f_2] \circ [f_1]$.

We soften the algorithmic equivalence claimed above, for allowing a higher degree of tolerance when applying algorithmic entities, by introducing the set of *standard connectors* $\mathfrak{C}^{std} \subset$ ERA. Until its formal definition, we consider it a set of *unary* ERA transformations $\mathcal{C}^{std}$, capable of elementary adaptations for facilitating implementation of ERA terms. $\mathcal{C}^{std}$ will be used for filling the gaps between separate algorithmic implementations. We ensure that generated query plans are executable by assuming that the host system is capable of providing expedient implementations $[\mathcal{C}^{std}]$ for $\mathcal{C}^{std}$-expressions.

Definition.8: Equivalence Configuration. Two algebraic expressions f, g are in *equivalence configuration* $f \overline{\leq} g$, iff equivalence can be achieved by expansion of f, using *standard input connectors* $\mathcal{C}_{I_i}^{std}$ and a *standard output connector* $\mathcal{C}_{O}^{std}$:

$\forall$ n-ary $f, g \in$ ERA:

$f \overline{\leq} g \Leftrightarrow \exists\, \mathcal{C}_{O}^{std}, \mathcal{C}_{I_1}^{std}, \dots, \mathcal{C}_{I_n}^{std} \in$ ERA: $\mathcal{C}_{O}^{std} \circ f \circ \left(\mathcal{C}_{I_1}^{std}, \dots, \mathcal{C}_{I_n}^{std}\right) \equiv g$

As consequence of Definition.6 and Definition.8 follows

Corollary.2: **Equivalent implementation.** Supposing that $f, g \in$ ERA with $f \overline{\leq} g$, and there exists an algorithmic implementation $[f]$ and implementations $\left[\mathcal{C}^{std}\right] \in$ [ERA] for arbitrary $\mathcal{C}^{std}$, then there also exists an *equivalent implementation* of all representatives of $\overline{g}$, because

$g \equiv \mathcal{C}_{O}^{std} \circ f \circ \left(\mathcal{C}_{I_1}^{std}, \dots, \mathcal{C}_{I_n}^{std}\right)$

$$\equiv [\mathcal{C}_O^{std}] \circ [f] \circ ([\mathcal{C}_{I_1}^{std}], \dots, [\mathcal{C}_{I_n}^{std}])$$

Equivalence configuration allows far more permissive replacement of algebraic expressions than algorithmic equivalence does. It defines a partial order on ERA, as it is reflexive, anti-symmetric, and transitive. We can effectively exploit the properties of partial orders for maximizing the number of substitutions allowed by equivalence configuration. Let $\mathrm{ERA}_{\mathrm{i}}^{\overline{\leq}}$ be a partition of ERA, such that for every pair $f, g \in \mathrm{ERA}_{\mathrm{i}}^{\overline{\leq}}$: $f \overline{\leq} g \vee g \overline{\leq} f$. If we choose $h \in \mathrm{ERA}_{\mathrm{i}}^{\overline{\leq}}$ such that $h = Min_{\overline{\leq}}\left(\mathrm{ERA}_{\mathrm{i}}^{\overline{\leq}}\right)$, then $[h]$ is a suitable algorithmic implementation for all elements in $\mathrm{ERA}_{\mathrm{i}}^{\overline{\leq}}$. Under this aspect, equivalence configuration offers remarkable substitution capabilities, if the elements of [ERA] are chosen with careful consideration.

In the following, we will concentrate on identifying a compact subset of *standard connectors* $\mathfrak{C}^{std}$ in ERA, allowing reasonable adaptations and thereby providing general applicability for arbitrary algorithmic units.

2.4.2. Applicability

Each algorithm may have several specific functional and performance-relevant requirements for its application. These requirements must be met in order to make the algorithm work correctly and efficiently. As an example, the merge algorithm in a sort-merge operation assumes that its input streams are sorted on the join-relevant attributes. A straightforward approach would model such requirements as additional sort operations in the ERA replacement pattern of the sort-merge algorithm, making the sort operation an integral part of the algorithm. But this strategy will encourage monolithic algorithmic units and thereby reduce their flexibility and applicability. Additionally, implementing sort operations unconditionally in the algorithm would also hinder global optimization. The integral sort operation cannot be removed afterwards, even if the input of the join already exhibits the desired sort order.

Therefore, we propose to represent such input requirements as directives that are statically attached to each input stream of an algorithmic unit. These directives are invariants that confine the optimizer to apply algorithms only to suitable input. In particular, when inserting an algorithm possessing input requirements, it lies in the optimizer's responsibility to extend the plan in such way, that all directives are initially met. In the sort-merge example, the optimizer must add the appropriate sort operators for satisfying the directives. These sort operations are autonomous operations and may remain subject to continued optimization efforts, allowing the optimizer to refine, move, or remove them as appropriate. The original

directive however has to remain unaltered and intact for ensuring the plan's integrity and correctness.

We informally introduced the set of *standard connectors* $\mathfrak{C}^{std} \subset$ ERA as a set of *unary* ERA transformations $\mathcal{C}^{std}$ and we already used connectors as adapters for compensating minor discrepancies in substitution of algebraic expressions and algorithmic entities. In the following, we will extend the connector concept for describing input directives. Our goal is to devise a set of connectors that allows to *apply* an algorithmic unit $[f_{\mathcal{A}}]$, which is applicable with some input directive $\mathcal{A}$ as a substitute for ERA expression f, such that the query optimizer may satisfy the input directive by inserting a connector $\mathcal{C}_{\mathcal{A}}$, or formally:

$$f \circ g(R) \xrightarrow{apply\,[f]} [f_{\mathcal{A}}] \circ \mathcal{C}_{\mathcal{A}} \circ g(R)$$

While $\mathcal{C}_{\mathcal{A}}$ may undergo revision during further query optimization, the original input directive $\mathcal{A}$ remains in place. We will now survey various use-cases of this concept, before we eventually establish connectors in a formal definition.

For an efficient nested-loop join of relations S and R, it is of vital importance that the costs of finding a join partner in R (inner loop) are sufficiently low. This is achieved by transforming the algorithm's replacement pattern using algebraic equivalence. The original join predicate $\varphi = \{t: t \in S \times R \ \wedge \ S.s_1 = R.r_1 \ \}$ is converted into an equivalent correlated predicate $\varphi' = \{r: r \in R \ \wedge \ \exists s \in S: \ S.s_1 = R.r_1\}$, to be used for a possible direct lookup in the inner loop. Technically this predicate is iteratively evaluated by referencing the outer loop's current value of s when processing the inner loop. Thereby the free variable s of S is bound and becomes a constant for every pass through the inner loop. The correctness of this transformation is a consequence of the algorithm used by the nested-loop join. The nested-loop algorithm ensures that the correlated predicate is exhaustively tested for every pair of s and r. For an arbitrary algorithm, the optimizer cannot know whether such transformations are legal. Therefore the transformation is embedded in the algorithms replacement pattern as an input directive, where it describes this property of the algorithm.

Placing the directive on the inner input branch also allows the removal of the original predicate from the join operation. The optimizer must enforce the directive and thereby guarantees that only matching tuples ever reach the join algorithm by pre-filtering the inner input stream using standard selection operations. With this mechanism, the nested-loop replacement

achieves outsourcing of the selection into an autonomous operator, which can be refined in further optimization steps.

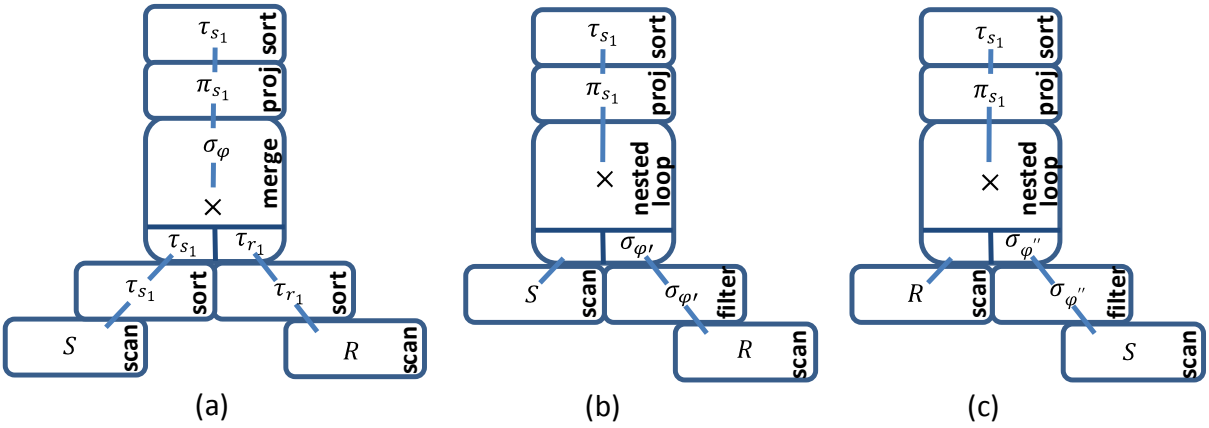


Figure.7 Implanting algorithmic units with applicability requirements. On the left, the join is accomplished using the sort-merge algorithm. This algorithm introduces sort directives for each of its input streams (small boxed sections at the bottom of the merge box). At this stage of query optimization, these directives are satisfied by introducing the corresponding primitive sort operations. The middle plan applies a nested-loop algorithm. The original join predicate φ is transformed into the correlated predicate φ'.This φ' is placed as input directive $\sigma_{\varphi'}$ on the right input stream (inner loop) for the nested-loop algorithm. Again the directive is satisfied by installing the corresponding primitive implementation of the predicate. The rightmost plan is a symmetric variant of (b) using a corresponding φ''. The optimizer has to consider all three variants, because each may exhibit different costs.

Figure.7 resumes our previous example. It shows two alternatives of join algorithms, namely the already mentioned sort-merge and nested-loop joins, which were chosen for this example because both are well-known and conveniently cover many of the aspects to be presented. In plan (a) the optimizer chooses the sort-merge algorithm for implementing the join. Auxiliary sort operations are fitted into the original plan, for supporting the merge algorithm and ensuring correctness. The merge algorithm can rely on receiving sorted input, thus greatly simplifying its task and therefore its own complexity. The nested-loop joins in plans (b) and (c) initiate performance-relevant transformations by decoupling the join predicate and allowing it to be pushed further downwards. This predicate ensures that the nested-loop operation will only receive matching rows in the inner loop, dramatically reducing the complexity of the join operation to mere concatenation of input stream tuples.

Enforcement of such input requirements makes an algorithm universally applicable and limits its implementation complexity. For generally ensuring applicability in an operator tree, the properties demanded from its input must not be overly complex. On the other hand, if its

requirements are too lax, they do not provide the quality of assertions required for a significant reduction of the algorithm's implementation complexity. These conflicting goals have to be carefully balanced. A reasonable approach is to focus on the stream properties established by *basic unary operators* $\{ \pi, \sigma, \delta, \tau \}$. This approach exceptionally elevates the importance of this operator class. They become the ribbon that ties the more advanced components of a query plan together, by accepting the output of the preceding operation and transforming it to become adequate input for the next operation.

In addition to the four basic unary operators $\{ \pi, \sigma, \delta, \tau \}$ providing data interoperability, we also introduce data representation χ as the fifth applicability operator. Any algorithm must be able to interpret its input in order to function correctly. Therefore, two adjacent algorithmic units must agree on a mutual data exchange format. It should be noted, that only the standard data representation χ_{std} is a true prerequisite for the Access Manager framework. If it is the only data representation accepted and produced by all algorithmic units in the [ERA] pool, there is no need to consider representation. We have silently assumed the use of χ_{std} throughout the course of query planning until now.

However, in some cases it is desirable to exchange data in a more appropriate, non-standard way and thereby establish a tight coupling between two independent, consecutive algorithmic units. As example, two operators might want to exchange data as bitmap vectors instead of standard tuples, either because of the compactness of the bitmap representation or because bitmaps are particularly well suited for conducting the respective relational operation, e.g. a cascade of independent bitmap unions and intersections, exchanging and manipulating data in bitmap representation χ_{bm}. An illustrated example covering important aspects of representation is provided in Figure.8 of the following section *2.4.3 Exploitability*. In contrast to the *basic unary operators* $\{ \pi, \sigma, \delta, \tau \}$, χ is not a relational operator and it has no analogon in SQL. Hence, transformation into non-standard representation is neither triggered by SQL queries, nor is it expressible in the SQL-based replacement pattern of an algorithmic unit. These facts actually inhibit the existence of dedicated algorithmic entities for representation transformation in QEPs. Still we admit that some algorithmic entity may express its desire for receiving input in non-standard representation, e.g. by demanding bitmap representation through a corresponding χ_{bm} input directive. For sustained general applicability, an algorithmic unit accepting a non-standard data representation *must* also be capable of accepting and producing data sets in standard data representation χ_{std}. The query optimizer will recog-

nize which representations are supported by an algorithmic entity and the optimizer will also make the decision which representation is ultimately used.

Non-standard representations χ_k introduce the concept of *optional* input requirements, since algorithms may accept both standard and non-standard representation. An expansion of this general concept of optional input requirements upon all basic unary operators offers promising new possibilities. Assume a nested-loop join is receiving its outer-loop input in a sort order that corresponds to the inner relation's physical clustering. Under this precondition, the nested-loop join will actually traverse the inner relation only once, and this single traversal is conducted in a near-sequential manner. This special form of a nested-loop join closely resembles a sort-merge join, but it has the additional ability to 'skip' irrelevant tuples on the inner relation by exerting its direct lookup capabilities. In contrast to this skip-merge join, the standard nested-loop join uses direct lookups for matching join partners to an unsorted outer stream, resulting in fully random accesses to the inner relation. Therefore, the skip-merge join will perform better than the standard nested-loop join, if either the necessary sort order on the outer relation is already present, or sorting is less expensive than fully random access to the inner relation. Still the skip-merge join remains algorithmically identical to the standard nested-loop join, drawing its elevated performance characteristics solely from an *optional* input requirement. This example and alternative input representations clearly demonstrate the necessity for a general concept of optional input requirements for algorithmic entities.

When considering optional input requirements of n-ary algorithms, we also have to contemplate possible dependencies between optional requirements of individual input streams. For example, the skip-merge join may exploit an adequate sort order on its outer stream, but it will only outperform a sort-merge join, which is also applicable under the given circumstances, if the inner relation also provides an appropriate access path allowing direct positioning. Only when both input requirements are satisfied at the same time, the full potential of this algorithm is unleashed. Such dependencies between input requirements are to be expected and need to be integrated into the query planning mechanism. The concept of optional input requirements enables one single implementation of the nested-loop join to operate in several modes, where each mode defines its own combination of input requirements for both input streams. Naturally, each mode results in its own characteristic join complexity and finally in different overall costs.

It is important to distinguish optional input requirements of one single algorithm from alternative implementations of equivalent ERA expressions. As an example, some group algorithm

will work efficiently, if it can exploit an adequate input sort order on the grouping attributes. On the other hand, efficient grouping is also accomplished without an input sort order, if the group operator implements a hash-group algorithm. Evidently, both algorithms implement equivalent ERA expressions. They differ only in their input requirements. Yet they represent algorithms that are substantially distinct. In such cases, it is advisable to add both implementations as separate entities to the pool of algorithms, instead of reverting to optional input requirements for integrating them into the same module.

With data representation and optional input requirements, we conclude our considerations on necessary properties of a set of ERA transformations for providing general interoperability of algorithmic entities. We define them formally in

Definition.9: Sets of Connectors $\mathfrak{C}^{std}$ and $\mathfrak{C}$. A *standard connector* $\mathcal{C}_{\mathcal{S}}^{std}$ is an ERA expression composed of the basic unary ERA operations $(\pi_{\mathcal{S}}, \sigma_{\mathcal{S}}, \delta_{\mathcal{S}}, \tau_{\mathcal{S}}, id_{\chi_{std}})$, operating according to specification $\mathcal{S}$, but strictly in standard representation. The set of all standard connectors is $\mathfrak{C}^{std}$. A *generic connector* has the form $\mathcal{C}_{\mathcal{S}} = (\pi_{\mathcal{S}}, \sigma_{\mathcal{S}}, \delta_{\mathcal{S}}, \tau_{\mathcal{S}}, \chi_{\mathcal{S}})$, and the set of generic connectors is $\mathfrak{C}$. It follows that $\mathfrak{C}^{std} \subset \mathfrak{C} \subset$ ERA.

Connectors represent the mechanism for assembling discrete algorithmic entities into a computable query plan. They facilitate substitution of ERA terms with algorithmic implementations, by providing adaptations within the scope of *equivalence configuration* (cf. Definition.8 on page 30), thereby filling the gaps between algorithmic building blocks. Connectors are also suitable for ensuring interoperability between algorithmic entities, by establishing necessary transformations for meeting input requirements. These input requirements establish general applicability of algorithmic units and reduce the algorithm's complexity by accepting only appropriately preformatted input. Besides reduction of algorithmic complexity, externalization of common tasks (like sorting) promotes global optimization. It allows accomplishing necessary transformation non-locally, as externalized transformations remain subject to ongoing optimization. Such transformations may be pushed downward in the query, plan, permitting to conduct them in an earlier phase of query execution in a cost-effective way or to exploit the same stream property as an input requirement for several consecutive operations. The individual components of a connector $\mathcal{C}_{\mathcal{S}} = (\pi_{\mathcal{S}}, \sigma_{\mathcal{S}}, \delta_{\mathcal{S}}, \tau_{\mathcal{S}}, \chi_{\mathcal{S}})$ establish transformations according to a specification $\mathcal{S}$. The actual sequence of transformations remains nondescript, since there exist dependencies between the connector's individual components, which are induced by the actual specification $\mathcal{S}$. Projection $\pi_{\mathcal{S}}$ maps the original projection of the input

stream to a projection that is opportune for the following algorithm's operation. For example, a join implementation may request that join attributes are uniformly arranged as the first n attributes on all input streams, such that efficient comparison of the tuple prefixes becomes possible. Selection σ_S guarantees that the input meets certain restrictions via adequate pre-filtering. As an example, consider an *m*-way merge-join operation with join predicates on n attributes. If any of the n join fields of *m* input streams is the SQL NULL value, the tuple has definitely no join partner and may be removed from the join's input stream. Such externalized selection operators remain subject to further optimization, in particular selection push-down and recombination with other ERA operators to compound algorithms are options to be considered by the optimizer. As already discussed, sort operations τ_S and similarly distinction δ_S do not have to be implemented by operators that rely on these properties. Instead, the standard implementation of these operators is applied to the input streams if these directives are present. Finally, when representation $\chi_S$is specified as an input requirement of an algorithmic unit, it expresses the algorithm's capability to process input in that representation.

Especially noteworthy is the subset of $\mathfrak{C}^{std}$-connectors, since its exclusion of non-standard representations enables the host system to provide an expedient set of algorithmic modules implementing all $\mathfrak{C}^{std}$-expressions. Therefore $\mathfrak{C}^{std}$ greatly alleviates the composition of computable query plans while $\mathfrak{C}$ permits tight interoperation of consecutive algorithms using non-standard representation. Based on these two fixed sets of connectors, we formulate the general applicability of algorithmic entities:

Definition.10: Applicability. We introduce three distinct qualities of applicability for arbitrary $f, g \in$ ERA:

(1) An arbitrary algorithm implementation $[f]$ is *strictly applicable* in a query evaluation plan, iff it possesses no input requirements, i.e.

$$f \circ g(R) \xrightarrow{apply\,[f]} [f] \circ g(R)$$

(2) $[f]$ is *regularly applicable*, iff its input requirements are satisfied by applying standard connectors $\mathcal{C}^{std} \in \mathfrak{C}^{std}$ to a given relational input. The operator '$\star$' denotes composition in the presence of non-trivial input requirements. The host system is capable of supplying *strictly applicable* implementations $[\mathcal{C}^{std}]$ for every $\mathcal{C}^{std} \in \mathfrak{C}^{std}$.

$$f \circ g(R) \xrightarrow{apply\,[f]} [f] \star g(R) \equiv [f] \circ [\mathcal{C}^{std}] \circ g(R)$$

(3) $[f]$ is *weakly applicable*, iff it has input requirements $\mathcal{C} \in \mathfrak{C} \backslash \mathfrak{C}^{std}$. The host system cannot provide an immediate implementation for $\mathcal{C}$.

$$f \circ g(R) \xrightarrow{apply\,[f]} [f] \star g(R) \equiv [f] \circ \mathcal{C} \circ g(R)$$

Following these conventions, a constant term $R \in$ ERA, representing an input relation, is strictly applicable. It has no input whatsoever and consequently it has no input requirements. For regular applicability, the host system possesses strictly applicable implementations of arbitrary $\mathcal{C}^{std} \in \mathfrak{C}^{std}$. As a consequence, every regularly applicable algorithm operating in standard representation always possesses a complete algorithmic implementation in [ERA]. Implementations for non-standard connectors must be provided as extensions to the [ERA] pool. Otherwise, connectors for weakly applicable algorithms are *currently* not computable, since their algorithmic implementations cannot be supplied by the host system.

A comprehensive way of providing implementations of non-standard connectors emerges when contemplating the set of algorithmic entities provided by the host system for implementing standard connectors $\mathfrak{C}^{std}$. Correspondingly, for arbitrary algorithms operating on some non-standard representation χ_k, sustained applicability comparable to *regular applicability* is achieved by extending [ERA] with a set of *k*-connector implementations $\mathfrak{C}^k$. Whether such an implementation is provided as singular algorithmic entity $[\mathcal{C}^k]$ or as several individual operations, e.g. $[\mathcal{C}^k] \circ [\mathcal{C}^{std}]$ applying some relational operations in std-representation using a standard connector, before switching into k-representation, is functionally irrelevant, but may affect performance. A discussion of such a non-standard connector for bitmap representation (bitmap connector $\mathcal{C}^{bm} \in \mathfrak{C}^{bm}$) is provided in section *5.5 Bitmaps* of the *Proof of Concept* chapter. As an alternative to the provision of custom implementations of non-standard connectors, we will also devise another mechanism for achieving computability of query plans containing weakly applicable algorithms.

All functionality for application of algorithmic entities in query plans is concentrated in the *Apply* function. The *Apply* function is an integral part of every algorithmic implementation. Together with the algorithm's algebraic equivalent, this function provides the basis for facilitating algorithmic substitutions and for dealing with the algorithm's input requirements. The *Apply* function is defined as follows:

Definition.11: Apply function $\mathcal{A}$. For $f, g, h_1, \ldots, h_n \in$ ERA, the algorithmic implementation $[f]$ is applied in a query plan as replacement for an n-ary f with the $\mathcal{A}^{[f]}$ function.

The mapping of attribute positions in input stream tuples to the attribute references in f is provided as vector $\pi^{in} = \left(\pi_i^{in}\right)_{i=1,\dots,n}$. The function parameter $opt \in \mathbb{N}_0$ is an integer choosing an operational mode for f from an enumeration of available modes. Each opt settings effectuates different *optional input requirements* for f. The result of $\mathcal{A}^{[f]}$ are connectors $\left(\mathcal{C}_{\mathcal{A}_i:f_{opt}}\right)_{i=1,\dots,n}$ defining input requirements for each input stream. The formal definition of the $\mathcal{A}^{[f]}$ is:

$$\mathcal{A}^{[f]}\colon \{\pi\}^n \times \mathbb{N}_0 \to \mathfrak{C}^n,$$

$$\mathcal{A}^{[f]}\left(\pi^{in}, opt\right) = \left(\mathcal{C}_{\mathcal{A}_i:f_{opt}}\right)_{i=1,\dots,n}:$$

$$\forall g, h_i \in \text{ERA}: g \circ f \circ (h_1, \dots, h_n) \xrightarrow{\mathcal{A}^{[f]}} g \circ \left[f_{opt}^{\pi^{in}}\right] \star (h_1, \dots, h_n)$$

$$\equiv g \circ \left[f_{opt}^{\pi^{in}}\right] \circ \left(\mathcal{C}_{\mathcal{A}_i:f_{opt}} \circ h_i\right)_{i=1,\dots,n}$$

where $[f_{opt}^{\pi^{in}}]$ is the implementation of f, configured to the current input mapping π^{in} and optional input requirement setting opt.

The $\mathcal{A}^{[f]}$ function is implemented in the algorithmic module $[f]$. It uses input permutations π_i^{in} (cf. Definition.5, page 28) as parameters, defining a mapping of actual attribute positions in input stream i to attribute references in f. By setting the function's parameter $opt = 0$, minimal input requirements $\mathcal{C}_{\mathcal{A}_i:f_0}$ of an algorithmic entity are determined. For modeling optional input requirements, an algorithmic unit may provide additional input configurations $\mathcal{C}_{\mathcal{A}_i:f_{opt}}$, with $opt > 0$. During query planning, the optimizer will iteratively evaluate optional configurations and eventually choose the most suitable option for the given query. Every call to the $\mathcal{A}^{[f]}$-function configures $[f]$ to the parameter set supplied with the function call, i.e. $\mathcal{A}^{[f]}\left(\pi^{in}, opt\right)$ configures $[f]$ to $\left[f_{opt}^{\pi^{in}}\right]$.

We demonstrate substitution in an example, covering the concepts of equivalence configuration, algorithmic implementation and the $\mathcal{A}$-function. Let $f, f' \in$ ERA, both unary with $f \overline{\leq} f'$, and let $[f]$ be a *regularly applicable* implementation of f. Hence, an arbitrary ERA expression f' may be replaced as follows:

$$f'(R) \equiv \mathcal{C}_{O:f}^{std} \circ f \circ \mathcal{C}_{I:f}^{std}(R) \qquad \text{(equivalence configuration)}$$

$$\equiv \mathcal{C}_{O:f}^{std} \circ [f] \star \mathcal{C}_{I:f}^{std}(R) \qquad \text{(algorithmic implementation)}$$

$$\equiv \mathcal{C}^{std}_{O:f} \circ \left[f^{\pi^{in}}_{opt}\right] \circ \mathcal{C}^{std}_{\mathcal{A}:f_{opt}} \circ \mathcal{C}^{std}_{I:f}(R) \qquad (\mathcal{A}^{[f]}\text{-function})$$

$$\equiv \left[\mathcal{C}^{std}_{O:f}\right] \circ \left[f^{\pi^{in}}_{opt}\right] \circ \left[\mathcal{C}^{std}_{\mathcal{A}:f_{opt}}\right] \circ \left[\mathcal{C}^{std}_{I:f}\right](R) \quad (\mathfrak{C}^{std} \text{ implementation})$$

In this example, the substitution mechanism allows a complete implementation of the expression f'. For *weakly applicable* expressions, such general implementations remain unavailable.

Equivalence implementation and applicability introduce several $\mathfrak{C}$-connectors into query plans. By definition, equivalence implementation produces only $\mathcal{C}^{std} \in \mathfrak{C}^{std}$ connectors, while applicability will generate arbitrary $\mathfrak{C}$ connectors. The example above demonstrates how connectors will accumulate between separate algorithmic implementations.

Corollary.3: **Coalescence and Decomposition.** Two arbitrary $\mathcal{C}_1, \mathcal{C}_2 \in \mathfrak{C}$ can be coalesced into one single $\mathcal{C}_3 \in \mathfrak{C}$, such that $\mathcal{C}_1 \circ \mathcal{C}_2 \equiv \mathcal{C}_3$. In particular, for any $\mathcal{C}^{std}_1, \mathcal{C}^{std}_2 \in \mathfrak{C}^{std}$ exists a $\mathcal{C}^{std}_3 \in \mathfrak{C}^{std}$, such that $\mathcal{C}^{std}_1 \circ \mathcal{C}^{std}_2 \equiv \mathcal{C}^{std}_3$. Decomposition describes the inverse operation.

With coalescence, the result from the example above is further simplified to:

$$f'(R) \equiv \left[\mathcal{C}^{std}_{O:f}\right] \circ \left[f^{\pi^{in}}_{opt}\right] \circ \left[\mathcal{C}^{std}_{\mathcal{A}I:f_{opt}}\right](R) \qquad \text{(coalescence)}$$

An efficient implementation of the $\mathcal{A}$-function will specify only configuration parameters $\mathcal{C}_{\mathcal{A}}$ that are minimal for guaranteeing functioning of $[f]$ in mode opt. 'Minimal' means in particular that components $op_{\mathcal{A}} \in \{\pi_{\mathcal{A}}, \sigma_{\mathcal{A}}, \delta_{\mathcal{A}}, \tau_{\mathcal{A}}\}$ of $\mathcal{C}_{\mathcal{A}}$ may remain unspecified, i.e. $op_{\mathcal{A}} = id$. Similarly, any $op_{\mathcal{A}} \neq id$ should also be minimal, e.g. a minimal $\tau_{\mathcal{A}}$ will only specify those attributes, where sort order is essential. Minimal specification of input requirements improves flexibility when combining algorithmic components.

The optimizer has to guarantee that the applicability directives of an algorithmic unit are met. This task also includes finding an adequate and cost-effective arrangement of operations $op_{\mathcal{A}}$ and the resolution of potential interrelations of these operators. For example, it is reasonable to apply $\delta_{\mathcal{A}}$ before sorting in accordance to some $\tau_{\mathcal{A}}$, if the input stream is already presorted and efficient calculation of $\delta_{\mathcal{A}}$ is immediately possible. Duplicate elimination $\delta_{\mathcal{A}}$ will then reduce the amount of data to be sorted in $\tau_{\mathcal{A}}$. But if the input stream is not presorted, it will be smarter to sort according to any given $\tau_{\mathcal{A}}$ before applying $\delta_{\mathcal{A}}$. Especially, if $\tau_{\mathcal{A}}$ is not covering all attributes, the optimizer is also responsible to extend $\tau_{\mathcal{A}}$ to a covering sort order $\tau_{\mathcal{A}'}$, for allowing an efficient application of $\delta_{\mathcal{A}}$.

Global query optimization may achieve that individual $op_{\mathcal{A}}$ can be moved downwards in the operator tree, allowing them to be acquired non-locally and potentially inexpensively. Also repeated exploitation by several consecutive operations using the same input requirements becomes possible.

2.4.3. Exploitability

Additionally to its primary purpose, an algorithmic unit may implement auxiliary ERA operations. Joins, for example, are likely to produce huge result sets. Thus, it is reasonable to reduce the join's result size by incorporating a final projection into the join algorithm, for returning only those attributes that are required for further processing, as opposed to returning the join's designated output attributes, plus the attributes relevant for conducting the join. This final projection is only responsible for conclusively formatting the result and it is not part of the actual algorithm. Its configuration is completely flexible and therefore the projection can be exploited for arranging the output attributes as requested by the following algorithmic unit. This construct allows the join algorithm to project its result in arbitrary ways for cooperating seamlessly and with optimal efficiency with the consecutive algorithmic unit.

Applying two *equivalent implementations* (cf. Corollary.2 on page 30) $[f_1]$ and $[f_2]$ as replacement for one algorithmic sub-expression f, where $f_1 \overline{\leq} f$ and $f_2 \overline{\leq} f$, to identical input is bound to have side-effects on the output, as the algorithms may add, modify or remove interoperability properties of the data stream within the scope allowed by *equivalence configuration*. In our previous examples, the merge-join requires sorted input on the join-relevant attributes. At the same time, the standard implementation of the merge-join will always return its results in that specific order .It is an inherently order-establishing operation, whereas the nested-loop join is at best order preserving, if the input of its outer loop happens to be presorted. Such output order can be exploited, as it might serve as input requirement for consecutive operations. An algorithmic entity will generally function in several *configurable modes*, each mode influencing properties of the output stream. We consider such configuration modes as exploitable parameters of an algorithmic unit.

In order to effectively exploit auxiliary operations and other configurable properties of algorithms, it is worthwhile to examine how to equip an algorithmic unit $[g]$ with enough configurable flexibility for satisfying all input requirements of a consecutive algorithm $[f]$. Such flexibility is not generally feasible and expedient for every $[g]$, but it is a promising approach for constructing query evaluation plans that guarantee optimal interoperability between

arbitrary algorithmic entities. The ideal set of configurable output parameters of $[g]$ is identical to the functionality provided by generic *connectors* $(\pi, \sigma, \delta, \tau, \chi) \in \mathfrak{C}$, which are used for expressing applicability requirements of $[f]$. Such congruence of input directives and exploitable output configuration clearly provides optimal adjustments to data exchanged between consecutive algorithmic units, i.e.

$$[f_{opt}] \circ \mathcal{C}_{\mathcal{A}:f_{opt}} \circ [g] \xrightarrow{exploit\,[g]} [f] \circ [\mathcal{C}_{\mathcal{E}} \circ g]$$

Just like applicability, exploitability is also based on the connector concept, but exploitation is often an imperfect operation, resulting in decomposition of the original connector, because algorithmic units will not generally absorb an arbitrary connector completely. In addition to the already established *coalescence* and *decomposition* of connectors (Corollary.3 on page 40), we will have to contemplate the individual *primitive* components of a connector. We will temporality omit representation and revisit it later for a separate discussion. We consider the remaining primitive elements $op_{\mathcal{A}} \in \{\pi, \sigma, \delta, \tau\}$ of a standard connector $\mathcal{C}_{\mathcal{A}}^{std}$ as required *applicability* specification of some algorithmic unit $[f]$ towards some preceding $[g]$. Individual applicability for each $op_{\mathcal{A}}$ is obtained by installing $op_{\mathcal{A}}$ between the algorithmic entities, as in

$$[f] \circ op_{\mathcal{A}} \circ [g]$$

The desired *applicability* specification $op_{\mathcal{A}}$ of $[f]$ is obtained by configuring the preceding $[g]$ to an *exploitable* specification $op_{\mathcal{E}}$, i.e. $[op_{\mathcal{E}} \circ g]$. This $op_{\mathcal{E}}$ is followed by an additional operation $op_{\mathcal{R}}$ on the result of $[op_{\mathcal{E}} \circ g]$. $op_{\mathcal{R}}$ represents the *rejected* part of $op_{\mathcal{A}}$, that cannot be obtained by configuration of $[g]$, because $[g]$ does not provide the necessary flexibility. For each $op_{\mathcal{A}}$ holds the following equation:

$$op_{\mathcal{A}} \equiv op_{\mathcal{R}} \circ\ op_{\mathcal{E}}$$

The resulting expression is

$$[f] \circ op_{\mathcal{A}} \circ [g]\ \equiv [f] \circ op_{\mathcal{R}} \circ [op_{\mathcal{E}} \circ g]$$

Any of the participating operations may be the identity, i.e. $op = id$. The general relationship between $op_{\mathcal{A}}, op_{\mathcal{E}}, op_{\mathcal{R}}$ is defined as follows:

$$\forall op_{\mathcal{A}}, op_{\mathcal{E}}, op_{\mathcal{R}}, op \in \{\pi, \sigma, \delta, \tau\} \begin{cases} op_{\mathcal{A}} = id \Leftrightarrow \text{no output specification is requested,} \\ \quad \Rightarrow op_{\mathcal{E}} = op_{\mathcal{R}} = id. \quad \text{(trivial case)} \\ op_{\mathcal{E}} = id \Leftrightarrow \text{requested output specification was rejected,} \\ \quad \Rightarrow op_{\mathcal{A}} = op_{\mathcal{R}} \neq id. \quad \text{(worst case)} \\ op_{\mathcal{R}} = id \Leftrightarrow \text{requested output specification was accepted,} \\ \quad \Rightarrow op_{\mathcal{A}} = op_{\mathcal{E}} \neq id. \quad \text{(ideal case)} \\ else \text{ requested output specification was partially accepted,} \\ \quad \Rightarrow id \notin \{op_{\mathcal{A}}, op_{\mathcal{E}}, op_{\mathcal{R}}\} \quad \text{(general case)} \end{cases}$$

A non-standard representation $\chi_{\mathcal{A}}$ can be accepted as $\chi_{\mathcal{E}} = \chi_{\mathcal{A}}$. If $\chi_{\mathcal{A}}$ is rejected, then data is exchanged in standard representation, i.e. $\chi_{\mathcal{E}} = \chi_{std}$. Using non-standard representation $\chi_{\mathcal{E}} = \chi_{\mathcal{A}}$ for exchanging data allows exceptionally tight coupling of consecutive algorithmic units. In this case, it is particularly important that no applicability directives are rejected, i.e. $\forall op_{\mathcal{R}},\ op \in \{\pi, \sigma, \delta, \tau\}: op_{\mathcal{R}} = id$. It is clear, that otherwise transformations $op_{\mathcal{R}}$ would have to operate on non-standard data representations. Since the primary goal of non-standard data exchange is tight coupling of two algorithmic units, such a prerequisite for perfect configuration is consistent and reasonable. Perfect configuration also guarantees that the query plan can be implemented in [ERA], without a non-standard connector $\mathcal{C} \in \mathfrak{C}$. If perfect configuration is not possible, then the system tries to resort to a custom connector $\mathcal{C}$. If no such $\mathcal{C}$ is available in [ERA], the query plan using non-standard representation cannot be implemented. The following example illustrates the exploitability of non-standard representation.

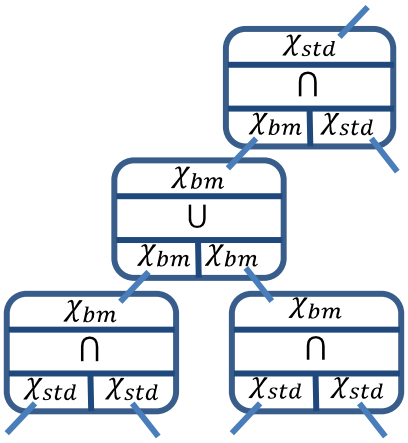


Figure.8 Data representation in a cascade of bitmap operations. The depicted family of customized operators is capable of processing data in bitmap representation χ_{bm}. Applicability ensures that input data is retrieved in convenient χ_{bm} representation wherever possible. Exploitability achieves that data is also delivered in χ_{bm}. Both concepts cooperate in minimizing the number of representation transitions. For general applicability, the algorithms are capable of accepting and producing data in standard representation

χ_{std}, if required. Their flexibility makes it possible to prevent the necessity for explicit operators χ_{bm} for transforming representations.

The isolated examination of decomposition of individual $op \in \{\pi, \sigma, \delta, \tau, \chi\}$ into $op_{\mathcal{A}} \equiv op_{\mathcal{R}} \circ op_{\mathcal{E}}$ is presented here, as it is more comprehensible, but this simplified approach is not practicable during actual query planning. The query optimizer has to consider functional and cost-relevant dependencies between these individual transformations. Functional dependencies arise from the fact, that $op_{\mathcal{E}}$ components are integrated into the preceding algorithmic unit, while $op_{\mathcal{R}}$ transformations are conducted by a connector unit. Hence, chronologically all $op_{\mathcal{E}}$ are conducted before any $op_{\mathcal{R}}$ is applied. As a consequence, it is not legal to exploit some projection $\pi_{\mathcal{E}}$ that removes an attribute, if the same attribute is also referenced in a rejected $\sigma_{\mathcal{R}}$. In this case, the inability to exploit the selection also prevents the exploitation of a projection removing the selection-relevant attribute. Hence, the isolated decomposition $op_{\mathcal{A}} \equiv op_{\mathcal{R}} \circ op_{\mathcal{E}}$ is fundamentally correct, but not every decomposition is also of practical relevance. A procedure for finding valid and cost-effective decompositions is presented in section *2.4.5 Negotiation*. For now, we observe that applicability and exploitability are not configured individually for every $op \in \{\pi, \sigma, \delta, \tau, \chi\}$, but for all components of one given connector at once. We formally define exploitability:

Definition.12: Exploitability. We introduce two distinct qualities of exploitability for arbitrary $[f], [g] \in$ [ERA]:

(1) An algorithmic implementation $[g]$ is *fully exploitable* towards a connector $\mathcal{C}_0$, iff $[g]$ allows integration of $\mathcal{C}_0$, such that

$$[f] \circ \mathcal{C}_0 \circ [g] \xrightarrow{exploit\,[g]} [f] \circ [\mathcal{C}_0 \circ g]$$

(2) An algorithmic implementation $[g]$ is *partially exploitable* towards a connector $\mathcal{C}_0$, iff $[g]$ allows decomposition of $\mathcal{C}_0 \equiv \mathcal{C}_1 \circ \mathcal{C}_2$, such that an implementation $[\mathcal{C}_1]$ exists in [ERA] and $\mathcal{C}_2$ can be integrated into $[g]$:

$$[f] \circ \mathcal{C}_0 \circ [g] \xrightarrow{exploit\,[g]} [f] \circ [\mathcal{C}_1] \circ [\mathcal{C}_2 \circ g]$$

This concept complies with the important properties we identified for exploitation of auxiliary operations within algorithmic implementations. If the preconditions for full exploitability are satisfied, exploitation renders the QEP fragment under consideration in a form that can be implemented immediately with [ERA] assets. This guarantee holds also for partial exploitability, if $\mathcal{C}_0$ is a standard connector. This follows immediately from its decomposition

to $\mathcal{C}_0^{std} \equiv \mathcal{C}_1^{std} \circ \mathcal{C}_2^{std}$, with $[\mathcal{C}_1^{std}] \in$ [ERA]. In addition, exploitability offers an instrument for eliminating non-standard connectors $\mathcal{C}_0 \in \mathfrak{C}$ from a query plan. Ideally, such connectors are integrated completely into a fully exploitable algorithmic unit. In case of partial exploitation, a custom implementation of the non-standard $\mathcal{C}_1$ has to exist in [ERA]. If such an implementation is unavailable, the query optimizer has either to backtrack to an alternative opt setting for applying $[f]$, resulting in an alternative $\mathcal{C}_0'$ or even $\mathcal{C}_0^{std}$. If no alternative opt settings are available, then the QEP must be discarded as not implementable.

Of course, exploitation must be cost-driven, as it is only interesting, if it leads to a reduction of the overall query execution costs. In case of partial exploitation, the optimizer will therefore verify whether the local costs satisfy the local efficiency constraint $[\mathcal{C}_0] \circ [g] \succcurlyeq [\mathcal{C}_1] \circ [\mathcal{C}_2 \circ g]$. The corresponding constraint for full exploitation is $[\mathcal{C}_0] \circ [g] \succcurlyeq [\mathcal{C}_0 \circ g]$. Both cases allow reliable decisions, based solely on local costs. Costs will be examined in greater detail in the discussion in section *2.4.6 Cost Function*.

During query planning, output configuration of an algorithmic entity $[g]$ is conducted by the *Exploit* function, which is an integral part of every algorithmic entity. It is defined as follows:

Definition.13: Exploit function $\mathcal{E}$. Let $f, g \in$ ERA, implemented as $[f] \circ \mathcal{C}_{\mathcal{A}} \circ [g]$. The connector $\mathcal{C}_{\mathcal{A}} \in \mathfrak{C}$ represents coalesced transformations required for substitution and application of $[f]$ and $[g]$. The complexity of such a query evaluation plan can be reduced by integrating functionality from $\mathcal{C}_{\mathcal{A}}$ into $[g]$, using the $\mathcal{E}^{[g]}$ function. The result $\mathcal{C}_{\mathcal{R}:g}$ of the $\mathcal{E}^{[g]}$ function represents the part of $\mathcal{C}_{\mathcal{A}}$ that was rejected by $[g]$:

$$\mathcal{E}^{[g]}: \mathfrak{C} \to \mathfrak{C},$$

$$\mathcal{E}^{[g]}(\mathcal{C}_{\mathcal{A}}) = \mathcal{C}_{\mathcal{R}:g}:$$

$$\forall f, g \in \text{ERA} \begin{cases} \mathcal{C}_{\mathcal{R}:g} = id \;\Leftrightarrow \text{fully exploitable} \\ \qquad \Rightarrow [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \xrightarrow{\mathcal{E}^{[g]}} [f] \circ [\mathcal{C}_{\mathcal{A}} \circ g] \\ \mathcal{C}_{\mathcal{R}:g} = \mathcal{C}_{\mathcal{A}} \;\Leftrightarrow \text{not exploitable} \\ \qquad \Rightarrow [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \xrightarrow{\mathcal{E}^{[g]}} [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \\ \text{else} \qquad \text{partially exploitable} \\ \qquad \Rightarrow [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \xrightarrow{\mathcal{E}^{[g]}} [f] \circ \mathcal{C}_{\mathcal{R}:g} \circ \left[\mathcal{C}_{\mathcal{E}:g} \circ g\right] \end{cases}$$

Here $\mathcal{C}_{\mathcal{E}:g}$ denotes an *exploitable* configuration of $[g]$ towards obtaining the desired specification $\mathcal{C}_{\mathcal{A}}$. In case of full exploitability, $\mathcal{C}_{\mathcal{A}}$ is absorbed completely, while $\mathcal{C}_{\mathcal{R}:g}$ degrades to id

which is eventually omitted. If $[g]$ provides partial exploitability for $\mathcal{C}_{\mathcal{A}}$, the optimizer will arrange for auxiliary transformations $\mathcal{C}_{\mathcal{R}:g}$ for compensating *rejected* configurations. If $[f]$ and $[g]$ exchange data in standard representation, the $\mathcal{E}$-function will yield $\mathcal{C}_{\mathcal{R}:g}^{std} \in \mathfrak{C}^{std}$. Since the host system provides implementations for standard connectors $\left[\mathcal{C}_{\mathcal{R}:g}^{std}\right]$, partial exploitability on standard representation automatically guarantees the existence of an implementation of $f \circ g$ in [ERA]. Full exploitability also guarantees such implementation, regardless of the present representation. The following example illustrates partial exploitation using standard connectors.

$$
\begin{aligned}
f \circ g(R) &\equiv [f] \star [g](R) \\
&\equiv [f] \circ \mathcal{C}_{\mathcal{A}:f}^{std} \circ [g](R) && (\mathcal{A}^{[f]}) \\
&\equiv [f] \circ \mathcal{C}_{\mathcal{R}:g}^{std} \circ \left[\mathcal{C}_{\mathcal{E}:g}^{std} \circ g\right](R) && (\mathcal{E}^{[g]}) \\
&\equiv [f] \circ \left[\mathcal{C}_{\mathcal{R}:g}^{std}\right] \circ \left[\mathcal{C}_{\mathcal{E}:g}^{std} \circ g\right](R)
\end{aligned}
$$

We now proceed with our example by exploring exploitability of our query plan candidates.

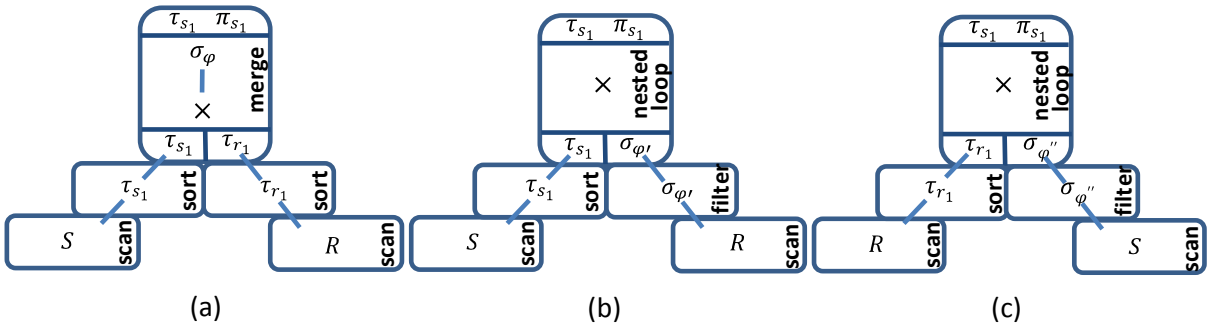


Figure.9 Exploiting sort and projection capabilities of algorithmic units. In all three cases, the optimizer can eliminate the final projections and sort operations by integrating them into the preceding join algorithms. In case of the sort-merge join (a), the algorithm can easily accept the sort specification, since this sort order is already present as a direct consequence of its algorithm's order preservation and its existing input directive τ_{s_1}. A suitable sort operation is already in place for facilitating the merge-join operation. The nested-loop joins (b) and (c) do also accept the sort specification. The nested-loop join is order preserving in the outer loop. Therefore (b) and (c) can provide the requested sort order by propagating the sort specification as a directive to their left input streams. In plan (c) we also exploit $\tau_{s_1} = \tau_{r_1}$, a consequence of the join predicate $s_1 = r_1$. The optimizer has to satisfy the modified input directives in (b) and (c) by installing the corresponding primitive sort operations.

2.4.4. Propagation

Applicability and exploitability can be combined to form the concept of propagation in query planning. Selection push-down and projection push-down are famous examples of such propagation in the algebraic conception of query plans. They translate directly into our algo-

rithmic perspective. The nested-loop joins in Figure.9 illustrate the general idea of such propagation, founded on the principle of preservation of individual stream properties. In Figure.9b the nested-loop join preserves the input order τ_{s_1} of its outer loop. In Figure.9c propagation depends on order preservation τ_{r_1} and on the given join predicate $S.s_1 = R.r_1$. Propagation becomes possible, whenever some algorithmic unit is permeable for stream properties established by unary operators $\{\pi, \sigma, \delta, \tau, \chi\}$. For example, every *streaming* algorithmic entity $[f]$ receiving sorted input will also produce its output in the same sort order. If $[f]$ itself does not rely on this sort order, i.e. $[f]$'s applicability directive specifies $\tau_{\mathcal{A}} = id$, then it is irrelevant whether the sort order is established before or after $[f]$. More generally, if several consecutive streaming algorithmic units $[f_i]$ are capable of preserving τ, then it is possible to establish some required τ at an arbitrary location along the $[f_i]$ chain. This allows the optimizer to propagate τ downwards through the query plan, such that the optimizer may choose, on basis of cost estimations for the various alternatives, at which point τ can be established at minimal costs. We also illustrate this with our example in Figure.9. Here propagation of τ_{s_1} below the order preserving join algorithms is interesting, if the join result has a higher cardinality (and size) than the input stream determining the result order (i.e. outer loop in the nested-loop case). The goal is cost reduction by sorting less data. If the join reduces the cardinality, one should contemplate applying the sort operation after joining. But even in this case, it may still pay off to propagate the sort operation downwards, if the sort order of the larger set below the join can be established inexpensively, e.g. by using an already existing order (cf. Figure.10a), or if early sorting allows using more efficient join algorithms, like the skip-merge algorithm in cases (b) and (c). Once the preconditions for propagation are met, the ultimate decision for employing propagation is made by the optimizer on basis of cost estimations.

Compared to exploitability, propagation represents an additional means for decomposing a *connector* $(\pi, \sigma, \delta, \tau, \chi) \in \mathfrak{C}$ ensuring applicability between two consecutive algorithmic units. When considering

$$[f] \circ op_{\mathcal{A}} \circ [g]$$

with $op \in \{\pi, \sigma, \delta, \tau\}$ representing one individual connector component, then we can describe this decomposition for each op in analogy to the corresponding exploitability equation above as:

$$op_{\mathcal{A}} = op_{\mathcal{R}} \circ op_{\mathcal{E}} \circ op_{\mathcal{P}}$$

Here $op_{\mathcal{P}}$ is the *propagated* part of the original connector $op_{\mathcal{A}}$. Each term on the right hand side of this equation describes transformations towards $op_{\mathcal{A}}$ at different locations relative to the currently examined algorithm $[g]$. $op_{\mathcal{P}}$ is a transformation applied on the algorithm's input stream, $op_{\mathcal{E}}$ represents the algorithm's actual exploitability towards $op_{\mathcal{A}}$, and $op_{\mathcal{R}}$ is the rejected part that has to compensate for lack of propagation and exploitability of $[g]$, i.e.

$$[f] \circ op_{\mathcal{A}} \circ [g] \equiv [f] \circ op_{\mathcal{R}} \circ [op_{\mathcal{E}} \circ g] \circ op_{\mathcal{P}}$$

Again, representation has to be treated separately. If a non-standard representation $\chi_{\mathcal{A}}$ is rejected, then data is exchanged in standard representation. As a general recommendation, any algorithmic unit capable of accepting a transition to non-standard representation $\chi_{\mathcal{A}}$ as $\chi_{\mathcal{E}} = \chi_{\mathcal{A}}$, should also contemplate permitting its propagation as $\chi_{\mathcal{P}} = \chi_{\mathcal{A}}$. Such propagation is attractive from the cost perspective, as each transition of representation inevitably incurs costs. The ability of an algorithmic unit to propagate non-standard representation effectively supports reducing the number of representation transitions in a QEP. It also opens additional opportunities in query optimization for finding ideal locations for switching between representations.

We extend the concept of exploitability for a formal definition of propagation

Definition.14: Propagation. We introduce two distinct qualities of propagation for arbitrary $[f], [g], [h] \in$ [ERA]:

(1) A connector $\mathcal{C}_0$ is *fully propagatable* through an algorithmic implementation $[g]$, iff a decomposition $\mathcal{C}_0 \equiv \mathcal{C}_1 \circ \mathcal{C}_2$ exists, such that

$$[f] \circ \mathcal{C}_0 \circ [g] \circ [h] \xrightarrow{propagate\ [g]} [f] \circ [\mathcal{C}_1 \circ g] \star [h]$$
$$\equiv [f] \circ [\mathcal{C}_1 \circ g] \circ \mathcal{C}_2 \circ [h]$$

(2) A connector $\mathcal{C}_0$ is *partially propagatable* through an algorithmic implementation $[g]$, iff $[g]$ allows decomposition of $\mathcal{C}_0 \equiv \mathcal{C}_1 \circ \mathcal{C}_2 \circ \mathcal{C}_3$, such that an implementation $[\mathcal{C}_1]$ exists in [ERA] and $\mathcal{C}_2$ can be integrated into $[g]$, i.e.

$$[f] \circ \mathcal{C}_0 \circ [g] \circ [h] \xrightarrow{propagate\ [g]} [f] \circ [\mathcal{C}_1] \circ [\mathcal{C}_2 \circ g] \star [h]$$
$$\equiv [f] \circ [\mathcal{C}_1] \circ [\mathcal{C}_2 \circ g] \circ \mathcal{C}_3 \circ [h]$$

Conceptually propagation is very similar to exploitation. While exploitation provides strictly local integration of connectors into the preceding algorithmic unit, propagation prepares relocation of $\mathfrak{C}$-transformations by propagating them downwards through algorithmic units. The '$\star$' operators established by propagation indicate that $[g]$ was modified by propagation, resulting in modified application requirements. If $[g]$'s applicability requirements remain unaffected by propagation of some $\mathcal{C}$, then propagation is exactly equivalent to exploitation.

Propagation of $\mathfrak{C}$-transformations over algorithmic entities is provided by the *Propagate* function. It is an integral part of every algorithmic implementation and it is defined as follows:

Definition.15: Propagate function $\mathcal{P}$. Let $f, g, h \in$ ERA, implemented as $[f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \circ [h]$. The connector $\mathcal{C}_{\mathcal{A}} \in \mathfrak{C}$ represents coalesced transformations required for substitution and application of $[f]$ and $[g]$. The complexity of such a query evaluation plan can be reduced by propagating functionality from $\mathcal{C}_{\mathcal{A}}$ through $[g]$, using the function $\mathcal{P}^{[g]}$, where $\mathcal{C}_{\mathcal{R}:g}$ represents the part of $\mathcal{C}_{\mathcal{A}}$ that was rejected by $[g]$:

$$\mathcal{P}^{[g]}\colon \mathfrak{C} \to \mathfrak{C},$$

$$\mathcal{P}^{[g]}(\mathcal{C}_{\mathcal{A}}) = \mathcal{C}_{\mathcal{R}:g}\colon$$

$$\forall f, g, h \in \text{ERA} \begin{cases} \mathcal{C}_{\mathcal{R}:g} = id \;\Leftrightarrow \text{fully propagatable} \\ \quad \Longrightarrow [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \circ [h] \xrightarrow{\mathcal{P}^{[g]}} [f] \circ \left[\mathcal{C}_{\mathcal{E}:g} \circ g\right] \star [h] \\ \mathcal{C}_{\mathcal{R}:g} = \mathcal{C}_{\mathcal{A}} \;\Leftrightarrow \text{not propagatable} \\ \quad \Longrightarrow [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \circ [h] \xrightarrow{\mathcal{P}^{[g]}} [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \circ [h] \\ \text{else} \qquad \text{partially propagatable} \\ \quad \Longrightarrow [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \circ [h] \xrightarrow{\mathcal{P}^{[g]}} [f] \circ \mathcal{C}_{\mathcal{R}:g} \circ \left[\mathcal{C}_{\mathcal{E}:g} \circ g\right] \star [h] \end{cases}$$

The following example demonstrates the combined effects of application, propagation, and exploitation:

$$\begin{aligned} f \circ g \circ h(R) &\equiv [f] \star [g] \star [h](R) \\ &\equiv [f] \circ \mathcal{C}^{std}_{\mathcal{A}:f} \circ [g] \star [h](R) && (\mathcal{A}^{[f]}) \\ &\equiv [f] \circ \mathcal{C}^{std}_{\mathcal{R}:g} \circ \left[\mathcal{C}^{std}_{\mathcal{E}:g} \circ g\right] \star [h](R) && (\mathcal{P}^{[g]}) \\ &\equiv [f] \circ \mathcal{C}^{std}_{\mathcal{R}:g} \circ \left[\mathcal{C}^{std}_{\mathcal{E}:g} \circ g\right] \circ \mathcal{C}^{std}_{\mathcal{A}:g} \circ [h](R) && (\mathcal{A}^{[g]}) \\ &\equiv [f] \circ \mathcal{C}^{std}_{\mathcal{R}:g} \circ \left[\mathcal{C}^{std}_{\mathcal{E}:g} \circ g\right] \circ \mathcal{C}^{std}_{\mathcal{R}:h} \circ \left[\mathcal{C}^{std}_{\mathcal{E}:h} \circ h\right](R) && (\mathcal{E}^{[h]}) \end{aligned}$$

$$\equiv [f] \circ \left[\mathcal{C}^{std}_{\mathcal{R}:g}\right] \circ \left[\mathcal{C}^{std}_{\mathcal{E}:g} \circ g\right] \circ \left[\mathcal{C}^{std}_{\mathcal{R}:h}\right] \circ \left[\mathcal{C}^{std}_{\mathcal{E}:h} \circ h\right](R)$$

This also exemplifies how the propagated part $\mathcal{C}_{\mathcal{P}}$ of a connector $\mathcal{C}_{\mathcal{A}}$ never physically appears in a query plan. It is immediately integrated into the modified application directives of the algorithmic unit conducting the propagation, which is indicated by the special composition operator '$\star$'.

Propagation represents an iterative process which, when used in turns with application and exploitation of algorithmic entities, is capable of moving transformations downwards through a query plan, in order to determine the optimal location for an operation. Propagation naturally terminates when it reaches the leaf nodes of a QEP, or when an algorithmic unit does not permit further propagation. In the latter case, propagation degrades to plain exploitation. In addition, the optimizer may deliberately stop propagation at any time. Therefore, it uses explicit exploitation after several propagation steps, if the QEP requires local consolidation.

In summary applicability, exploitability, and propagation build a robust and simple, yet efficient and powerful mechanism allowing the query optimizer to find the optimal configuration for all participating algorithms in a query plan, without knowing the internal workings of the algorithms. The query plan can directly benefit from the configurable degree of freedom provided by algorithmic units. Still every implementation of a relational algorithm may choose independently how much flexibility it wants to provide. A higher flexibility will allow better integration with adjacent algorithms, but higher flexibility will usually involve higher implementation complexity. Conversely, missing flexibility of algorithmic units is automatically compensated by applying the necessary customizations in form of auxiliary connector units. The implementer of a relational algorithm may reevaluate this trade-off in iterative development cycles, while the mechanisms for negotiating configurations will automatically adapt to the present capabilities of the algorithmic entity.

Propagation, as the only non-local instrument of query optimization, is strictly directed downwards. Hence, the optimization process will always terminate. But, without further measures, each attempt at propagation will potentially trigger a complete recursion, with devastating impact on the complexity of our approach during query planning. This issue will be addressed in the following section. But first, we complete our example.

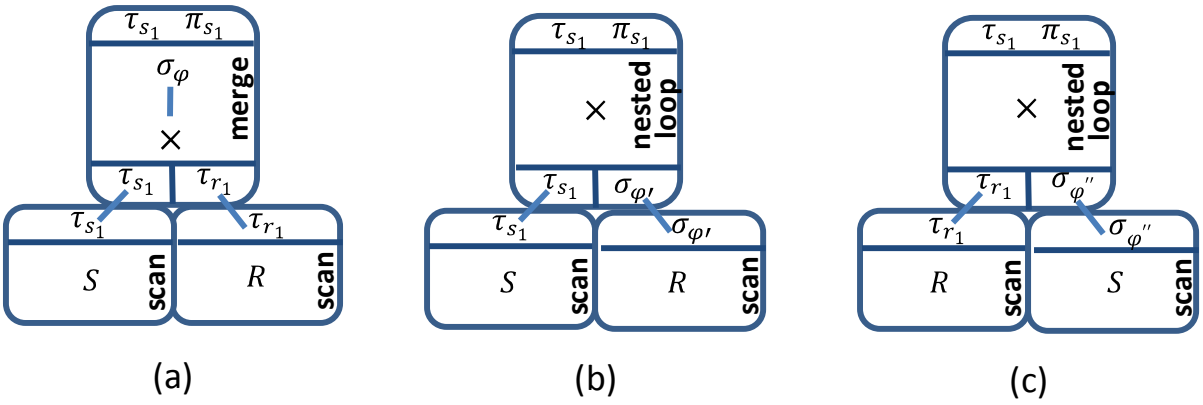


Figure.10 Propagation of configuration parameters. *S* and *R* shall be physically stored in B-trees, offering direct access capabilities and also exploitable storage orders τ_{s_1} and τ_{r_1}. Although sorting was the last operation in the initial query plans, propagation makes it the first transformation to be applied in the final QEPs. It is pushed directly into relational access of *S* and *R*, where it is implemented inexpensively as non-blocking ordered relational scan. Similarly for the nested-loop joins (b) and (c), the original join predicate φ is transformed into the correlated predicate φ', and φ'' resp., and propagated for exploiting efficient direct access to join partners in the inner loops.

As result of successful configuration of algorithmic entities, all plan alternatives are now consisting only of three algorithmic units each. The final task of choosing the most cost-effective plan is left to the optimizer and the cost functions of the algorithms.

2.4.5. Negotiation

Negotiation is the configuration process between two independent algorithmic units. Its purpose is to ensure efficient cooperation by adjusting exploitable configuration properties of one unit in order to match the applicability requirements of a consecutive algorithm. Negotiation operates on connectors $\mathcal{C}$, or rather on their primitive components $op \in \{\pi, \sigma, \delta, \tau, \chi\}$, which are established between individual algorithmic units. These connectors originate either from necessary adaptations resulting from *equivalent implementation* (cf. Corollary.2 on page 30), or they implement additional *applicability directives* of algorithmic units. The goal of negotiation is to configure participating algorithmic units such that connectors are optimally arranged throughout a query plan and potentially absorbed by algorithmic units implementing the necessary transformation as an auxiliary, configurable functionality. Although the optimizer can spot such auxiliary *op*s in an algorithm by examining its ERA replacement pattern, it can neither assess the extent of possible configurations, nor is it possible to estimate the implication such configurations will have on the overall behavior of the algorithm. Even worse, our example demonstrated the nested-loop algorithm's capability for retaining exploitable sort orders, originating not from an internal sort operation, but from order preservation

by propagating the requirement to one of its input streams. Hence, the existence of an operator op in the replacement pattern is neither a sufficient prerequisite for configurability, nor is it necessary.

Configurability depends solely on the implementation of the algorithm. For finding an adequate configuration for an algorithmic unit, each unit must be able to negotiate its own configuration within the scope of its capabilities. In addition, it must be able to identify configuration properties that are suitable for propagation to its various input streams. Finally, it has to provide a cost function that allows the DBMS optimizer to assess the quality of one particular configuration.

Negotiation improves interoperability between two algorithmic entities exchanging data via a connector unit, e.g. $[f] \circ \mathcal{C}_{\mathcal{A}} \circ [g]$. The connector $\mathcal{C}_{\mathcal{A}}$ is conducting necessary adaptations specified as a set of configuration parameters $\{\pi_{\mathcal{A}}, \sigma_{\mathcal{A}}, \delta_{\mathcal{A}}, \tau_{\mathcal{A}}, \chi_{\mathcal{A}}\}$. In order to simplify or even eliminate $\mathcal{C}_{\mathcal{A}}$, the negotiation process tries to exploit functionality provided by $[g]$ for implementing $\mathcal{C}_{\mathcal{A}}$. Exploitation shall map $\mathcal{C}_{\mathcal{A}}$ to the *cost-optimal* configuration $\{\pi_{\mathcal{E}}, \sigma_{\mathcal{E}}, \delta_{\mathcal{E}}, \tau_{\mathcal{E}}, \chi_{\mathcal{E}}\}$ to be integrated into $[g]$. Alternatively, functionality from $\mathcal{C}_{\mathcal{A}}$ shall be propagated over $[g]$, if $[g]$ will preserve the propagated transformations $\{\pi_{\mathcal{P}}, \sigma_{\mathcal{P}}, \delta_{\mathcal{P}}, \tau_{\mathcal{P}}, \chi_{\mathcal{P}}\}$ established by such early adaptations and their propagation is cost-effective. Finally, a set of rejected configuration transformations $\{\pi_{\mathcal{R}}, \sigma_{\mathcal{R}}, \delta_{\mathcal{R}}, \tau_{\mathcal{R}}, \chi_{\mathcal{R}}\}$ shall compensate possible mismatches, such that for any $op \in \{\pi, \sigma, \delta, \tau, \chi\}$ the following equation holds:

$$op_{\mathcal{A}} = op_{\mathcal{R}} \circ op_{\mathcal{E}} \circ op_{\mathcal{P}}$$

Naturally, there exist applicability specifications $op_{\mathcal{A}}$ without efficient decompositions. In this case, necessary transformations shall remain within the original connector, such that $op_{\mathcal{R}} = op_{\mathcal{A}}$. We call a decomposition of $op_{\mathcal{A}}$ *efficient*, if its implementation satisfies the general inequality

$$cost([op_{\mathcal{A}}]) \geq cost([op_{\mathcal{R}}] \circ [op_{\mathcal{E}}] \circ [op_{\mathcal{P}}])$$

In this consideration, interrelations between different ops are not yet represented adequately. Again, we temporarily omit representation χ, because this is negotiated independently. The remaining four operators $\{\pi, \sigma, \delta, \tau\}$ are largely orthogonal in their functionality, but there are several dependencies as depicted by the following Figure.11. In order to devise a suitable strategy for finding efficient configurations, it is worthwhile to analyze the rules of interrelations between configurable operators.

The exploitable operators $op_{\mathcal{E}}$ are integral parts of $[g]$, while propagated configurations $op_{\mathcal{P}}$ are applied before, and the supplemental transformations $op_{\mathcal{R}}$ are installed afterwards. Although the elements of each separate set $\{\pi_{\mathcal{P}}, \sigma_{\mathcal{P}}, \delta_{\mathcal{P}}, \tau_{\mathcal{P}}\}, \{\pi_{\mathcal{E}}, \sigma_{\mathcal{E}}, \delta_{\mathcal{E}}, \tau_{\mathcal{E}}\}$, and $\{\pi_{\mathcal{R}}, \sigma_{\mathcal{R}}, \delta_{\mathcal{R}}, \tau_{\mathcal{R}}\}$ are applied in a currently nondescript order, elements from different sets are never interweaved, i.e. all $op_{\mathcal{P}}$ are applied before any $op_{\mathcal{E}}$, which in turn always precede any $op_{\mathcal{R}}$.

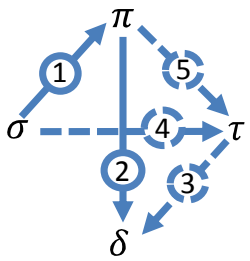


Figure.11 Dependencies of configuration parameters. Configuration *must* comply with functional dependencies, represented as solid lines: (1) Selection σ_{φ} before projection π avoids removal of attributes required by selection. (2) Distinction δ follows after projection π is completed. Non-functional dependencies are efficiency considerations marked with dashed lines: (3) Distinction δ should be applied after a suitable order is available. Sorting τ should be preceded by a reduction of cardinality and arity through selection σ (4) and projection π (5).

Following the dependencies described in Figure.11, we can now devise general rules for finding efficient configurations:

(1) We start considering selection σ, as it has no dependencies on other configuration parameters. In accordance to selection push-down, we break up the predicate of $\sigma_{\mathcal{A}}$ and apply selections as early as possible, i.e. try to propagate maximum selectivity as $\sigma_{\mathcal{P}}$. For the remaining predicate, we use exploitability for installing $\sigma_{\mathcal{E}}$, if possible. After completion of this step the settings for $\sigma_{\mathcal{P}}$ and $\sigma_{\mathcal{E}}$ are not modified anymore. Consequently, $\sigma_{\mathcal{R}}$ also remains fixed in the following steps.

(2) Then we apply the desired projection $\pi_{\mathcal{A}}$ reducing the breadth of the input set. $\pi_{\mathcal{P}}$ will remove attributes that were referenced in $\sigma_{\mathcal{P}}$, but are not required afterwards. We make sure not to remove attribute references that are required for post-filtering the rest-predicate in $\sigma_{\mathcal{E}}$ or $\sigma_{\mathcal{R}}$ (dependency 1). Analogously $\pi_{\mathcal{E}}$ removes attribute references that have become obsolete after application of $\sigma_{\mathcal{E}}$. In this step, projections removing attributes are relevant for dependency considerations, while mere attribute rearrangements are not significant and are ignored. And even if projections reduce the number of attributes, they are only performance-relevant, if they are followed by a blocking operation, in particular by a conventional sort operation that takes advantage of the reduced breadth (cf.

Figure.11, dependency 5). For streaming operators, the tuple breadth has no impact on the costs, since storage costs can be neglected for streaming operations, and the projection can be safely postponed to the last possible point in time, i.e. $\pi_{\mathcal{E}}$ or even $\pi_{\mathcal{R}}$, without performance penalty. At this point configurations of selection and projection are completed.

(3) Now we attend to propagation and exploitation of the sort order $\tau_{\mathcal{A}}$. Propagation is particularly attractive, if it is possible to establish $\tau_{\mathcal{A}}$ by means of a more efficient method than conventional sorting. Such propagation is also reasonable, if it achieves only a prefix of the lexicographical order specified by $\tau_{\mathcal{A}}$. In this case, $\tau_{\mathcal{A}}$ is fully established using a subsequent sort operation in form of an inexpensive partial sort operation $\tau_{\mathcal{E}} = \tau_{\mathcal{A}}$ or $\tau_{\mathcal{R}} = \tau_{\mathcal{A}}$, which will also benefit from further reductions of cardinality and arity by earlier application of predicates and projections (dependencies 4 and 5).

(4) Finally, we address duplicate eliminations $\delta_{\mathcal{E}}$. Duplicate elimination should be postponed until the final projection is established (Figure.11, dependency 2) and a suitable sort order is available (dependency 3). If necessary, available sort orders have to be extended to cover all attributes for supporting efficient duplicate elimination.

Our conception of query optimization in the presence of opaque and configurable algorithmic units assumed the existence of a system-inherent query optimization component, providing and controlling the search strategy for generating efficient query execution plans on a global scale. One of the optimizer's essential tasks is limitation of complexity and costs of query optimization. Therefore, external algorithmic entities may only contribute in a temporally and spatially limited way to this procedure. The presented methodology of negotiation, based on applicability, exploitability and propagation is sufficiently flexible for devising optimization algorithms dealing with configurable algorithmic entities. At the same time, each phase of negotiation operates strictly locally and on a comparatively low level of complexity, leaving overall control to the host system's optimization component.

In the following, we will outline a modified query optimization strategy, suitable for general cost-driven optimization based on negotiation. In addition, we will sketch the rule-based optimization strategy, which is used in the Transbase prototype implementation. This shall demonstrate that the proposed methodology is generally suitable for arbitrary optimization strategies, and only moderate modifications to the original optimization algorithms are required.

First we examine a cost-based optimization model using dynamic programming for building alternative plans simultaneously in a bottom-up fashion. In optimization step *p* of a query joining *n* relations, this model considers query plan fragments joining *p*+1 individual relations. This is done by extending plans joining *p* relations, retained from the previous iteration, with one additional join. Up to this point, cost-based optimization is strictly conforming to its original proposal in [Sel79]. The fundamental difference when using configurable algorithmic units results from the following observations:

> (1) a partial plan joining *p+1* relation cannot be fully configured, since the configuration of its topmost operator is depending on applicability requirements of its subsequent operator, which is still unavailable in optimization step *p*.
>
> (2) the missing configuration of the topmost operator also implies that the plan fragment's final output sort order is still unspecified. This prevents classification of the plan fragment with respect to *interesting orders*.
>
> (3) an incompletely configured plan fragment cannot be associated with costs, thus cost-based pruning is inhibited.

These issues are circumvented as demonstrated in Figure.12. After optimization *phase one*, the pool of query candidates contains *j* plans for joining relations A and B, each using a different one from *j* available join methods. The pool contains a total of $j \cdot \binom{n}{2} \cdot 2!$ plans for joining every *ordered* pair of *n* relations, using one of *j* join methods. The classical optimization algorithm will now initiate a pruning phase, which will reduce the pool size to a lower boundary of $\binom{n}{2}$ plans. If *interesting orders* are present, then the number of plan candidates will be correspondingly higher. The computational complexity correlates with the total number of necessary cost estimations, which is equal to the original number of plan candidates $j \cdot \binom{n}{2} \cdot 2!$. In our case however, the plans resulting from phase one are still incompletely configured (marked white in Figure.12), since the join operator's *applicability* requirements are met, but the topmost join operator is not yet configured for *exploitation* by its still missing consecutive operator. As a consequence, pruning at this stage is not possible (cf. reasons (1)-(3) above) and optimization must momentarily retain all plan candidates.

In *phase two*, each plan is extended for joining one additional relation (depicted white in Figure.12, phase 2). By enumerating all possible join methods, we can generate j^2 plans for joining relations A, B and C, resulting in $j^2 \cdot \binom{n}{3} \cdot 3!$ plans for joining any three of *n* relations.

At this point, it becomes possible to complete the configuration of the joins built in the previous phase (now marked light blue), as the necessary consecutive join was just installed. The *principle of optimality*, on which cost-based query optimization using dynamic programming is founded, claims that an optimal left-deep plan joining p+1 relations is built from an optimal plan joining p relations, extended by one additional join. Consequentially, we may now apply cost estimation and subsequent pruning to the fully configured plan fragments of phase two (light blue). Cost estimation will identify the one optimal plan for joining relations A and B (although interesting order may introduce additional alternatives). This plan fragment also specifies the optimal join-method and join-sequence for joining A and B. According to the *principle of optimality*, it now becomes possible to discard all plan candidates containing sub-optimal strategies for joining A and B. This leaves only $j \cdot (n-2)$ plan candidates, joining the result of the optimal join of A and B with one arbitrary third relation, using all available join methods. After pruning has processed all two-way joins from phase one, the pool contains a lower boundary of $\binom{n}{2}$ optimal plan fragments joining any two relations (plus interesting orders). Correspondingly, a minimum of $j \cdot (n-2)\binom{n}{2}$ plan candidates have to be retained by the end of phase two. In general, the storage complexity of the classical algorithm at the end of phase p is at least $\binom{n}{p+1}$, while our variant using configuration and retarded pruning stores a minimum of $j \cdot (n-p)\binom{n}{p}$ plans. With $j \cdot (n-p)\binom{n}{p} = j \cdot (p+1)\binom{n}{p+1}$ follows that the storage complexity of retarded pruning is $j \cdot (p+1)$ times higher than that of the classical algorithm, reaching its maximum at $p = \left\lfloor \frac{n}{2} \right\rfloor$. The same factor also applies when comparing computational complexities, as the computational complexity of phase p depends directly onto the number of retained plans from the previous phase. As we anticipate small numbers of j, the total complexity increases by a factor of $\mathcal{O}(n)$. This is comparable to the increased complexity introduced by *interesting orders* and therefore similarly acceptable.

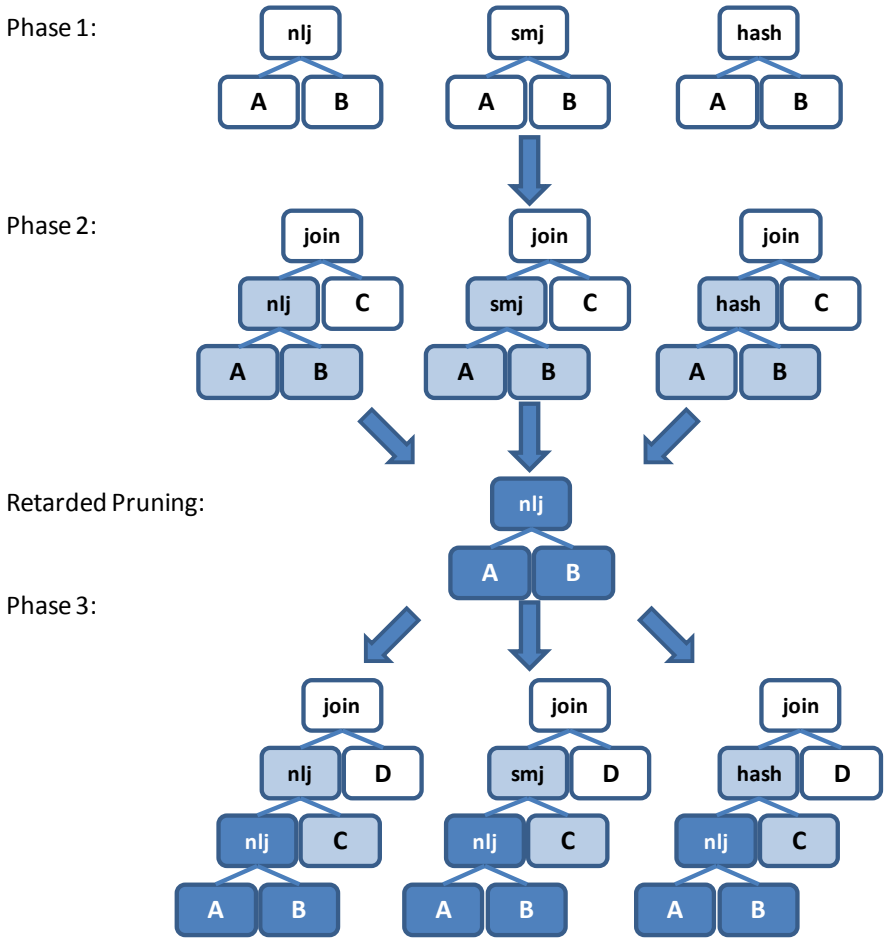


Figure.12 Cost-based join optimization with configuration. Exemplified join optimization in a system providing $j = 3$ different methods for joining n relations. The graph depicts only a small fraction of the plan candidates of each join phase. Pruning after phase one is impossible, but the principle of optimality allows retarded pruning in the subsequent phases. In this example, the nested-loop join (nlj) is identified as the optimal method for joining A and B by the end of phase two. This form of pruning is equally effective as in the original optimization algorithm without configurable operators.

As we enter *phase three* for adding the next join, all considered plan fragments are containing fully configured and optimal sub-plans (marked dark blue).

Note that *propagation* is not required for this type of query optimization. The idea of constructing an optimal plan from optimal plan fragments contradicts the refinement of existing plans via propagation. The instrument of propagation is intended for optimization algorithms relying on algebraic transformations rather than enumeration and iterative construction.

For rule based optimization, we can also employ applicability and exploitability for generating optimal plans. This form of optimization, as employed by Transbase, builds an initial plan from the original SQL query. This plan is then modified using equivalence transformations during several optimization phases, where each phase is strictly self-contained and dedicated

to a specific optimization goal. To limit computational complexity, each optimization phase is conceptually one single traversal through the query plan. Therefore, the complexity of each optimization phase, and consequentially the complete optimization algorithm is linear in the number of involved operators. In this form of query optimization, the system builds its initial QEP, based exclusively on the primitive algebraic representation. This early QEP already appoints a preliminary join sequence, but the joins are still conducted using the primitive algebraic Cartesian product operator. All relations are accessed using strictly sequential traversals via primary access paths and no index selection has taken place yet. The optimizer subsequently applies algebraic transformations to this initial QEP, employing heuristics for selecting substitution rules from a set of available equivalence transformations. These transformations are organized in several top-down traversals through the current query plan. During this process, the optimizer also selects suitable algorithmic implementations for algebraic sub-expressions. Finally, the algorithmic entities' functionality for applicability, exploitability, and propagation are employed for achieving optimal algorithmic interoperability. Currently Transbase uses no cost estimation for justifying transformations during query optimization. When considering the application of custom algorithmic units, the optimizer assumes that, if such an alternative algorithm is available and applicable, then it is also more efficient than the built-in variant.

Similarly, the Access Manager's instruments for configuration and costing of algorithmic units are suitable for other transformation-based approaches to query optimization, as found in various popular DBMSs. In particular, the cost-driven depth-first branch-and-bound optimization harmonizes exceptionally well with the Access Manager approach, since this form of query optimization emphasizes a small working set, early configuration and sustained pruning.

2.4.6. Cost Function

The costs of an operation are a gauge, used primarily during query optimization, for describing the estimated utilization of limited resources during query evaluation. Limited resources are an abstraction of real hardware resources, like CPU, I/O system, and memory, as well as immaterial assets like system responsiveness, throughput, and latency. Algorithmic units, when used for extending a DBMS in accordance to our conception, have to actively participate in cost estimation. Their implementation details are hidden from the DBMS, leaving no possibility for deriving reliable cost estimation from their visible algebraic representation. The accuracy of a unit's cost estimations is similarly important for the sound operation of the

overall system as the correctness of its implementation. This encapsulation and the need for accuracy inhibit centralized cost estimation, performed solely by the host system. In accordance to foregoing design decisions, implementation complexity of cost functions inside a single operation is kept simple, in order to provide maximum usefulness at minimal implementation effort. Similarly to negotiation, cost functions are provided as a supporting instrument for justifying optimization decisions and managing resource allocation. Cost functions must operate strictly locally. In particular, they must not make any assumptions or conduct any inspections of other algorithmic units in their vicinity. As a consequence of encapsulation of algorithmic units, such inspections are hardly possible and making assumptions is dangerous, since the QEP is still subject to ongoing optimization efforts when the cost function is called, and cost estimation based on assumptions on a unit's vicinity will eventually become inaccurate. The result of cost functions must depend exclusively on the data provided as direct input. The optimizer will permanently observe whether the input parameters of cost estimation were influenced by recent QEP transformations, and eventually reinitiate cost estimation whenever necessary. Finally, as cost functions are expected to be called frequently, they must be inexpensive operations of low computational complexity.

To satisfy these prerequisites for a generic cost function, the Access Manager framework models the costs of an algorithmic unit as a set of events that potentially inflict costs, rather than associating an operator with absolute cost values. The optimizer is in charge of rating the impacts of these various events and eventually it will convert them into actual costs. This additional level of abstraction allows the description of costs in a graphic way, and relieves the implementation of a cost function from the necessity to rate its costs relatively to other operators in the system. In addition, the host system can dynamically adapt its cost model when assigning costs to individual events, allowing flexible response to changing system parameters, such as volatile system load, but also to altered system configurations. These events are classified in three main groups, in ascending order of importance with respect to their relative impact: (1) CPU instructions, (2) primary memory requirements and (3) secondary I/O. CPU assets are subdivided into total sequential expenditure and a ratio specifying the extent of parallelizable code. The latter serves for estimating potential decrease of execution times, when relocating computational load to additional CPUs by applying intra-operator parallelism. Memory consumption is subdivided into minimum in-memory temporal storage (size of the working-set) and total temporal storage requirements for completing a given task. The system will automatically anticipate costs for secondary I/O, if the total available primary memory size is exceeded. Finally, we distinguish various mimics of block I/O, namely expen-

sive random I/Os, usually more favorable sequential I/O and potentially inexpensive read-ahead/ write-ahead operations using asynchronous I/O. The class of I/O events exemplifies and emphasizes the necessity for using cost events instead of cost values. The operator's cost function has no effective means for assessing I/O costs with absolute costs, since it cannot know the I/O characteristics of the addressed hardware (e.g. conventional hard drives, RAID systems, network I/O or random access secondary memory appliances based on SSD (Solid State Drive) technology). The optimizer, as an intrinsic component of the DBMS, possesses the necessary information for centralized and accurate costing of I/O events. The presented collection of cost related events does clearly not possess the expressiveness to model every cost scheme accurately, but this is not required. It just has to be sufficient for providing a suitable approximation, biasing the query optimizer towards the correct decisions.

The abstraction of using cost events instead of absolute cost values offers an additional perception of costs. During optimization, one query plan is chosen from several candidates based on its cost events and on the rating of these events at the time of query optimization. But query plans may remain in the system over a long period of time. Stored queries, for example, are usually optimized only once, but they are intended to be reused perpetually. Hence, this optimization usually happens 'ahead-of-time', well before query evaluation. However, the actual costs for executing a query may change over time, as they depend on numerous volatile system conditions, like system load and resource allocation. Consequently, the quality of cost estimation is also subject to change. In contrast to absolute query execution costs, the estimation of cost events will always remain valid and constant. Based on these constant cost events, it becomes possible to rapidly reassess the costs 'just-in-time', immediately before the query plan is actually executed. If the system finds that the current costs are significantly different from the costs that originally justified the decision for this particular query plan candidate, the system may choose to reinitiate the query optimization process or choose an alternative plan from a cache of plan candidates.

Next the question arises, how to derive cost incidents of a single operator using a cost function that operates strictly locally on the currently considered operator, while conserving the global and versatile interrelations with other operators of the plan. In traditional cost models, local costs are composed from costs for providing the input to the operator plus the local processing costs, as expressed in the following formula.

$$\text{cost}\big([g] \circ ([f_1], \ldots, [f_\text{n}])\big) = \text{cost}([g]) + \sum_{\text{i}=1}^{\text{n}} \text{cost}([f_\text{i}]) \qquad \text{(I)}$$

Naturally, the local processing costs depend on various properties of the actual input data. These properties are approximated using cardinality and selectivity estimations, based on statistical information that is maintained by the DBMS for every stored relation. During cost estimation, this statistical information is extrapolated for describing all intermediate results in the operator tree [Sel79]. For optimal cost estimation, the host system provides such data-centered statistics for each input stream and also for the output stream. This allows the cost function to incorporate all available information for cost estimation of maximum accuracy.

The simple and obvious algorithm calculating the costs of a QEP is a recursive traversal through the query plan, accumulating costs while proceeding bottom-up. This approach allows supplying the cost function of an operator with readily available precalculated costs for its input data as well as with 'just-in-time' assembled statistical information.

The recursive accumulation of cost incidents, as reflected by the cost function above, is fully sufficient for costing the relational calculus in its algebraic conception, where every operation is implicitly blocking. But the flow of control in the algorithmic perspective is significantly different, with consequences on the cost model. In particular, in situations where it is not required to fetch the complete input of an operator for generating the complete output, this simplistic cost model becomes inaccurate. Refer to Figure.13 for an example based on a restriction in conjunction with exploitable sort orders. A general cost model has to consider these situations. An alternative cost model uses *cost-per-tuple*, as proposed and discussed in [Hel93] and [Cha99]. Cost functions based on this concept account only for tuples that are actually fetched.

$$\text{cost}\big([g] \circ ([f_1], \ldots, [f_n])\big) = \text{cost}([g]) + \sum_{i=1}^{n} q_i \cdot \text{cost}([f_i]), \quad 0 \le q_i \le 1 \qquad \text{(II)}$$

Cost function (II) is almost identical to formula (I), except that only a quota q_i of the data provided by input stream *i* is actually considered in cost estimation. This approach assumes a linear distribution of costs and it was found particularly useful when dealing with the operator classes exhibiting linear cost distribution, e.g. restrictions as user-defined predicates [Cha99]. Non-linear cost distribution is generated in the presence of blocking operators. A sort operation, for example, has to process its input completely before generating the first output tuple.

From the perspective of the consecutive operator retrieving input from a blocking sort operation, the production of the first input tuple is immensely expensive. Afterwards all subsequent tuples are available almost instantaneously, incurring very low additional costs. Conversely to the non-linearity of a blocking operation, linear cost distribution is a common quality of streaming operators, such as the aforementioned restrictions. In summary, cost function (I) is valid for costing plans consisting only of blocking operators, while cost function (II) applies to streaming operators, but neither is accurate for hybrid plans.

This dilemma is resolved by subdividing total costs into two major cost accounts: the cost portion for *blocking* operations is henceforth denoted as $cost_B$ and streaming cost portion is represented as $cost_S$. Both cost accounts are maintained independently while traversing the query plan. The local costs incurred by evaluation of the topmost operator, denoted as $cost_L$, add either to the blocking or streaming account, depending on the nature of the topmost operator.

Definition.16: *Streaming* Cost Calculation. The cumulated costs of an *n*-ary algebraic expression, concluded by a *streaming* algorithmic implementation of g, calculate as:

$$\mathrm{cost_B}([g] \circ ([f_1], \dots, [f_n])) = \sum_{i=1}^{n} \mathrm{cost_B}([f_i])$$

$$\mathrm{cost_S}([g] \circ ([f_1], \dots, [f_n])) = \mathrm{cost_L}([g]) + \sum_{i=1}^{n} q_i \cdot \mathrm{cost_S}([f_i])$$

All local costs add to the $\mathrm{cost_S}$ account, but blocking costs originating from a preceding blocking operation remain in the $\mathrm{cost_B}$ account.

For accurately modeling blocking operations, we have to distinguish two components of local costs, namely $\mathrm{cost_{L_{1st}}}$ for producing the first tuple, and $\mathrm{cost_{L_{rest}}}$ for producing all remaining tuples. Therefore, the local costs of a blocking algorithmic implementation of g are defined as:

$$\mathrm{cost_L}([g]) = \mathrm{cost_{L_{1st}}}([g]) + \mathrm{cost_{L_{rest}}}([g])$$

Definition.17: *Blocking* Cost Calculation. The cumulated costs of an *n*-ary algebraic expression, concluded by a *blocking* algorithmic implementation of g, calculate as:

$$\mathrm{cost_B}([g_n] \circ ([f_1], \dots, [f_n])) = \mathrm{cost_{L_{1st}}}([g]) + \sum_{i=1}^{n} \mathrm{cost_B}(f_i) + q_i \cdot \mathrm{cost_S}(f_i)$$

$$\text{cost}_{\text{S}}([g_n] \circ ([f_1], \ldots, [f_n])) = \text{cost}_{\text{L}_{\text{rest}}}([g])$$

More generally, when tuples are fetched from an operator exhibiting both streaming and blocking costs, then blocking costs incur for fetching the first tuple, while streaming costs are assumed to be evenly distributed over all other tuples. The following example illustrates how the various cost-relevant factors of a query plan are integrated into a sound cost model.

```
SELECT t FROM T WHERE t < c ORDER BY t
```

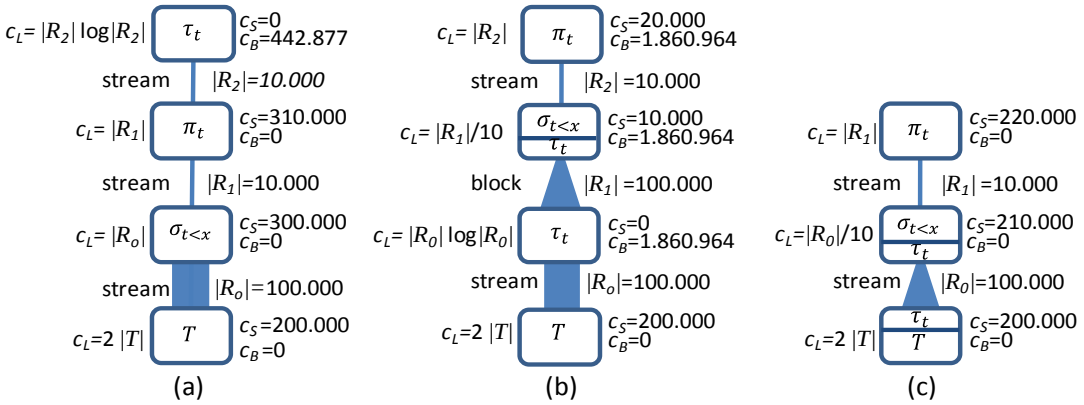


Figure.13 Costing blocking and streaming operations. The example shows the costing of three alternative plans for evaluating the query above, operating on input relation *T* storing 100.000 tuples. The cardinality $|R_i|$ of tuples exchanged between operators corresponds to the thickness for the connections. Every operator is annotated with a simple local cost function c_L, depending on the number of processed input tuples $|R_i|$, e.g. scanning one tuple from *T* is costing 2 units and scanning *T* completely costs c_L=2|*T*|. Depending on the blocking or streaming nature of a particular operator the cumulated costs for evaluating a sub-tree are charged to the corresponding accounts. For clarity we examine only one CPU related cost event.

Local costs c_L and the costs for generating input are incorporated into the operator's total costs c_S and c_B. Plan (a) in Figure.13 reduces the number of tuples early by applying the restriction first, thereby lowering the costs for projection and sort. Plan (b) applies the sort operation first. The restriction can exploit the sort order and projection benefits from the reduced cardinality. Plan (b) is attractive for predicates whose costs per tuple are high relative to sorting, or for predicates that are able to exploit the sort order for reducing the number of processed tuples. In certain circumstances plan (b) can be transformed into plan (c), which eliminates the costs of the sort operation by exploiting presortedness of *T*.

This cost model still requires some adaptations to be suitable for configurable algorithmic units. First, it has to embrace the actual configuration of the operator, which was appointed in the foregoing negotiation process. The configuration of the currently considered operator is

implicitly available to the operator's own cost function, thus there is no need to pass it as an explicit parameter. Nevertheless, it must be noted that configuration plays a decisive role in costing. Configuration relies on applicability requirements, which were demanded by an algorithmic unit and established during the negotiation process. When assessing the costs of such a configured algorithmic unit, the cost function will assume that applicability requirements are met. However, it may not consider input stream properties that exceed the requested applicability requirements, e.g. it is not legal to factor a present input sort order into cost estimation, when this input order is not explicitly enforced through applicability requirements. Such additional input stream properties are subject to change during further optimization steps and their influence on previous cost estimations is not visible to the optimizer, leading to inaccurate cost estimation, based on outdated and invalid assumptions.

Another necessary arrangement is the extraction of the complex calculation of cardinality and selectivity from the costing process, in order to keep the implementation of cost functions as simple as possible. We already observed that the boundaries of algorithmic units always coincide with complete algebraic sub-expressions. Therefore, it becomes possible to calculate selectivity and cardinalities solely on basis of the algebraic equivalent of a relational operator, which is independent from any algorithmic implementation. This task can be separated from the cost calculation process and it is entrusted to one centralized component of the host system's optimizer. This relieves the individual cost function of each operator from the difficult burden of maintaining statistical information. It also guarantees that statistics are calculated consistently in one single specialized module. Statistics on the local input streams and on the output stream are made available by passing the precalculated values as parameters to the local cost functions. These statistics are used directly for local cost estimation. Only for extensive algorithmic units, further extrapolation in the local cost function might become necessary, for accurately associating cost events with individual tasks within the complex algorithm.

Finally, our concept of a cost function complies with the aspired design goals. Such functions are provided by every algorithmic implementation. They have to assemble local cost events and accumulate them with cost events reported by their input streams. For conserving the overall integrity of the cost model, each individual cost function has to comply with the aforementioned common definition of cost calculation rules. The complex burden of cardinality and selectivity extrapolation is handled by a dedicated component of the host system, which analyses QEPs in algebraic representation on granularity of algorithmic units.

2.5. Scan Operator

Until now, we have discussed how a host DBMS can find optimal query evaluation plans when dealing with custom implementations of relational operators. In the following, we will focus on the leaf operators of retrieval QEPs. They provide actual access to relational data stored in the database, by generating the input for further processing in the query plan's internal operator nodes.

With relational scans for *data retrieval* being the leaves of operator trees, only the configuration of their upward exploitation using the $\mathcal{E}$-function is of primary importance. Applicability via the $\mathcal{A}$-function is always unconditionally possible. If alternative access paths are available, the host system's optimizer will eventually choose one access path (index selection problem), and this decision is based on cost-related considerations or heuristics, rather than on the strict functional dependencies provided by applicability.

Secondary indexes are redundant data structures for providing an alternative access path to a stored relation. Unless they are covering indexes, they contain only a projection of the data stored in their base relation. The optimizer may decide that the most efficient access to a relation is provided by such a lean index, although the index does not contain all fields requested by the query. In this case, the information retrieved from the index tuples is used to identify and retrieve the corresponding tuple from its base relation. The base tuple is covering all attributes, and it will be used to complete the index tuple. This lookup operation of a base tuple is called *materialization.* In the algorithmic perspective, materialization is an operator receiving input from an index and producing output using a scan on the base relation. It represents the only form of relational scan occurring as internal nodes of a query plan. As for all internal nodes, its negotiation capabilities comprise applicability, exploitability, and propagation.

Relational scans are also used for *data modification* (insertion, deletion, and updates). To this end, a scan on the relation to be modified is placed at the root of operator trees. The operator tree supplies the input required by the modification scan, i.e. data for insertion or the specification of tuples to be deleted or updated; in the latter case, it will also provide the replacement data. For this class of scans applicability is essential. It is used, for example, for defining a preferred order of modification by requesting a corresponding sort order on its input stream. Exploitability is of no significance for this class of relational scans, since modification scans

are always the concluding operations in modification QEPs, and they do not produce any relational output.

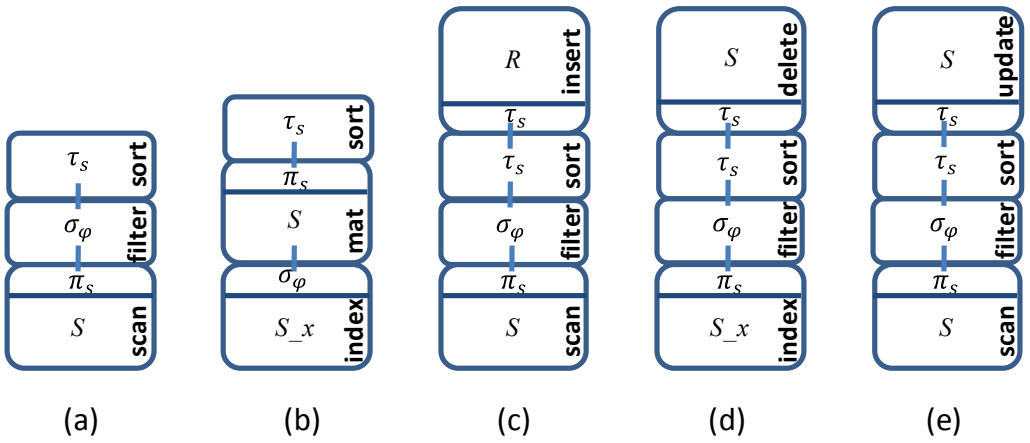


Figure.14 QEPs for retrieval and modification. *Scans* on relations and indexes are leaf operators in retrieval queries (a) and (b). They generate the input for consecutive relational operators. Scans may also be internal operators in a query plan, if materialization (*mat*) from the base relation is required after an index access (b). The result of a retrieval query is relational data, i.e. the result set produced by the root operator of the query. In modification queries (c-e), a scan on the manipulated relation serves as the root of the operator tree, for performing *insert*, *delete*, or *update* operations. It is fed with relational data required as input for modification. The result of a manipulation query is not a relation, but an integer number representing the number of tuples affected by the query.

The granularity of the minimal algorithmic unit for accessing a relation corresponds to the basic functionality required for a sequential relational scan. For improved interoperability of this class of algorithmic units, we add functionality arising from the combinatory possibilities of applicability and exploitability in form of configurable parameters $\{\pi, \sigma, \delta, \tau, \chi\}$. Although more complex scan operators are generally possible, we will demonstrate that narrowing the scope of a relational scan to this fundamental functionality provides extensive expressiveness, rich functionality, and sophisticated interoperability to the resulting compound operator. We will refer to this class of compound operators accessing permanently stored relations as *scan operators*.

In the following, we will briefly discuss the impacts of the different configurable parameters on the functional scope of access methods, and we will also sketch possible evaluations techniques for retrieval queries. An in-depth discussion of various access method implementations, including data modification, integrity, and concurrency will be provided in *Chapter 4: Architecture*, followed by an extensive use-case analysis in *Chapter 5: Proof of Concept*.

2.5.1. Sequential Access

Relations offer full abstraction from internal data representation. Therefore, the functional requirements for providing access to a relation are minimal: in the Iterator Model, a relation's only task is to traverse its data set and iteratively present one unaltered tuple after the other to its parent operator. For doing this, there must exist a linearization, allowing the operator to traverse the stored relation in a way that visits every tuple exactly once. There are no further demands to this *primary linearization*; no particular sort order is required, no selections, projections or other transformations occur. Every visited tuple is output unconditionally and unmodified. As a consequence there is no impulse whatsoever to exploit any characteristic features a particular representation of a stored relation might have. This is the primitive relational scan operator $scan(R)$ on an arbitrary relation *R*.

The host DBMS possesses all information required for resolving available access paths to a relation referenced in a database query, namely the attribute names and attribute types covered by any primary or secondary access path. This information is available from the system's data dictionary. If multiple alternative access paths are possible, then cost functions or heuristics are applied for finding the most promising query plan.

The estimated costs of pure sequential relational access, which are calculated by the scan operator's cost function, are primarily I/O related. One major cost criterion is the amount of data to be read for traversing a relation (compactness of representation). For example, an access path that stores compressed data may outperform another one using uncompressed representation in mere data retrieval, while incurring higher CPU costs for decompression.

When processing relations by following their linearization, it is important to know whether there also exists a *physical* analogon to the *logical* primary linearization. This will make the difference between inexpensive sequential I/O and random I/O. In general, access structures do not actively enforce or preserve physical inter-page clustering when updated, because of the enormous costs this would inflict. But in practice, most databases are generated via initial mass loading processes. Such mass-loading often utilizes a preferred insertion order of the access structure, which generally corresponds to the access structure's primary linearization. Therefore, the bigger part of relations is physically clustered. This clustering is preserved in read-mostly database applications, where data is not undergoing massive modifications. Otherwise clustering can be restored using reorganization facilities provided by the access structures. Hence, the cost of a scan operator performing sequential access correlates with the

number of blocks to be read and their classification into sequential or random I/O. Our cost model expresses these different cost factors as alternative I/O event classes. The actual costs for events of each class are assessed by the host system, weighted under consideration of the available hardware resources, actual workload profile, and system configuration settings.

In addition, an access method may actively contribute to the reduction of I/O costs by issuing block I/O request in a read-ahead/ write-ahead fashion against the DBMS's I/O subsystem. A first positive effect of this strategy arises from the possible combination of several individual requests into a batch of I/Os. Pooling a sufficiently large set of random I/Os allows reordering for near-sequential I/O behavior on disk-based secondary storage. Secondly, early scheduling of I/O operations, ideally well before data must be available for processing, permits overlapping I/O operations with concurrent computational tasks, like decompression or query processing. Finally, the scan operator may consider the current cache situation by inspecting or estimating whether pages that are required in the near future are already present in the DBMS cache. While processing the relation, the scan will make arrangements for avoiding the replacement of these pages in the DBMS cache. This can be achieved by processing cached pages immediately, i.e. at the beginning of the scan operation, meaning that data is processed in an order that is different to the storage structure's inherent storage order. If such ahead-of-time processing is not possible, the scan operator may choose to prevent that required pages in the system cache are replaced by other operations, before they are processed. Instead of I/O costs, this operation will incur costs for temporal storage, because these cache frames are temporarily not available for replacement by other operations. The task of the operator's cost function is to accurately assess the costs events of processing an upcoming sequential scan operation, by considering its capabilities of employing the techniques described above.

2.5.2. Sorted Access

The combination of a basic sequential relational access with a consecutive sort operation creates the first compound scan operator, which is capable of delivering an input relation R according to a given lexicographical sort criterion τ.

As already stated, data must be linearized for storage on the linear address space of primary or secondary memory, i.e. all tuples are stored in accordance to some arbitrary order. Access paths exhibiting some significant linearization are often characterized as *clustering* access paths. If such clustering resembles a lexicographical sort order, this sort order can be ex-

ploited when the relation is traversed during query execution. In addition to a *primary linearization*, data may also exhibit several *secondary linearizations.* These are maintained by auxiliary data structures such as chaining of records, or they are derived from the primary linearization using some functional dependency. Both primary and secondary linearizations may also allow navigation 'backwards' through the data, permitting data retrieval in an order that is *inverse* to the actual linearization.

SQL's data definition language deliberately chooses to provide no indication towards storage order, as one purpose of DDL is abstraction from the physical data representation. The DBMS system catalog, which is based on DDL, cannot provide this information either. Therefore sort orders, like all other configuration parameters, are negotiated exclusively by the access method's exploitability function $\mathcal{E}$. This approach provides maximum flexibility for the host system in requesting arbitrary sort orders.

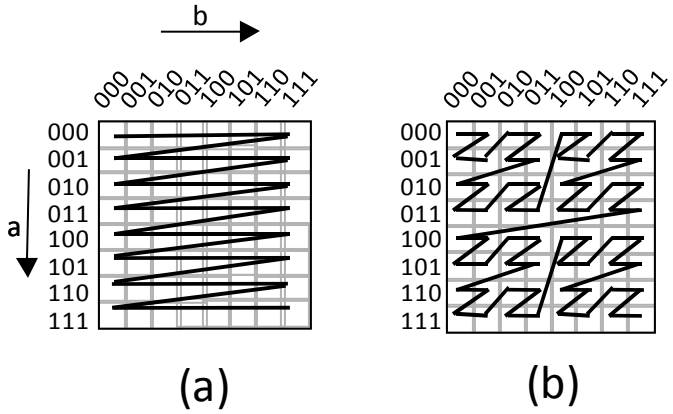


Figure.15 Linearization and exploitable sort orders. The graphs show two possible linearizations of space spanned by a relation of two attributes a and b, each having a domain of 8 distinct values. In case (a) the linearization resembles the lexicographical order $\tau_{a,b}$. This can be directly exploited in query evaluation. In case (b) the linearization is generated by bitwise interleaving the binary representations of *a* and *b*. Such linearization serves as space filling curve of the UB-tree, and it can only be exploited under considerable effort.

If a sort criterion $\tau_{\mathcal{A}}$ is to be established on the data stream originating from a scan operator, then this scan operator may choose, in the course of negotiating exploitability, to accept this sort criteria completely as $\tau_{\mathcal{E}} = \tau_{\mathcal{A}}$, whereas $\tau_{\mathcal{R}} = id$. This happens if $\tau_{\mathcal{A}}$ is compatible with a present data linearization, because the relation exhibits a linearization based on a lexicographical storage order and $\tau_{\mathcal{A}}$ is a prefix of this order. Then data needs not to be sorted conventionally, but it will be accessed in the 'right order', by exploiting the presortedness of the access structure. Thus, the access method absorbs the sort operation completely. Inside the

scan operator, the sort criterion serves as a mere directive how the data is to be accessed. This simple ordered scan operator relieves the DBMS from actually sorting a relation, offering massive savings on computational and temporal storage complexity. In addition, the ordered scan operator is a streaming operator, whereas the primitive sort operation is inherently blocking.

In less ideal cases (cf. Figure.16 for examples), a sort criterion is not fully compatible with the available linearization ($\tau_{\mathcal{A}} = \tau_{\mathcal{R}} \circ \tau_{\mathcal{E}}$, $\tau_{\mathcal{R}} \neq id$), because linearization and sort criterion are matching only on a common prefix of attributes. Then the scan operator might still exploit partial presortedness and establish the demanded sort order by using an inexpensive non-blocking partial sort $\tau_{\mathcal{R}}$.

Some clustered access methods use linearizations that are not suitable for sorted access. These linearizations favor other functionality over sorted sequential access. In the example of the UB-tree, the primary function of the employed space-filling curve (Figure.15b) is its ability to linearize multidimensional space, while preserving spatial vicinity, thus enabling efficient support for multidimensional range queries. Still, such access methods are well aware of the eminent importance of lexicographical sort orders in relational query processing, as the endeavor for sorted operations with the so-called *Tetris algorithm* ([Mar99b], [Zir99]) on the UB-tree demonstrates. Such methods integrate seamlessly into the Access Manager model (cf. *5.3 UB-Trees* on page 194 for more details). Any scan operator may freely choose whether it can provide some requested sort order. By accepting a sort order, the access method guarantees that its result tuples are sorted in accordance. There is no commitment whatsoever, as to how the sort order is actually achieved and the scan operator may employ any sort algorithm of its choice. The higher costs for providing a lexicographical sort order by means of built-in reordering has to be reflected adequately in the operator's cost function, typically in form of computational and temporal storage complexity.

Finally, if the relation cannot be accessed along the attribute with the highest weight in the sort criterion, the scan operator degrades into its original components, i.e. a sequential unordered relation scan and a blocking full sort operation ($\tau_{\mathcal{R}} = \tau_{\mathcal{A}}$ and $\tau_{\mathcal{E}} = id$). The following example demonstrates possible outcomes of negotiating sort orders of scan operators.

```
SELECT a,b FROM R ORDER BY a,b
```

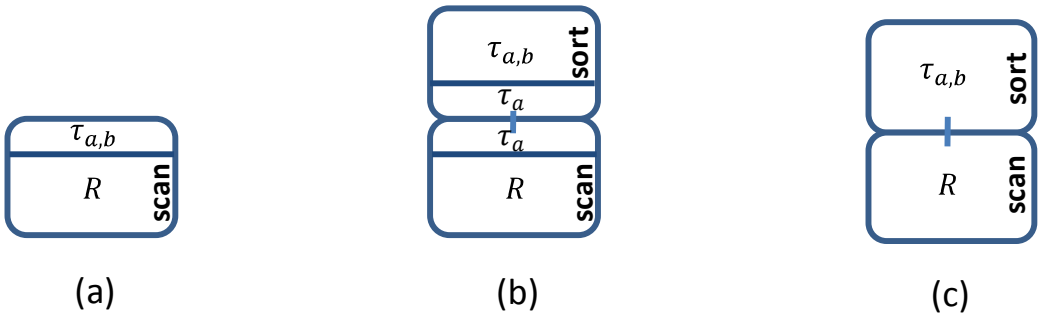


Figure.16 Sort order compatibility. Overlapping available presortedness and requested sort order. The scan operator absorbs the maximum common sort prefix. In (a) the scan operator manages to cover the complete sort criterion, the original sort operation becomes obsolete. In (b) a prefix matches, the scan operator result undergoes a final non-blocking sort operation, which relies on presortedness via its applicability directive τ_a. Similarly, the UB-tree would absorb τ_a and establish this output order by means of its integral Tetris algorithm. In (c) the sort criterion is not exploitable and the scan's result needs to be sorted with a conventional blocking sort operator.

Exploiting clustering averts costs for applying a conventional sort operation on the scan result, but ordered scans are not free of costs either. Processing a relation under a full order constraint can limit and even prevent beneficial strategies, such as prefetching and effective caching. An ordered scan operator must not start by processing pages that are readily available in cache, if these pages are not the first pages according to the sort criterion. Consequentially, these pages are at risk of being replaced, because their cache frames are assigned to other data, before the scan reaches and processes them. Therefore, the scan's order constraint will incur direct costs by necessitating repeated reading of pages, or indirect costs by blocking scheduled pages in cache until they are processed, and thereby limiting the number of replaceable cache frames.

When an ordered scan pursues a logical linearization, then the resulting I/O profile is likely to resemble random I/O, if the logical linearization does not correspond to physical clustering. Even with prefetching, this effect cannot be fully compensated. When prefetching a batch of *n* pages, the I/O subsystem will retrieve those pages in an unpredictable order, determined by the requested block's physical layout on the storage device. Yet, an unordered scan can process any page immediately as soon as its I/O operation is completed, allowing to subsequently release the associated cache frame, which becomes immediately available for other purposes, e.g. for the next I/O batch. In case of an ordered scan conducted in batches of *n* random I/Os, an average of $\frac{n}{2}$ pages are completed before the one page becomes available, that is logically the first one to be processed. In other words, half of a prefetch batch is completed and its result is retained in the system cache, occupying valuable cache frames, but these

frames can neither be processed nor replaced because of ordering constraints. In consequence, is this leads to poor overlapping of I/O operations and processing of retrieved pages, since query evaluation starts only when the first logical page becomes available, i.e. processing begins only after an average of $\frac{n}{2}$ pages have been retrieved. This leads to fluctuations in cache and CPU utilization, with adverse impact on the overall system performance characteristics. Therefore, ordered scans still inflict additional costs over unordered scans, although no actual sorting is performed.

Contrary to possible performance penalties of sorted retrieval, mass-insertion may benefit from data being delivered in an adequate order. For example, if the insertion order corresponds to the access path's primary linearization, then data insertion is conducted in one single traversal along the primary linearization, achieving a higher locality and consequently a lower I/O profile. Access structures can express their request towards a favored lexicographical sort order by means of applicability directives. But if the desired insertion sort order does not correspond to a lexicographical order, as it is the case for UB-trees, such a special order cannot be produced by conventional sort operators in a QEP. In this case, the access method has either to implement its own sort operation or relinquish this form of performance improvement. The UB-tree's *Tetris algorithm* [Mar99b] represents such an integral sort operation, which exploits a sort order on one single index attribute for fitting data into the linearization with inexpensive partial sort operations.

In summary, the ordered scan operator offers a genuinely new feature compared to its primitive components. If the data is structured along an adequate linearization, this operator provides the concept of a scan position with navigation forwards (and optionally backwards) relatively to its current position and according to the given sort criterion. This concept is of fundamental importance for efficient processing of relational queries.

2.5.3. Selection

Combining relational access with selection introduces the concept of *direct access* to the resulting scan operator, as opposed to the sequential traversal used in the relational calculus. The specification of a selection predicate on *k* attributes of the form $a_1 = c_1 \wedge \ldots \wedge a_k = c_k$ is equivalent to the specification of constant coordinates $(c_1, \ldots, c_k)$ for positioning in the *k*-dimensional space spanned by the domains of the attributes $a_1, \ldots, a_k$. These kinds of selection predicates are generally referred to as *point queries*. If direct access is supported by the data representation of an auxiliary index structure on relation R, this allows answering point

queries with sub-linear complexity, typically $\mathcal{O}(\log|R|)$ or even $\mathcal{O}(1)$, depending on the actual index structure. If combined with linearization, considerable navigational capabilities arise, allowing to position the scan freely at any coordinate and then to move forwards (or backwards) following the space-filling curve traversing the multidimensional space.

The combination of these concepts makes it possible to answer so-called range queries of the form $a_1 \geq c_{1_{low}} \wedge a_1 \leq c_{1_{high}} \wedge \ldots \wedge a_k \geq c_{k_{low}} \wedge a_k \leq c_{k_{high}} = \left[\left[c_{i_{low}}, c_{i_{high}}\right]\right]$ efficiently. Such predicates define ranges in multiple dimensions spanned by the attributes' domains, resembling multidimensional query boxes. A possible algorithm for processing such query boxes starts by positioning the scan on the lowest coordinate (relating to a chosen linearization) inside the query box spanned by the selection predicate. Then the scan follows the chosen linearization and returns tuples as long as they are inside the query box. If the space-filling curve leaves the query box at some point, the scan operator uses its random access capabilities to position the scan on the next entry point along the linearization. With this skip-scan-algorithm, which represents a generalization for arbitrary linearizations of the UB-tree's *Range Query Algorithm* [Mar99a] or the composite B-tree's *Skipper Technique* [Ram02], it is possible to calculate the result of any given query box. Whether this algorithm is also efficient, depends on the number of necessary skip operations and on the cost ratio for positioning compared to the expenditure of sequential scanning. It is also important to note, that the effective amount of data to be retrieved from secondary storage is typically higher than the exact volume of the query box, since data is usually retrieved from a block I/O device. Consequently every page contributing to a minimal coverage of the query box has to be retrieved completely, in order to satisfy the selection predicate.

An alternative to the skip-scan algorithm also starts by positioning the scan at the lowest coordinate of the query box. But then it continuously scans forward until it reaches the highest coordinate of the query box. While scanning, all tuples are validated against the selection predicate and inappropriate data is discarded. In contrast to the first algorithm, this alternative typically involves more I/O, but it also has a sustained and more predictable I/O profile and thus may outperform the first variant by employing smart prefetching.

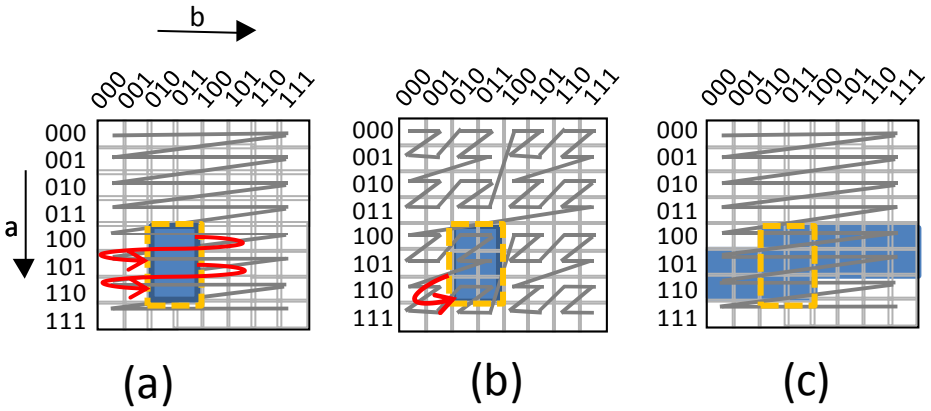


Figure.17 Query box evaluation techniques. Three examples for evaluating one query box (dashed box). In (a) and (b) the skip-scan algorithm is applied on two different space filling curves. In both cases the amount of retrieved data (dark blue) represents a minimal coverage of the queried data. In (c) the algorithm chooses to scan sequentially along the linearization, instead of skipping data. An interesting strategy, if skipping is more expensive then retrieving and filtering extra data.

Whether a scan algorithm employs any of these techniques, or other possible algorithms, is left to the operator and its implementer to decide. The essential criterion for correctness is the compliance of the scan output with the selection predicate. If alternative access paths are available, they must provide sufficiently accurate cost estimations for supporting the query optimizer in choosing the best access path for a given task.

To match the expressiveness of SQL, we now extend the concept of multidimensional query boxes. Query boxes are defined as intervals in one or more dimensions, or formally

$$\left[\left[c_{i_{low}}, c_{i_{high}}\right]\right], i \geq 1$$

SQL allows the specification of several such query boxes when accessing a relation and also permits logical combinations (AND/ OR/ NOT) of such regions. This form of conjunction and disjunction of sets of multi-attribute range queries in selection predicates is the scope of selection predicates for index structure implementations anticipated by the Access Manager framework. The framework does not limit the solution domain for this problem, but it limits the problem description domain to multi-attribute selection predicates that can be expressed in conjunctive normal form:

$$\bigwedge_i \bigvee_j (\neg) \left[\left[c_{ij_{low}}, c_{ij_{high}}\right]\right]$$

This limitation is necessary for providing a well-defined interface for negotiation and for exchanging selection predicates between host system and custom algorithmic units.

Until now, we silently assumed that an access path can enforce all kinds of restrictions evenly on all of its fields. This is not true in general. Actually, an access path is likely to provide direct access capabilities only to a subset of the attributes in its relation, and even for these attributes the restrictions are enforced with varying quality. The DBMS host system only knows attribute names and attribute types of a relation from its data dictionary. Therefore it must use *negotiation* based on exploitability for identifying those restrictions that are efficiently enforced by the scan operator. Consider the predicate $a_1 = c_1 \wedge a_3 = c_3$ and a B-tree access path on attributes a_1, a_2, a_3. Only the restriction on attribute prefix a_1 is efficiently supported by the access structure, while the restriction on a_3 is enforced by subsequent conventional filtering. In this scenario, it seems advisable for the scan only to concentrate on filtering a_1 and to ignore the other restrictions. This means that the scan operator will accept only the exploitable part $\sigma_{\mathcal{E}}$ of the original selection predicate $\sigma_{\mathcal{A}}$, while the other part $\sigma_{\mathcal{R}}$ is rejected, i.e. $\sigma_{\mathcal{A}} = \sigma_{\mathcal{R}} \circ \sigma_{\mathcal{E}}$. In this example $\sigma_{\mathcal{R}} = \sigma_{a_3=c_3}$ and $\sigma_{\mathcal{E}} = \sigma_{a_1=c_1}$. The result of the scan operation is then post-filtered in a second step against the rejected predicate, in order to retrieve the final result set. More generally, the predicate accepted by the scan operator as $\sigma_{\mathcal{E}}$ serves to efficiently reduce the retrieved tuples to a superset of data satisfying the complete predicate. In some cases, as in the example above, the predicate of the post-filtering step $\sigma_{\mathcal{R}}$ is simplified compared to the original predicate, but in general post-filtering will be forced to test the original predicate $\sigma_{\mathcal{A}}$ completely.

The decision how to split the predicate is made by general negotiation. With this, the algorithmic unit is relieved of supporting every possible kind of predicate and it may also deliberately refuse to enforce restrictions that it cannot support efficiently.

2.5.4. Projection

Integration of projection into the scan operator allows reordering attributes as required for consecutive operators. Attributes that are not needed afterwards, in particular those that are only used in the local selection predicate or sort criteria, are eliminated. Projection may lead to cost reduction by transporting leaner tuples to the next operator. This cost reduction may be insignificant for *streaming* subsequent operations, but if a *blocking* operator follows, it becomes highly attractive, because of reduced temporal storage requirements. To compensate for missing configurable projection capabilities $\pi_{\mathcal{E}}$, the query optimizer may always insert a

standard projection in form of $\pi_{\mathcal{R}}$. Even if the supplementation of projection as configurable component of a scan adds comparatively little functionality and efficiency to the compound scan operator, its importance for the coherence of a query plan through elimination of auxiliary projections should not be underestimated.

There also are cases, where an integrated projection may directly influence costs and performance. If, for example, an access path uses vertical partitioning for storing its data, then a projection reducing the number of attributes will result in direct reduction of I/O volume and computational costs, since only the requested columns have to be retrieved. A corresponding example can be found in section *5.9 Data Partitioning*.

2.5.5. Distinction

Distinction $\delta(R)$ is typically implemented to operate on a sorted stream. The actual sort order is irrelevant for the algorithm, but the sort order must cover all attributes of the tuple stream, for being useful. This allows direct comparison of the current tuple t_{n+1} with its predecessor t_n on the stream. Matching duplicates are discarded immediately. If consecutive tuples do not match, then t_{n+1} is output, as it represents a new distinct value, and henceforth it will also serve as the new reference tuple for comparison against subsequent input tuples. This algorithm is inherently non-blocking, but as it relies on an input sort order, the preceding sort makes the operation de-facto an expensive blocking operation.

Clearly a much more efficient solution exist, if an existing linearization of a relation can be exploited. The scan operator can employ a technique similar to the skip-scan algorithm for circumventing sorting and preserving streaming. Therefore, the algorithm positions onto the first tuple with respect to linearization, which is also the first representative of a group. In the following step, the algorithm uses its direct access capabilities to position onto the first representative of the next group, bypassing all duplicates of the same group. This technique can be effectively combined with restrictions and projections, as the following example illustrates.

```
SELECT DISTINCT a FROM R where b>c0
```

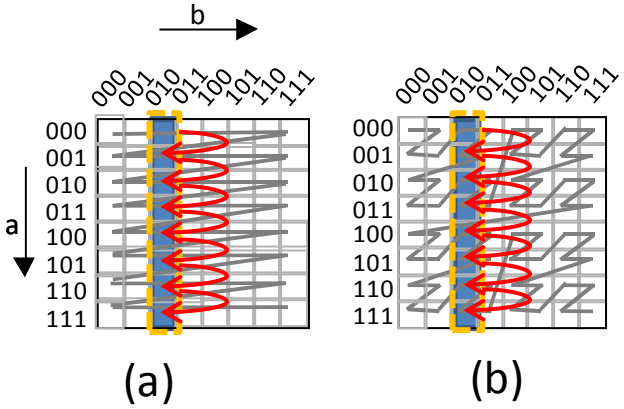


Figure.18 Distinction and linearizations. The exploitability of a scan operator makes it possible to exercise projection, selection and distinction directly on the physical data representation. This technique is interesting, if the cardinality of attribute *a* is low, since this limits the number of necessary scan repositioning.

2.5.6. Representation

Representation is the final configuration parameter. It enables a relational scan to participate in a succession of operators exchanging data in non-standard representation. This form of data exchange offers tight integration of consecutive algorithmic units. It typically comes into consideration, if the negotiation process yields *full exploitability* and all other functional configuration parameters have been absorbed without exception, i.e. $\forall op \in \{\pi, \sigma, \delta, \tau\}: op_{\mathcal{A}} = op_{\mathcal{E}} \rightarrow op_{\mathcal{R}} = id$. If this precondition is not met, then the implementation of a non-standard connector is required for providing necessary adaptations in non-standard representation. Finally, if such a non-standard connector is unavailable, then data exchange has to be conducted in standard representation.

The benefits of using data in non-standard representation as input or output apply also for relational scans in a most notable manner. The data exchange format is relieved from the burden of using a predefined, inflexible representation. Operators can freely establish whatever form of communication is most convenient for the task at hand. As an example, imagine a scan on a bitmap index. The most prominent advantage of bitmap indexes is the compactness of the bitmap representation, amplified with additional compression. This representation proves itself not only advantageous for storage, but it is also highly suitable for efficient intersection and union operations on bitmap structures. Likewise to a bitmap index implementation, any custom relational scan operator may be accompanied by a family of relational algorithms operating on the same specialized data representation. The bitmap example will be pursued in greater detail in the chapter *5: Proof of Concept*. A second application will also be

presented in the course of discussing advanced query processing techniques for B-tree and UB-tree access methods.

2.6. Chapter Summary

In this chapter, we derived the theoretical foundations for a framework allowing general extensibility of the stash of algorithmic options of a host RDBMS. The main focus lies on access methods to secondary storage. The design goal was to supplement alternative relational operators as extensions to an operational DBMS. Intrinsic components of the host system remain unaffected by these extensions, in particular SQL compiler, query optimizer, and query evaluation engine. These system components merely issue calls against a uniform interface implemented by all DBMS extensions. In addition, the host system exports a set of functions for providing access to diverse DBMS functionality. This architecture provides substantial flexibility when implementing relational operators, while averting all requirements for modifications of the host system.

Starting from the general Relational Algebra, we developed a model that allows the provision of arbitrary alternative algorithmic implementations for ERA expression within the expressiveness of the host DBMS. We determined a sensible granularity of equivalent ERA terms for such algorithmic extensions and presented configurable parameters to ensure interoperability with other components in a query plan. We demonstrated how the host system will employ configuration of algorithmic units during query planning and presented a generic cost model to be used in cost-based query optimization.

Finally, we examined the impacts the model of configurable algorithmic units has on relational scans, which are our primary concern. We demonstrated that a minimal relational scan operator, in combination with functionality that arises from its configuration, opens a wide range of algorithmic opportunities to exploit peculiarities of the physical representation of an access method. Therefore, our framework exhibits sufficient flexibility for profound implementations of auxiliary access methods. The instrument of exchanging data in non-standard representation between independent operators allow the implementation of families of tightly coupled relations, indexes and relational operators, tailored for a particular application domain.

3. Related Work

A rich body of scientific publications exists in the field of database system extensibility, contemplating DBMS customizations from comparatively small scales, like user-defined data-types and functions, over enhancement of complete DBMS components, like optimizers and buffer managers, up to building specialized systems from scratch, using predefined building blocks in 'generator' or 'toolkit' approaches. Common to all approaches is the idea of reusing an existing DBMS code basis and adapt it to a specific application domain.

3.1. Overview

A structured survey of existing work on DBMS extensibility is provided in [Dit01]. Besides presenting an extensive motivation for extensible database systems, the editors indentify important goals and also classify existing achievements in DBMS extensibility. They coin the general notion of CDBMS (component database management system) as a skeleton architecture for DBMSs that allows database users or third-party suppliers to extend a well-defined core system by adding new application specific functionality in form of *components*, allowing customization of the system by highly innovative experts in a given application domain, which naturally is not always the DBMS manufacturer. On this high level of abstraction, a component represents a coherent set of functionality, bound into an explicit software artifact with a formally defined interface. Apart from their interfaces, components function as black boxes, i.e. their implementation details are unknown. Each component adds either new features to the base system, or it serves as replacement for an existing module. In any case, individual components should exhibit maximal independence from each other, for promoting their immanent potential for reuse. A priori, the scope of functionality provided by such components is not limited. They may comprise general concepts like handling custom data types, integration of non-standard data models, DBMS adaptations in form of functional extension and replacement of existing DBMS modules, but also 'downsizing' of DBMS functionality that exceeds application requirements, and finally management and integration of external data sources. All these aspects contribute to the overall goal of providing a DBMS with the flexibility to adapt to a specific applications domain, instead of adapting the application to the requirements and capabilities of the database system. A common property of all CDBMS approaches is that components are extending functionality, while some basic DBMS framework provides the 'glue' to integrate these components into a sound system. Hence,

such frameworks define and restrict the ways in which the DBMS can be customized, but they also define the notion and the functional scope of a component. [Dit01] presents several CDBMS architectures and generalizes them into a formal, structured abstraction of DBMS extensibility. The remainder of that survey consists of a selection of papers discussing concrete approaches and aspects of existing extensible DBMS, some of which have been made available in commercial DBMS products. Although the book was released in 2001, and some of its contributions are dating back into the late 80ies and early 90ies, it still reflects the current state-of-art in DBMS customizability available today. As a consequence, the central perception from the book's foreword still holds, claiming that the presented technologies "contribute only very modestly to the lofty goals" of the ambitious agenda on DBMS extensibility. We concentrate on the main contributions of the editors, who assemble a quite complete summary of important properties of abstract extensible DBMSs and also devise an elaborate classification of general approaches to DBMS extensibility. We will employ this classification for identifying approaches that are related to our own concept and we will use their criteria for effective evaluation and differentiation of alternative approaches. The aforementioned classification distinguishes four general categories of CDBMSs, namely plug-in based CDBMS, middleware approaches, service-oriented DBMS, and configurable DBMS.

Plug-in components are unspecific software artifacts that are added to an existing and otherwise complete DBMS with standard functionality, in order to augment it for a specific purpose. The DBMS provides necessary facilities for hosting such extensions, in form of component interface specifications, instrumentation for designing, adding, and testing components and finally the means to employ such components in suitable operational scenarios. Plug-in components cover abstract data types (ADT), user-defined functions and stored procedures (UDF). ADTs are structured types, composed from primitive database types, allowing supplementary concepts like sub-typing and inheritance [Fuh99]. UDFs implement non-standard functionality and thereby extend declarative SQL with procedural concepts like scoped variables, loops, branches, and sub-procedure calls. They also allow implementation of specialized operations and predicates on non-standard data types. When associated with ADFs, they eventually imitate the general paradigm of object-orientation, including encapsulation and inheritance. Such extensions bridge the gap between an application's object-oriented data types, originating from the application domain, and the limited type system of classical RDBMS. They are subsumed under the general concept of object-relational DBMS (ORDBMS). However, these concepts are already well understood, and have been standardized in the third major SQL standard revision [ANSI99] and are ever since prevalent in

database technology. More sophisticated representations of plug-in components allow customization of complex relational operators, such as user-defined aggregations. The most advanced stage of integrating application logic into the DBMS is constituted in customized access methods, allowing efficient storage, maintenance, and retrieval of data, while preserving data structure and semantics from the application domain to a large extent. Such advanced plug-ins often concentrate on function-based indexing, i.e. *secondary* indexes based on some user-defined mapping, supporting data retrieval by applying natural predicates on domain-specific data that are not covered by the SQL standard, e.g. *contains()* for text documents or *overlaps()* for spatial data. After this mapping is applied, data is typically stored in one of the DBMS's built-in index structures, like B-trees, bitmaps, or hash indexes. The possibility to extend a DBMS with alternative base table structures and entirely new secondary access structures is usually not supported by available extension interfaces. A common observation in plug-in component systems is a domino-effect, where extensibility of one module stringently requires modifications to other DBMS modules, e.g. extensible indexing may necessitate adaptations to the query optimizer, storage layer, buffer manager, and concurrency control. Hence, the designers of extension interfaces must decide where this dependency chains are broken, in order to provide a sound and intelligible interface with a reasonable amount of flexibility. Secondary design goals are minimization of implementation complexity for new plug-ins, safe and reliable extensibility without compromising the integrity of the host system, and finally the prospect of a significant performance benefit is required for motivating plug-in development. The Access Manager approach clearly qualifies as plug-in component architecture, but it assumes that basic ORDBMS concepts such as ADTs and UDFs are already an integral part of the host system. In contrast to most other plug-in CDBMSs, its focus lies on storage and retrieval facilities, but its concept also covers extension of arbitrary relational operators.

The main focus of the *middleware* approach is the integration of external data sources, namely the combination of independent DBMSs into a Multi-DBMS, respectively integration of arbitrary heterogeneous information systems and other external data sources (files, e-mails, web), but also transient data such as information on the state of an operational system or a sustained stream of sensor data. In this approach, extension of a DBMS is accomplished by middleware components functioning as wrappers of individual data sources, leveling heterogeneous data models and concealing the physical location of distributed data sources, and eventually establishing a homogeneous, location transparent, logical view on the complete collection of data. With this, heterogeneous data sources are unified by transforming and

exchanging data in an intermediate data exchange format, facilitating data integration into global query processing and transaction management. The unification of data sources also represents the central challenge of middleware CDBMS architectures, as the individual capabilities of data sources may vary on a broad spectrum, concerning supported query languages, data models, transaction capability, concurrency control, etc. As a matter of fact, middleware approaches are a logical consequence of plug-in components. The transition from user-defined *value* functions to user-defined *table* functions is relatively simple. Table functions produce a result set (table) by iteratively returning one row at a time. If the returned data originates from some remote data source, then table functions obviously qualify as a middleware component. Nevertheless, table functions are comparatively primitive representatives of the middleware components, as they provide read-only access and support neither global query optimization nor distributed transactions. Yet their classification blurs the differentiation between plug-ins and middleware components. Consequently, most plug-in component systems also incorporate some aspect of the middleware approach. This is also true for the Access Manager architecture, whose condensed data access interface possesses all relevant properties for effective integration of heterogeneous, remote data sources. As a consequence, the Access Manager framework handles internal and external data sources equally on a location transparent level.

The *service-oriented* class of CDBMS divides the functionality of a monolithic DBMS into a collection of stand-alone database services. Here the term component addresses such a database service implementing an ‘unbundled’ subset of DBMS functionality, e.g. persistence services, transaction services, concurrency, query processing, etc. As each service is fully independent from other services, an application may dynamically compose the DBMS functionality it requires for its own operation, by requesting the corresponding services through service broker mechanisms. The interfaces of services for a particular purpose are standardized, allowing exchangeability of compatible service implementations. The goal of this approach is not extensibility or customizability of an existing monolithic DBMS, but rather the dynamic composition of a DBMS for a specific purpose. CORBAservices [Obj95] are an example for a standardization of such services by the Object Management Group (OMG). But besides establishing the basic principles, this approach is of little practical relevance. And as extensibility is not the primary objective of service-oriented CDBMSs, they share only very little similarity with our own approach.

The fourth class of CDBMS consists of *configurable* DBMS. In contrast to their service-oriented counterparts, where services are fixed and standardized parts that may be combined to form a complete DBMS, the components of configurable DBMSs correspond to subsystems, each implementing a subset of DBMS functionality, but neither interfaces nor the partitioning of functionality is initially fixed. In this conception, an operational DBMS is entirely composed of such components and there is no framework enforcing interoperability. There exists however an architecture model, defining the functionality of an individual component class. But this model is not fixed, allowing adaptation of component classes to new requirements and the definition of new component classes. The process of obtaining a functioning DBMS is a configuration process, where the DBMS implementer selects from a set of reusable components, or builds entirely new components, where each component implements some aspect of the desired functionality. Configuration allows mixing and matching components in such way that they integrate into a sound system. Again, this concept deviates strongly from the Access Manager approach, which is based on a strict framework, with an invariable host system at its core and a well-defined extension interface with compulsory interoperability protocols.

In the following, we will analyze several concrete examples of relevant CDBMSs, for providing a more detailed overview of existing technology. Wherever possible we will seize the opportunity to illuminate similarities and deviations with our own approach.

3.2. Production Systems

In the first part of our survey, we will concentrate on available CDBMS technology in industrial strength implementations, by reviewing several commercial and one open-source system.

3.2.1. Informix

The most powerful but highly complex technology for DBMS extensibility is available in the Informix Dynamic Server (IDS), which was acquired by IBM in 2001. Since then, IDS development is pursued in a branch parallel to IBM's primary DBMS product DB2, with particular focus on OLTP and embedded system environments. The IDS supports integration of *DataBlade* [Ube94] packages, allowing for extension and modifications on several layers of the host system. IDS's inherent ORDBMS concepts like ADT, UDFs and user-defined aggregates add to the system's ability to adapt to specific application domains. The most significant parts of the DataBlade technology are the Virtual Table Interface (VTI, [IBM03a])

and the Virtual Index Interface (VII, [IBM03b]), both based on the Iterator model, designed with focus on embedding *external* data as 'virtual' tables into the DBMS. The availability of these interfaces resulted in numerous DataBlade implementations for external data access, e.g. C-ISAM, text, image and video, spatial, geodetic, web, GiST (generalized search trees, see also 3.3.1) and GIN (generalized inverted indexes) integration. It is also possible to use VTI and VII for *internal* storage within the system's so-called 'smart blobspace', which is a dedicated IDS storage area for binary large objects (BLOB) of arbitrary contents. But even for internal storage, many essential DBMS concepts such as transactional contexts, buffer management, concurrency, locking, logging, and recovery are not commonly supported. They are left to the Blade-developer as an almost unbearable burden. VTI and VII offer many necessary concepts for effective development of genuine alternative access methods, but owing to the primary operational area as gateway to external data, they suffer from missing integration into the storage, concurrency, and recovery facilities of the host system.

3.2.2. Oracle

Since version 7, Oracle incorporates extensibility support in form of stored procedures within their database product, and with version 8 the first object-relational Oracle DBMS became available in 1999 ([Ora02], [Ora03]). This system possesses all typical ORDBMS features, i.e. an extensible type system and a server execution environment for UDFs. Beyond basic object-relational functionality, Oracle supports user-defined operators for selection and aggregation, function-based-indexing, access to external data sources and an extensible optimizer. The functionality for a specific application domain is provided in form of dedicated modules named *Data Cartridges*. A Data Cartridge integrates into the host system via the ODCI (Oracle Data Cartridge Interface), which is constructed of several components, each dedicated to a specific purpose. The interface for extensible secondary indexing is based on the Iterator model and data is stored internally in IOTs (index-organized tables), e.g. by operating a high-level procedural SQL (PL/SQL) interface. External storage is in principle possible, but it requires a considerable amount of code effort for maintaining consistency, backup, recovery, allocation, etc. As a lightweight alternative, the definition of a functional index on the mapping of column values using a user-defined function, offers lookup and materialization capabilities of pre-computed values. The concept of *Abstract Tables* allows access to external data outside the host DBMS. It provides a permanent reference to remote data in the system catalog, much like a view definition providing location and credentials for accessing a remote data source. The actual access is conducted over the Iterator-based ODCI.

A noteworthy characteristic of ODCI, in comparison to most other approaches, is its provision for extending the host system's optimizer with selectivity and cost estimations for user-defined operators. In summary, ODCI provides support for user-defined secondary indexing based on built-in underlying access structures, to be used for functional indexing. Access to external abstract tables is limited to full table scans and lookup of row ids (RIDs), resulting in possibly severe performance limitations. Implementation of alternative primary and secondary access structures is clearly beyond the scope of ODCI.

3.2.3. IBM DB2

In 1995, IBM presented its first object-relational extensions to the DB2 RDBMS, which were augmented and completed in subsequent releases of the DB2 Universal Database System (UDB) product. In addition to standard ORDBMS features (ADTs, UDFs, etc.), DB2 also comprises a framework for extensible user-defined indexing and access to external data sources. Such specialized functionality is available in prepackaged collections called *Extenders*, each dedicated to a certain application domain. Among the presented extensible DBMS, DB2 provides the most elegant indexing framework with respect to implementation complexity [IBM02a], [IBM02b]. To build a new index type, a programmer has to provide at most four user-defined functions that are used as hooks in the actual indexing framework operating a classic B-tree structure [Sto03]. Although tempting in its convenient simplicity, this approach suffers from its restriction to one single B-tree for indexing. Therefore, this interface is suitable for functional indexing, but it is not adequate for developing alternative indexing methods. The convincing usability of this approach motivated the discourse on its usability as a template for generic functional indexing (cf. section 5.7) in the *Proof of Concept* chapter of this thesis.

In addition, DB2 can be coupled with autonomous external information systems (IS), such as spatial databases, text retrieval systems etc. The system allows maintaining foreign key references through special user-defines types (e.g. 'handles' defined by the external IS) to external data and provides mechanisms to process predicates or retrieve external data by exploiting these references.

3.2.4. Microsoft SQL Server

The SQL Server possesses all typical ORDBMS functionalities for providing standard extensibility. Apart from this, Microsoft pursues a radically different approach to DBMS extensibility compared to its competitors, which is based on the OLE DB framework (Object Linking

and Embedding for Databases). The SQL Server gains advanced extensibility with its ability to participate actively in an OLE DB network interconnecting heterogeneous, distributed data sources.

The OLE DB framework is capable of assembling complex networks of heterogeneous OLE DB data providers and OLE DB data consumers, possessing highly flexible interfaces for data retrieval and data manipulation. Interface factoring and dynamic introspection allows each component to implement that subset of the complete OLE DB specification it finds convenient for modeling its actual capabilities. For example, primitive data providers allow forward-only, read-only access to their data, while complex data provider possess elaborate query language support for defining fully scrollable and updatable result sets. In addition to conventional relational navigation through data sets, they may allow hierarchical navigation (e.g. for navigating along referential constraints between relations) and navigation through heterogeneous collections of data. Such navigation is typically supported by adequate access paths that are inherently available as part of the data provider implementation. Data is exchanged in a common data representation, equipped by conversion and binding mechanisms, and enriched with metadata information. OLE DB components embrace the object-oriented paradigm by enabling the general concepts of encapsulation, sub-typing, and inheritance. In addition to composition of networks of data providers and consumers, OLE DB devises services as a third type of middle-tier components, functioning as consumer and provider at the same time. Services can bridge deficiencies between the capabilities of data providers and requirements of data consumers, by enabling additional abilities like data caching or relational querying processing on primitive data providers. Finally, this general approach allows the SQL Server (and any other DBMS) to function as data provider, but also as data consumer. The SQL Server's internal components also expose OLE DB interfaces, e.g. the relational query engine and the storage layer. Hence, the relational engine, acting as a data consumer, may connect to arbitrary data providers, whose individual capabilities may be leveled using OLE DB services like cursor services, data transformation services (DTS for ETL), OLAP services etc. OLE DB driver implementations are available for many applications and file formats is the Microsoft product family. Additionally, generic data providers based on common DBMS APIs like ODBC or JDBC, allow interconnectivity with all DBMSs conforming to these standards.

Conceptually, the OLE DB framework is very powerful, but it suffers from its high complexity, described in what the authors themselves call an "excruciatingly detailed specification"

(cf. [Dit01], page 172). The recognition of this shortcoming resulted in the specification of the ADO interface on top of OLE DB, as a tailored abstraction layer for application developers, hiding much of the OLE DB complexity. Also, in spite of its high complexity, OLE DB is not complete, as the integration of the Microsoft Search service (file system full text search) into an OLE DB component necessitated specialized adaptations beyond the scope of OLE DB, for resolving optimization, indexing and security issues (cf. [Dit01], page 164). The complexity of the interface specification results in high flexibility when composing OLE DB networks, but it also promotes an important weakness. If one component in this extensive network fails to comply with the complex specification, data integrity is jeopardized, resulting in a high potential for instability of the overall system. Also, maintenance and replacement of individual components in a network may provoke adverse behavior through unpredictable side-effects.

OLE DB and the Access Manager framework possess several analogies. Any access method can be seen as a data provider. Access methods may be stacked, i.e. a layer consumes data from the underlying layer and provides data to the layer above, just like an OLE DB service. In addition, each layer may choose to inherit, reuse, or overwrite functionality of an underlying layer. Finally, the host DBMS functions as a data consumer, retrieving data from all connected sources and submits it to query processing. But in contrast to the extensive OLE DB specification, the lean Access Manager interface is significantly more intelligible. The main difference however, is the Access Manager interface's emphasized support for global query optimization.

3.2.5. MySQL

Like any other representative of the class of open source database systems, MySQL [Ora10] (owned by Oracle Corporation) is predestined for customizations of the system, since the availability of the complete source code allows arbitrary intrusion into the system core. But to the best of our knowledge, extensions of the indexing framework are not actively encouraged by the system, as MySQL does not offer a dedicated, explicit, and documented interface for custom access methods. But in terms of extensibility, MySQL offers a different, genuinely unique approach. Instead of integrating alternative access methods, MySQL supports the replacement of the complete storage layer. To date, at least four distinct MySQL storage systems are available, namely MyISAM, InnoDB, MEMORY, and NDB (Network Data Base), each with its own set of capabilities with respect to transactional isolation, lock protocols, recovery, partitioning, and in particular with different index methods. The combination

of all storage layers supports B-tree, Fulltext, Hash, and R-tree indexing, but no single layer implements all four access methods. Hence, for implementing custom indexing methods, a developer has to attain a deep understanding of the internal workings of at least one storage layer, but possibly also of adjoining subsystems. This obviously results in a tremendous implementation and maintenance effort.

3.3. Research Prototypes

In the second part of our survey, we will present promising alternative approaches to DBMS extensibility that are not openly available in commodity database technology. Still it is likely that some of these findings have been incorporated into the design of internal interfaces of commercial DBMSs, serving for structured and systematic proprietary extensibility when new functionality is integrated into the system core by the DBMS vendor.

3.3.1. GiST

The framework for Generalized Search Trees (GiST) [Hel95]**,** [GiST90] defines the minimal common interface required for implementing generalized tree-based indexing structures and a number of such indexing structures have been made available as GiST modules, including B-tree and R-tree implementations. GiST has been used mainly in research prototypes using *libgist*, a stand-alone, file-based GiST implementation. However, all aspects of a surrounding database system are missing. Although there have been noteworthy efforts for integrating GiST into a major DBMS, e.g. into Informix Dynamic Server and PostgreSQL [Kor99], and into Oracle [Kle03], [Döl02], these solutions are not widely accepted. Still, the universality of this approach [Kor99], [Kor00] together with available advanced concepts like concurrency considerations [Kor97] or nearest neighbor search [Aok98], makes the available GiST prototypes interesting candidates for a possible integration into the Access Manager framework.

3.3.2. Starburst

IBM's Starburst [Haa89] project (1984-1992) resulted in the prototype of an operational extensible RDBMS. Starburst possesses an extensible query language (Hygrogen). In addition, the system is functionally extensible via plug-ins, e.g. relational operators and access methods called Low-Level Plan Operators (LOLEPOPs), representing algorithmic entities used by the system's query processor. An extensible query optimization mechanism, using rule-based query graph transformations on the algebraic representation of a query plan [Pir92], accepts definitions of supplemental grammar-like production rules, called Strategy

Alternative Rules (STARs) [Loh88]. After a succession of non-terminal transformations using STARs, the grammar finally maps algebraic query plan operators to algorithmic LOLEPOP terminals. Both STARs and LOLEPOPs may demand certain properties in order to be applied in a query plan, e.g. some specific input sort order. Starburst's *'glue' mechanism* establishes necessary requirements by installing auxiliary operators for permitting STAR transformations. A query plan composed completely of LOLEPOPs is submitted to cost estimation, before it eventually qualifies for evaluation. The impact of the Starburst approach is still relevant today, since it forms the basis of the IBM DB2 query optimizer and query processor. But unfortunately, in DB2 the flexibility of Starburst is not accessible to database architects, administrators or users.

On a high level of abstraction, Starburst shows much resemblance with our own approach. LOLEPOPs correspond to algorithmic entities in our conception, as parameterized, executable operators for query evaluation. STARs describe the replacement of an algebraic term with its algorithmic equivalent and the glue mechanism establishes the input requirements of an algorithmic entity. But in contrast to Starburst, the Access Manager approach does not require explicit STARs for globally transforming query plans with the intention of applying custom relational algorithms. It uses the algebraic equivalence pattern only for ensuring correct substitution. Global query plan transformation and substitution are conducted solely the by system's intrinsic query optimization component. Effective integration of an algorithmic unit into a query plan is conducted via the cost-driven negotiation process on a strictly local scale.

3.3.3. Garlic

The goal of IBM's Garlic [Car95] project was the design and development of an operational prototype of a wrapper architecture for integration of heterogeneous legacy data repositories (e.g. RDBMSs, web search, image servers, etc.) into one uniform information system with the ability of distributed querying across multiple repositories. Its technology is still in use today in IBM products for content integration. Garlic itself stores no data, except for metadata describing the attached repositories. Data residing in repositories is organized in *collections* of objects, based on an object-oriented data model. The elements of each collection are described using the custom object definition language GDL (Garlic Data Language), which is based on ODL (Object Definition Language) of the Object Database Management Group (ODMG, [Cat00]). In this description, each object is assigned to a class of objects, each having descriptive attributes and exporting an optional interface. In addition to this abstract data model, Garlic possesses a standard data representation for primitive data, which is used for parame-

ters and results of object method invocations, in particular in the object attribute get/ set methods. Garlic's design is particularly interesting, as it emphasizes query optimization aspects, while other approaches concentrate on mere data integration. Instead of demanding a declarative specification of the capabilities of each data source, or by forcing all data sources to implement standardized functionality, Garlic wrapper implementations participate actively in the query optimization process. Therefore, the Garlic optimizer [Haa97] provides a wrapper with a generic work request, representing the largest possible plan fragment from a multi-repository query, that may be dispatched to one individual repository. The wrapper may partially decline or accept the work request, by responding with one or more query plans representing those parts of the original request that are corresponding to the repositories capabilities. The wrapper may also annotate these plans with costs, statistical information (cardinality of result), and result set properties like sort order (details in [Tor99]). Based on these estimations, the cost-based Garlic optimizer will eventually choose one of the proposed query plan fragments for integration into the global query plan. Declined portions of the original work request are compensated by performing the necessary operations inside Garlic, after retrieving the data from the repository. This solution permits rapid development of wrappers with a low initial complexity by dynamically exploiting the effective capabilities of the wrapper. Initially simple wrappers may evolve over time, as each new release may accept more complex work requests, until the specific capabilities of a repository are sufficiently represented.

Similar to Starburst, the Garlic optimizer is based on transformation rules following the STAR approach [Loh88]. Depending on the capabilities of a wrapper, the optimizer may issue work requests describing accesses to single collections, including predicates and projection directives, but also grouping and aggregation, joins for two or more collections residing in the same repository, and finally arbitrary plan fragments limited to one repository. Therefore, the Garlic approach qualifies for iterative bottom-up query optimization based on dynamic programming and pruning [Sel79]. In the resulting tree-structured query plan, plan fragments generated by wrappers always show as more or less complex leaves, each modeling access to one or more collections residing in the same repository. The inner parts of global query plans consist of Garlic operators compensating for missing query capabilities of individual repositories, but also of joins performed on data originating from different repositories. Again, query evaluation is based on the Iterator model, as a partial plan is processed by iteratively retrieving result rows from the wrapper component.

Besides the obvious similarities between Garlic and the Access Manager framework, where both approaches lend themselves to query optimization and implementation complexity depends on the capabilities of a data access component, there are several important differences. Most significant is Garlic's restriction to read-only access to repositories, while the Access Manager allows transactional and fully consistent retrieval and manipulation of data, while upholding location transparency. Garlic uses a repository centric approach to query optimization, where a plan fragment is ascribed to one repository and details of a query plan like exact specification and chronology of applied transformation on a remote repository are not visible to the Garlic optimizer. Consequently, Garlic cannot support secondary indexes residing outside of the repository housing the base data source, nor has it control over index selection within a repository. The Access Manager optimizes queries in an access path centered approach, where access paths to the same base relation may reside in different access modules and also transparently on different sites. Custom operators, especially data access related operators like manipulation and materialization, may appear throughout the global query plan, intertwined with standard operators from the host system. The Access Manager approach consciously distinguishes between primary and secondary access path candidates, using a customizable cost model for its final decision. This cost model requires only cost information from a data access module, while a Garlic wrapper might be forced to maintain and supply statistical information like cardinality and selectivity for supporting the Garlic optimizer. Finally, the Access Manager is integrated into an operational RDBMS, whose own local data repository is expected to participate in most transactions, while Garlic accesses only remote repositories, storing no data of its own.

Hence, Garlic uses a distributed approach to query optimization, where every repository optimizes its private plan fragment, whereas the Access Manager promotes global optimization. This global optimization ultimately puts the Access Manager framework in the position to manage read/ write access and maintain consistency across distributed secondary access paths to some relation. Yet, this approach has not only advantages, as Garlic's concentration of read-only access into the leaves of its query plans enables joins across collections residing in the same repository, a feature that cannot be emulated with *basic* Access Manager assets (cf. *4.6.2 Data Integration Layer* on page 179 for an approach for relocating arbitrary relational transformations to remote repositories).

3.4. Discussion

The foregoing survey demonstrates that certain concepts are recurring frequently in alternative approaches to DBMS extensibility. In particular, the Iterator model is the preferred method of query evaluation for extensions, reappearing in different flavors, from simple sequential full-table scans, over lookup of row identifiers, to evaluation of predicates and other relational expressions. Yet among all presented approaches, the customization of relational scans in the Access Manager is unrivalled in its completeness and its tight interaction with the query optimizer. A strong focus on query planning in related approaches is stunningly rare, although it is obviously the key for effective employment of DBMS extensions in performance-critical environments. Query planning must happen on a reasonably fine level of detail, including important optimization concepts like index selection, subsequent materialization, and efficient index maintenance. Moreover, most extensions to monolithic DBMSs are subsequently mounted as fairly isolated attachments to the existing systems. This approach is often justified with concerns that plug-ins might compromise system stability and integrity, which admittedly are properties of paramount importance for information systems. Yet, obstructing access to crucial DBMS services such as transaction management, persistent storage layer, buffer management, locking, and recovery accomplishes a counterproductive goal. Every plug-in has to laboriously re-implement this functionality, resulting in a tremendously increased complexity and probably poor interoperability with host system components and other plug-ins. In our conception, facilitating plug-in development by providing recurring standard functionality and encouraging reuse of existing plug-in components is the best approach to achieve a consistent and responsive overall system. In case of transaction management, such integration is actually the only way to obtain a sound system, capable of maintaining data integrity over a diversity of external data repositories.

Access Manager Framework will prove that it provides unrivalled extensibility by accommodating plug-ins via a lean, universal, and adaptive interface. Although intrinsic components of the host system are neither extensible nor replaceable, their behavior may be influenced though convenient interfaces. Access to these internal components also allows reusing these components' functionality for recurring tasks and thereby achieves a tight integration of plug-ins into the host system, facilitates and accelerates plug-in development, and averts malfunctioning by reusing mature functionality provided by the host system. In addition to plug-in extensibility, the Access Manager also qualifies as middleware approach, since it allows location transparent access to arbitrary data sources by dynamically adapting its general-

purpose interface to a particular data source's capabilities. As a consequence, distributed query processing becomes possible, and the Access Manager's global optimization ensures its effectiveness.

The Access Manager approach is neither service-oriented, nor provides it extensibility according to customizable DBMS approach. The host DBMS represents a functionally sound and highly integrated system. We believe that such a system cannot be decomposed without sacrificing system integrity, stability, and performance.

In the following, we will present in detail how the Access Manger achieves the claimed goals, by surveying its interface specification and discussing its prospects towards extensibility, anticipated performance, and provisions for supporting plug-in and application development.

4. Architecture

Our approach towards DBMS extensibility is based on the Access Manager interface, which is presented in this chapter. Its design focus lies on simplicity and usability, making extension of a DBMS with user-defined access methods fairly easy, as the initial implementation complexity for simple access methods is low. Reuse of existing components is strongly encouraged when embedding a new access method into the host system, allowing rapid development by avoiding the necessity of time-consuming and error-prone re-implementations. Basic functionality is founded on a small and comprehensible set of interface routines and a simple and intelligible protocol. As a secondary design goal, the Access Manager interface allows thorough integration and efficient operation of new access methods inside the host system, while providing a high degree of flexibility for necessary adaptation of the host system to the specific requirements of a particular access method. This goal is achieved via specification of several optional Access Manager interfaces, permitting advanced operating modes for sophisticated access method implementations. This opens the possibility to start off with a simple implementation of an access method, by implementing only mandatory interface routines. Afterwards, the access method can be selectively enhanced through implementation of optional Access Manager interface routines, until the access method supports all necessary operations that correspond to its functional scope. Finally, each access method has a significant degree of control over performance-relevant behavior and strategies of some selected subsystems of the host DBMS. By actively influencing the host system, an access method will adapt the host system's behavior to its specific needs and thereby improve the tightness of its own integration. During this iterative implementation process, the complexity of an access module will evolve as functionality is added and tighter integration is attempted.

4.1. Layered System Model

In the following, we will provide an overview over all layers of the host DBMS that have to be adapted for affiliating custom access methods. We will also discuss non-standard DBMS interfaces, necessary for providing extensibility and for allowing thorough and unrestricted integration of new access methods. The interrelations of the various subsystems of the host DBMS are demonstrated by an in-depth inspection of the life-cycle of a custom access method.

An access method possesses a *tuple-oriented* interface, which is operated by the host DBMS's query processor according to the *Iterator* model. The access method itself is usually founded on the *page-oriented* interface of the host system's buffer manager and storage layer. The storage layer provides access to the system's sophisticated I/O facilities, including prefetching and asynchronous I/O capabilities, priority queues, and scatter/ gather I/O. This I/O subsystem is complemented with a powerful data caching facility, implemented in the host system's buffer manager. Interfaces to these subsystems allow active control over I/O strategies and caching policies. In addition, the storage layer provides access to valuable functionality in form of intrinsic locking and concurrency mechanisms. It also enforces transactional consistency and isolation and possesses logging and recovery functionality. All these complex DBMS features are fundamental for the implementation of expedient access methods. They are readily available as a service on page granularity, provided to all custom access modules based on the host system's built-in storage layer.

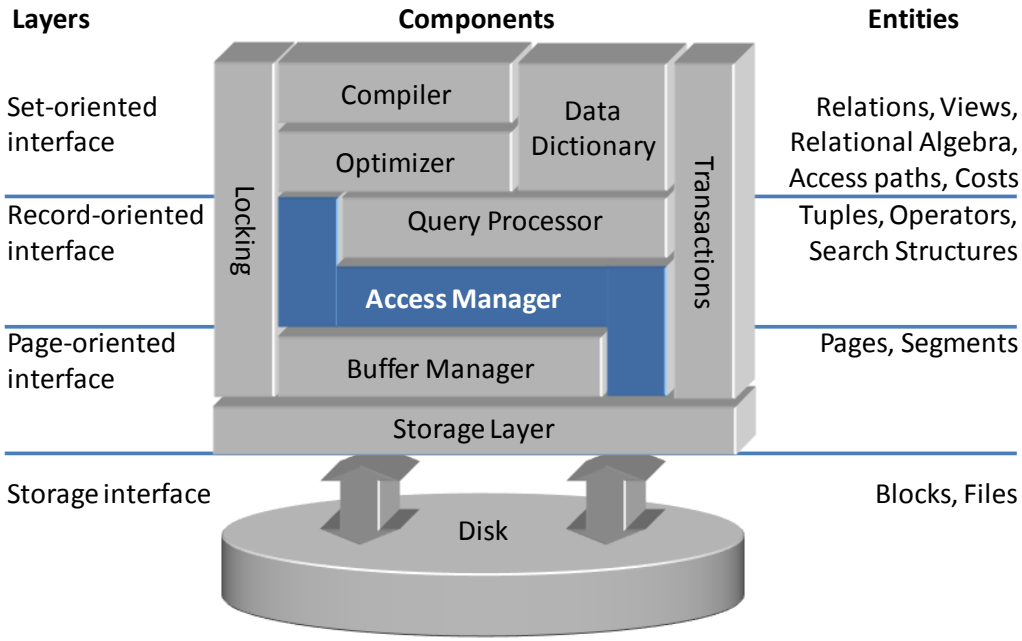


Figure.19 Classical layered DBMS architecture and the Access Manager. The Access Manager's primary purpose is to provide a tuple-oriented *Iterator* interface towards the query processor. Another interface is provided to the query optimizer for configuring and costing access methods. The Access Manager uses the buffer manager's page-oriented interface for accessing and manipulating pages and for controlling caching of individual pages. In addition, the Access Manager has to interface the host system's storage layer for directly influencing block I/O on persistent memory. Finally, optional interfaces of secondary importance are provided to the system's locking and transaction managers.

The Access Manager encapsulates all access methods available in the system, such that no direct interfaces to other system components exist. Therefore, access methods may only

interface each other and use the Access Manager as a gateway to other host system components. The Access Manager embraces the host system's built-in access methods, making them available as building blocks for new access methods. The B-tree structure is the ubiquitous example for search structures in database technology. It relies on the page-oriented interface of the storage layer for retrieving data from pages stored persistently on secondary memory. It possesses a tuple-oriented interface for providing sets of data tuples, to be used by the query processor. Access methods mapping pages to tuples represent the class of *full* access methods. Implementing full access methods from scratch is a laborious venture, therefore the Access Manager allows to reuse functionality from existing access modules when implementing new ones. As an example, consider an access method using one or more auxiliary B-tree structures for storing and retrieving data. A full-text index may be implemented using several classical B-trees, where one tree is used as the actual index structure, containing information on occurrences and positions of words in the indexed documents. A second B-tree contains a list of words that are insignificant for searches or frequently recurring words which are to be ignored by the index (stop words). A third B-tree provides mappings for word stemming. Such a full-text index implementation, based solely on the tuple-oriented interface of another full access module, is a representative of the class of *intermediate* access modules.

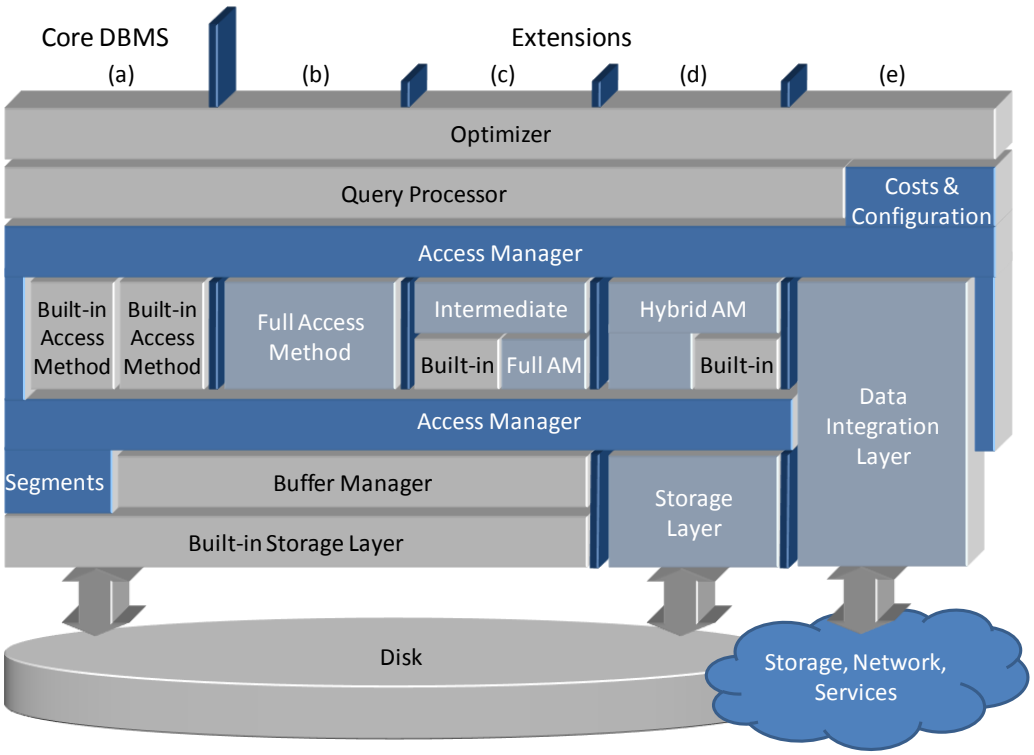


Figure.20 Types of access method implementations. The Access Manager encapsulates all access methods, while providing interfaces to the relevant parts of the host system. The diagram shows five different types of access methods, represented in vertical sections. (a) represents built-in *full* access methods implementation that are implanted into the Access Manager framework. (b) represents a supplemented full

access structure. It also provides a mapping from the page-oriented storage layer to the tuple-oriented query processor. (c) is a lean *intermediate* access method implementation, relying on the tuple-oriented interfaces of one built-in and one custom access method. (d) shows a hybrid access structure, using the page-oriented built-in storage layer for persistent storage, but also an auxiliary access method's tuple-oriented functionality. Independently from the hybrid access structure (d) also demonstrates the integration of a *custom storage layer*. (e) depicts a *data integration layer* accessing remote data outside of the host DBMS.

Another example for the class of intermediate access methods is the UB-tree, enhancing the classical B-tree technology for providing efficient access to multidimensional data. This is accomplished by a functional mapping of all indexed attributes (dimensions) to a suitable multidimensional address, using a bit-interleaving technique. The result of this process is a relation with one additional attribute with a strong functional dependency to all indexed attributes. This attribute is then used as the sole key attribute of a classical B-tree structure. The basic search and storage functionality of the B-tree module remains unaffected, but it is controlled by a lean intermediate UB-tree layer, providing the necessary functionality for advanced handling of multidimensional queries. These approaches to full-text and UB-tree indexing technology, based on an existing B-tree module, dramatically reduce implementation complexity, while the resulting intermediate layers offer significant new functionality compared to the technology used to implement it. We will pursue this approach in greater detail in the upcoming *Proofs of Concepts* chapter. Finally, it is also possible to combine the two aforementioned concepts for implementing *hybrid* access structures. These structures store data on pages provided by a page-oriented storage layer, i.e. they are full access structures. In addition, they employ auxiliary access structures based on existing access modules for efficient lookups, enforcement of data integrity constraints, or other supplemental functionality. An example, where a simple linear access path, organized as linked list of pages, is augmented with an auxiliary structure for obtaining look-ahead capabilities for exploiting the I/O system's prefetching facilities, is also provided in the *Proofs of Concept* chapter.

The Access Manager interface defines and publishes the page-oriented interface and semantics of the host DBMS's built-in storage layer. This storage layer serves as a common basis for custom access method implementations. But the availability of the interface definition and semantics may also serve for a second purpose. It offers the opportunity to allow *custom storage layers*, implementing the given storage interface. Such custom storage layers may serve as alternative to the built-in storage facility, tailored for particular access methods or storage hardware. Implementing a new storage layer is a highly intricate task and its benefits

are questionable. The same is true for the integration of existing storage layers, such as those discussed in *3.2.5 MySQL*. The general possibility of implementing or integrating custom storage layers using the Access Manager interface is mentioned here merely for the sake of completeness.

Of higher practical relevance is the incorporation of a *data integration layer* into the host DBMS. This layer provides a tuple-oriented interface towards the host system, while accessing remote data stored outside the DBMS. External data is registered within the database schema as ordinary relational table definitions, composed from standard SQL data types and without further indications that the data is not stored locally. Only when the data is actually accessed, the requests are delegated to the integration module, which is responsible to provide access to the remote data source. Depending on the capabilities of the integration module, such a relation may offer coherent access to remote data, i.e. reading and writing within a transactional context, while upholding location transparency. The full integration into the database schema allows also for creation and automated maintenance of internal secondary index structures on external data. Data sources may be local storage devices, or files storing data in a proprietary format, different from that of the DBMS storage layer, e.g. relational data in structured text files (CSV), XML documents, and other data sources. But the integration layer may also access remote services and data sources, for example a remote database server. With a collection of data integration layers, it becomes possible to create a uniform relational view over a network of interconnected heterogeneous data sources.

This form of data integration exceeds the flexibility of primitive table functions accessing remote data by far, since its approach to *global* query optimization offers query planning capabilities close to those of a homogeneous federated DBMS. Negotiation and configuration of remote table accesses allow relocating configuration directives to remote systems. This includes moving restrictions to a remote system, in order to reduce the data transfer. But also global optimization of sort orders and projections offer massive savings compared to the limited possibilities of table functions. But most important, global cost-driven optimization allows active and effective manipulation of the join sequence. Details on alternative storage layers will be addressed in section *4.6 Data Integration*.

The primary advantage of this component-oriented architecture, where all access methods implement identical interfaces, is the aforementioned flexibility arising from the possibility of arbitrary recombination of all available building blocks. On the other hand, it is not always possible to force different access methods into identical interfaces. Such a one-size-fits-all

interface is bound to be highly complex. Implementing a superset of strictly required interface functions contradicts the design goals for rapid prototyping and the tight coupling of implementation complexity and prototype functionality. Therefore, the Access Manager interface is constructed as a lean mandatory interface, augmented by additional optional interface routines. The following Figure.21 provides an overview over the tuple-oriented part of the Access Manager interface, the *access method interface*.

The compulsory part of the access method interface comprises all routines required for providing the functionality of a relational operator conforming to the Iterator model. As any other relational operator, an access method can be opened, iterated, and closed. It also possesses all means required for its configuration and cost assessment during query optimization phase. Finally, it provides interface routines for access path creation and deletion. Only these last-mentioned aspects of an access method's mandatory interface exceed the functionality required for a common relational operator.

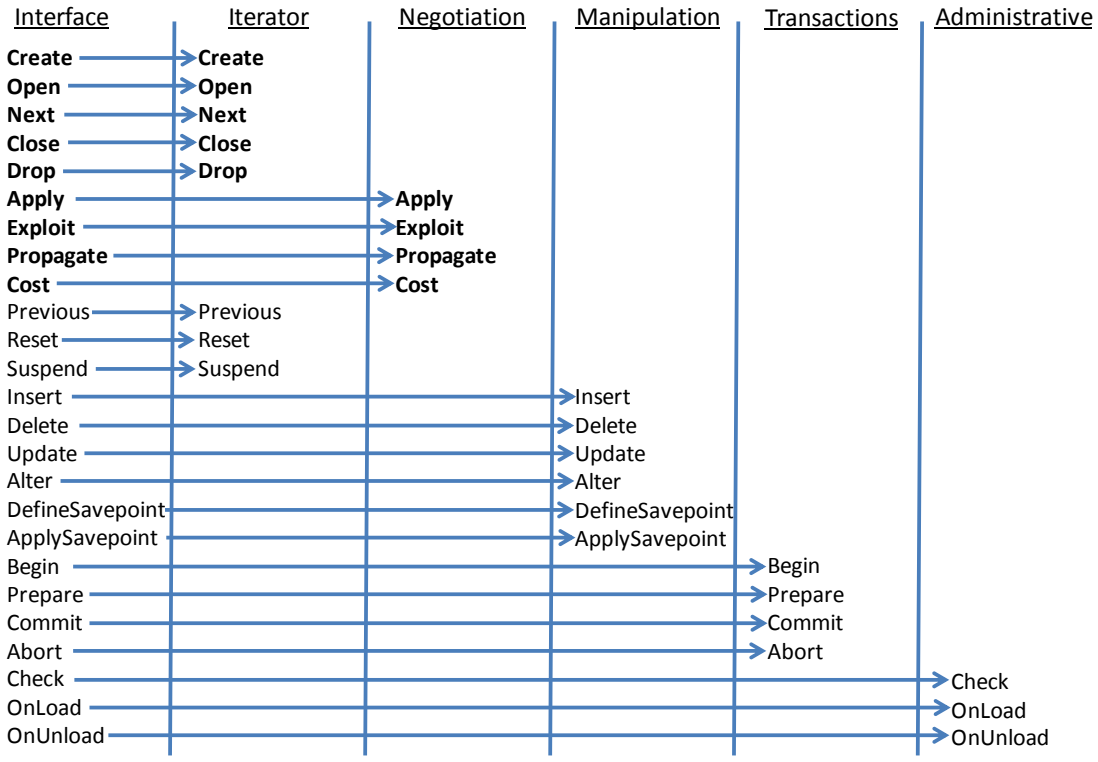


Figure.21 Composition of the tuple-oriented access method interface. Complete overview of the tuple-oriented access method interface, classified into its diverse tasks. The compulsory interface routines (in bold print) provide the functionality required for the Iterator model in query execution, and instruments used for negotiation during query optimization. The optional interface routines cover data manipulation assets, efficient alternatives to compulsory routines, call-back hooks for transactional transitions, and some administrative functionality.

Interface routines for data manipulation are optional, and the absence of their implementation indicates explicit read-only access methods. For implementing fully operational data manipulation, the presence of implementations of **`Insert()`** and **`Delete()`** routines is functionally sufficient. The remaining optional interface routines serve mostly for efficiency purposes. They provide a shortcut or an inexpensive alternative for calling an expensive sequence of compulsory interface routines. For example updating a tuple with the optional **`Update()`** routine can be simulated with the deletion of the old tuple, followed by insertion of the new one. Other optional interface routines provide efficiency by means of graceful fault recovery. The SQL standard demands that data manipulation is an atomic operation. If a dynamic error is encountered during manipulation, e.g. an integrity constraint violation, arithmetic overflow etc., then the complete manipulation has to be undone. The coarse method is to abort the enclosing transaction, which certainly satisfies SQL's claim for atomicity. But this approach has the drawback of revoking work that already has been completed successfully by preceding operations within the same transaction. As an alternative, the Access Manager will use the optional *savepoint* feature of an access method implementation, for providing the required fault tolerance. The interfaces addressing transactional transitions are call-back hooks, informing an access method about changes in the present transactional context. We will demonstrate that they are not required in standard access method implementation, but they are useful under specific circumstances. A detailed discussion of the functionality of the remaining optional interface routines will be provided in the course of following section.

The decision of using compulsory or alternative interface routines is made by the Access Manager in accordance to a strict interface protocol, which is part of the Access Manager specification. The decision is based on one simple assumption: if an optional interface is implemented, then it will be used, as it is considered superior to the 'normal' way of operation. When access modules call routines of other access modules, as intermediate access methods do by directly calling their auxiliary access methods, they are bypassing the Access Manager framework. For these direct calls, there is no instance involved for enforcing the mentioned protocol for optional routines. Still, intermediate access methods *should* follow the interface protocol, because an auxiliary access method implementing optional interface routines *expects* that these routines are used. An intermediate access method ignoring the interface protocol will still function correctly, but it may suffer from inaccurate cost estimation, since the cost function of an auxiliary access method silently assumes the use of optional routines when assessing the costs of a planned operation (e.g. update vs. delete/ insert). Only the strict compliance of all partaking access modules to the interface protocol will ensure a

consistent and predictable behavior of the overall system. As a direct consequence, an intermediate access method strictly following the Access Manager protocol is able to exchange underlying auxiliary access methods without any further adaptations of its own code basis, since every replacement supports the Access Manager protocol, even if it implements a different subset of the optional interface.

The page-oriented *storage layer interface* (Figure.22) has a similar design, consisting of mandatory and optional routines. The primary purpose of its compulsory interface routines is to provide page-oriented storage facilities. Besides persistent storage, it also offers extensive influence on the buffer manager's page caching strategies, with its ability to deliberately and selectively retain pages in cache. In contrast to the tuple-layer, the storage layer incorporates mandatory transactional interface routines, controlling transactional consistency of access methods constructed on top of this layer. In addition, it offers control over the host system's locking facilities on page granularity, as every page access is attributed with a lock type (read/ write/ exclusive). The subset of optional interfaces is limited to functionality for the savepoint feature on page granularity and some administrative functionality to be discussed later. With its built-in storage layer, the host system housing the Access Manager framework provides a complete and fully operational implementation of the storage layer interface.

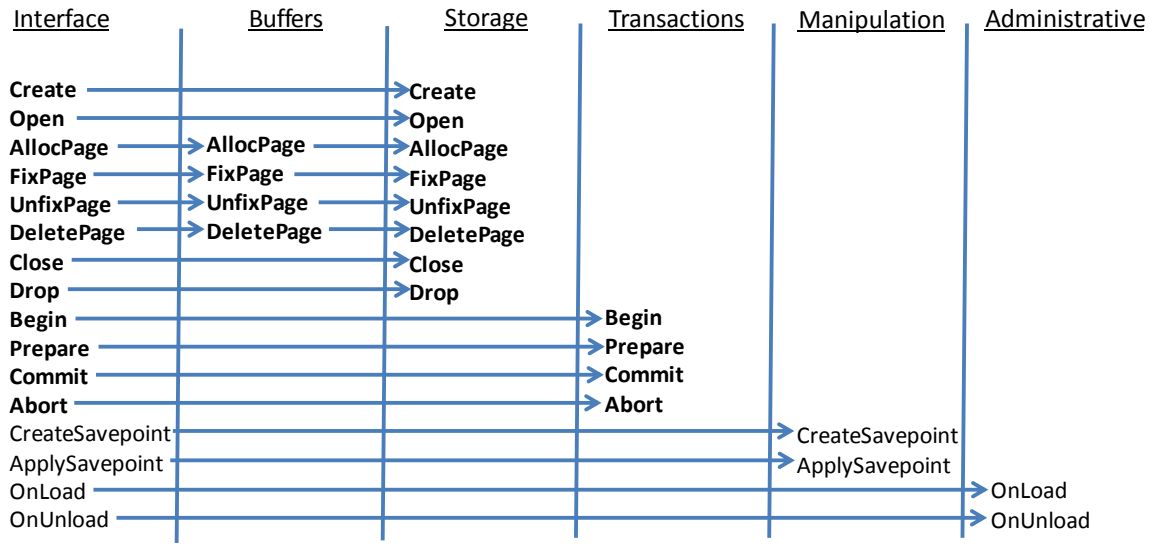


Figure.22 Storage layer interface. This lean interface comprises all functionality required for page-oriented persistent storage, and in particular, it provides control over the host system's buffer manager, which is an integral component of the storage layer, and operates via side-effects of the page access routines. In addition, the interface integrates access methods with the host system's locking and transactional consistency services.

The tuple-oriented interface routines described in Figure.21 are routines to be implemented by external access methods for integration into the host system. The Access Manager, as a component of this host system, is responsible for operating these interface routines, thereby enabling the host system to use custom access methods. The access methods themselves may only call tuple-oriented interface routines of other access methods, or use the storage layer interface as a gateway to the page oriented storage facilities. Hence, an access method is fully encapsulated within the Access Manager framework, having no direct influence on other subsystems of the host system. In addition to the storage layer, the Access Manager comprises auxiliary interfaces to some other selected host system components. These interface routines are combined into the narrow *system utility interface* (Figure.23).

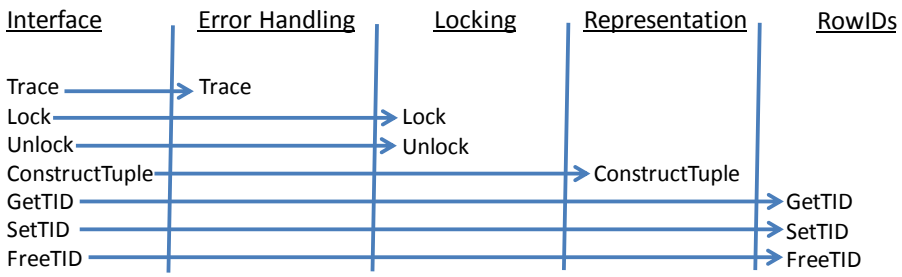


Figure.23 System utility interface. Additional gateways for access methods to useful host system facilities. They offer direct access to error handling and reporting, fine-granular locking based on the system's lock manager, construction of tuples in standard representation and provision of tuple identifiers (TIDs) for efficient secondary indexing.

The most important among these interfaces is the gateway to the host system's centralized error reporting system. The **`Trace()`** method is called for indicating error situations and warnings. It allows issuing descriptive textual messages to be displayed to the database user or to be inserted into the host system's event log. An access method may implement fine-granular locking by utilizing the host system's intrinsic lock manager implementation via its public **`Lock()`**/ **`Unlock()`** interface. With **`ConstructTuple()`** the system implements a central routine for constructing tuples in standard representation. Finally, the interface controls a facility for generating artificial tuple identifiers (TIDs), a compact surrogate for conventional primary keys, used mainly for tuple identification in secondary indexes. The purpose of this set of interface routines is tighter integration and further reduction of implementation complexity of new access methods, by supplying reusable utility functionality.

The remainder of this chapter will describe the routines of the three presented interface families in more detail and demonstrate how they collaborate according to a strict protocol, for providing the functionality of the Access Manager framework. Therefore we will proceed

bottom-up, starting with the storage layer interface which is implemented by the host system's built-in storage layer and serves as foundation for most access structures.

4.2. Built-in Storage Layer

As part of the Access Manager specification, the host DBMS provides a page-oriented interface to the host system's built-in storage facilities, implementing the Access Manager's storage layer interface (cf. Figure.22, page 102). This storage layer is a key service of the host DBMS in providing reusable functionality and thereby relieving the access method implementer from recurring implementation tasks. Besides plain persistent storage services, the storage layer also encapsulates caching functionality on page granularity, including the possibility of actively influencing cache frame replacement strategies. In addition, every access to a page can be optionally attributed with a lock type. These page accesses are automatically recorded by the system's intrinsic lock manager for enforcing a convenient locking mechanism on page granularity. The combination of the host system's caching and locking facilities to a multi-version concurrency control on page granularity allows any access method implementation to benefit from a readily available low-contention concurrency protocol. If page accesses are attributed with locks, then concurrency on page granularity is controlled by the host system. Therefore it generates and maintains multiple versions (copies) of database objects, as required for supplying concurrent transactions with the correct versions. Finally, the storage layer provides extensive and fully transparent recovery functionality. With this, the host system is in the position to restore the database state at the beginning of a transaction, *without* the requirement of *any* supporting functionality in the involved access modules. In addition, the host system possesses sophisticated *undo* functionality of individual DML operations, which is required, if data manipulation fails. The SQL standard demands such atomicity of data manipulation. This functionality is subsumed in the so-called *savepoint* feature, allowing the storage layer to restore the database state at the starting point of data manipulation, without aborting the enclosing transaction. In order to function, the savepoint feature has to be supported by all access modules participating in a DML action, by implementing a relatively simple protocol. The bulk of the savepoint functionality is then provided by the storage layer.

This combination of the storage subsystem services achieves compliance with the ACID paradigm in its claims for atomicity, consistency, isolation, and durability on page granularity. Only arrangements for guaranteeing consistency in manipulation of data structures constructed on top of the storage layer fall into the functional scope of corresponding access

method implementation. These important ACID assertions of the storage module are achieved by following an intelligible and simple protocol when using the storage layer interface.

4.2.1. Storage

A DBMS organizes its data in units of one or more files of considerable size (cf. Figure.24). Alternatively, it is also possible to use other extensive physical storage units like partitions or even complete devices. We refer to such physical storage units as *extents*. Which type or combination of types of extents is actually used, is ultimately irrelevant. The DBMS storage system conceals these storage allocation details and provides a uniform, contiguous *address space*, whose size corresponds to the concatenation of all available storage assets. This storage area, which can be subsequently extended by adding extents, represents the physical address space available to the DBMS for persistent storage. The arrangement and utilization of these physical storage units are important means for physical database schema design. For example, it is possible to increase storage throughput by partitioning data on independent physical devices. Alternatively, access times decrease when related data is closely clustered on one storage unit. Regardless of the configuration of storage units, the logical DBMS schema remains independent from such physical considerations.

The address space is then partitioned into *pages* of equal size. Each page represents a closed interval of consecutive addresses, which is identified by a unique consecutive number, the *page number*. Each page has a *page header*, a small administrative storage area at its beginning, while the remainders of the page are used for arbitrary storage purposes. Data stored on one page is retrieved or written as one logically atomic I/O operation. Consequently, the typical page size corresponds to the amount of data the underlying hardware processes as one physical block I/O operation, i.e. a logical page corresponds to one or more physical *blocks*. Data is also maintained in page-sized fragments by the DBMS caching facility, and finally, pages represent an opportune granularity for administering concurrency for parallel operations by page-wise locking data for shared and exclusive operations.

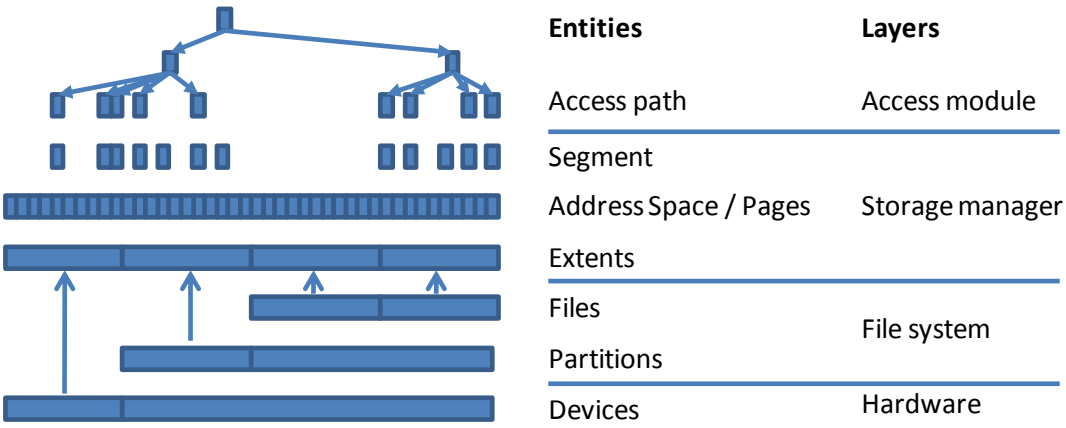


Figure.24 Storage units on different layers. The DBMS may use various assets for providing physical storage space, e.g. devices, partitions, or files. The system's storage layer provides a uniform view on the complete storage area as one consecutive address space. This space is subdivided into equal-sized pages, such that one page corresponds to one or more physical blocks of the underlying device, thereby providing manageable memory fragments. Collections of pages are combined to segments, each representing one access path. The storage manager maintains the assignment of each page to its segment by recording the page number of the associated segment's initial page in every page header. The logical structure of an access path is modeled by pointing to other pages of the same segment by referencing pages numbers inside a page's data area.

Individual pages are combined into larger logical storage structures. Such collections of pages are called *segments*. Segments are the storage entities containing relations. Every access path is stored in a separate segment, i.e. every base relation but also every secondary index is stored in a dedicated segment.

Initially all pages in the DBMS's address space are in the pool of free pages. They are not assigned to any segment. A segment is created, using the storage manager's **`Create()`** routine. It assigns one single free page to the new segment. A segment can grow by adding further pages via the **`AllocPage()`** routine. This routine offers explicit control of the location of a new page, in order to keep segments spatially clustered in the database's physical address space. Therefore, an optional parameter of the **`AllocPage()`** routine allows the specification of a desired page number for a new page. If the specified page is free, it will be allocated. Otherwise, the page is already in use and the storage manager tries to allocate the next free page within the same extent. Only if the specified extent is full, a free page in another extent will be allocated. If no target page number is specified, then the page allocation strategy is left entirely to the storage manager. Conversely to allocation, a segment shrinks by removing a page and returning it to the pool of free pages (**`DeletePage()`**). Due to these dynamics, segments do not necessarily cover contiguous address ranges. From the storage

manager's perspective, they are merely a collection of loosely coupled pages. This coupling is enforced by marking each page header with the *segment number*, i.e. the identification of the containing segment. The segment number is identical to the page number of the initial page of a segment, which was allocated by the **`Create()`** routine. This distinguished page represents the complete segment and therefore it cannot be freed while the segment exists. In addition to the segment number, the page header stores its own page number and a checksum value (CRC) over the complete page contents. These entries in the page header are not required for normal operation of the storage manager, but the information serves as fault detection mechanism. An access method provides page number and segment number as parameters when requesting access to a page. If this information deviates from the information maintained independently in the page header, this indicates a failure in the structural integrity of the access method, or malfunctioning hardware.

The storage manager's main purpose is to maintain memory management by recording whether an individual page is currently in use or not. Documenting affiliations of pages to segments, and even the whole concept of segments, is not required to this end. The task of implementing cohesion on a set of pages belonging to the same segment is delegated completely to the access method layer above. It maintains chaining between individual pages, representing the physical structure of the access method.

An access method may access existing pages by using the **`FixPage()`** routine. This routine takes the page number as input parameter and creates a copy of a persistently stored page in main memory, where it is held at a fixed location. The routine returns a stable pointer to this page for subsequent access. An additional parameter of the **`FixPage()`** routine controls the intention of the page access, i.e. whether a page is retrieved for read-only access or for modification. A page acquired using **`AllocPage()`** is implicitly fixed for writing access. When access to the page is completed, the access method may call the **`UnfixPage()`** routine in order to release the fixed page. Thereby the main memory allocated for the page is released. If the page was modified, the storage system guarantees that changes are stored persistently when the enclosing transaction is committed. The detailed implications of calling **`AllocPage()`**, **`FixPage()`** and **`UnfixPage()`** on the caching and locking facilities of the storage manager will be discussed separately.

Finally a segment is removed by calling the storage layer's **`Drop()`** routine. The access method maintaining the logical structure of the segment has to guarantee that all other pages belonging to this segment have been freed using **`DeletePage()`** prior to calling **`Drop()`**.

4.2.2. Caching

In addition to pure I/O services, the storage manager also provides a sophisticated caching facility. Access to the system cache is integrated into the storage layer interface, and automatic caching takes place when operating the already introduced page access routines. Caching operates, like all other storage layer assets, on page granularity and uses a least-recently-used (LRU) replacement strategy. Whenever a page is accessed, either by **`AllocPage()`** or **`FixPage()`**, the contents of that page are transferred into an empty cache frame in main memory. As long as the page is accessed, its frame is not on the LRU stack, i.e. the cache frame cannot be replaced. The page stays 'fixed' in the main memory. A page may remain in this fixed state for an arbitrary period of time, and concurrently access to this page becomes possible. This happens when a page is accessed from different transactions (concurrent read access), or from different scans of the same transaction (concurrent read/ write access). Every fix will increase the fix-counter of that page. After a page access is completed, the **`UnfixPage()`** routine will decrease the page's fix counter. Any process that fixes a page is also responsible to issue an identical number of unfix operations. When the fix counter eventually reaches zero, the page is unfixed and its frame moves on top of the LRU stack, since it now holds the most recently used (MRU) page. From there it starts descending through the LRU stack, as other pages are accessed. Any subsequent access will move it again into MRU position, but if the stack frame eventually reaches LRU position, its contents will be replaced with another page that is currently needed. If a page is deleted via **`DeletePage()`** while on the LRU stack, the cache frame is freed and becomes immediately available for holding another page.

This simple interface offers all functionality the access method implementer requires for efficient access to recurrently used pages. Caching requires no implementation effort whatsoever from the access method. In addition, it offers a certain degree of control over the caching strategy. An access method may periodically fix/ unfix pages of high value, thereby averting their replacement. Critical pages may even remain fixed over a longer period of time, inhibiting their replacement altogether. However, interference with the host system's caching strategies should be conducted with extreme care and only on a small scale, i.e. for few pages. One should keep in mind that the system cache represents a shared resource, used not only

across concurrent scans of one single query, but across concurrent transactions. Bending the cache frame replacement strategy in favor of one access method will penalize other access structures, and thereby incur potentially high overall costs that actually exceed the locally induced savings. In particular, if two or more access paths based on the same greedy access method are competing for the limited resources of the system cache, their endeavors are likely to cause cache contentions instead of bringing the originally intended speed-ups.

4.2.3. Locking & Concurrency

Locking and concurrency in DBMSs is implemented as combinations of locks on objects of various granularities. These granularities are organized on hierarchical levels, where typical scopes for locking are relations, pages, and records. Locks on complete relations also include implicit locks on redundant secondary access paths. In contrast, locks on individual pages only affect one single page, without immediate side-effects. If a page is updated, maintaining data consistency compels locking and updating dependent and redundant information. Row-level locks offer the most fine-grained locking strategy, allowing maximized concurrency at the expense of increased locking complexity. Row-level locking falls into the responsibility of the access method implementation that maintains the segment, while locking of relations and pages is provided as a service by the storage layer.

Locking in the Access Manager is presuming multi-version concurrency control, based on the well known RAX protocol [Bay80] (cf. Figure.25). Locking is therefore organized in three separate lock types. The **`READ_LOCK`** (also R-lock) allows parallel reading activities of concurrent transactions on the same object. The **`WRITE_LOCK`** (A-lock) is used, if an object is to be updated. Only one concurrent write operation on one object is possible at any time. Therefore, the transaction intending to modify an object creates a private copy (version) of that object. The original version (*before-image*), representing the object's state before the update took place, remains accessible for concurrent readers. The private copy of the update transaction is called *after-image*. When the update transaction is committed, it acquires **`EXCLUSIVE_LOCK`**s (X-locks) on all objects it modified in its course, i.e. **`WRITE_LOCK`**s are converted to **`EXCLUSIVE_LOCK`**s. If concurrent transactions still hold **`READ_LOCK`**s, the update transaction has to wait until these locks are released. Only when all objects are locked exclusively, the buffer manager will discard all before-images and the after-images eventually reflect the new state of the object. Conversely, if an update transaction is aborted, all after-images are deleted. This may be conducted without acquiring exclusive locks, since the after-

images are already private to the update transaction. The following matrix depicts the compatibility of the various lock-modes.

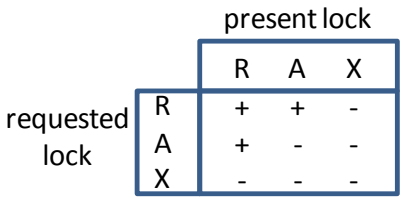


Figure.25 RAX compatibility matrix. A requested lock can be granted, if it is compatible with the present lock of another transaction (+). Otherwise (-), the requesting transaction has to wait until the conflicting lock is released.

For locking hierarchical structures, this lock protocol is augmented with the RIX protocol (Figure.26). In addition to the RAX-locks, this protocol also uses intention locks, e.g. IR (intention-read) and IX (intention-exclusive). This allows to use actual locks (R/X) on a fine-granular level, while the parent object is locked using the corresponding intention lock. If intention locks collide during lock acquisition, their compatibility is decided by consulting lock compatibility on a finer granularity.

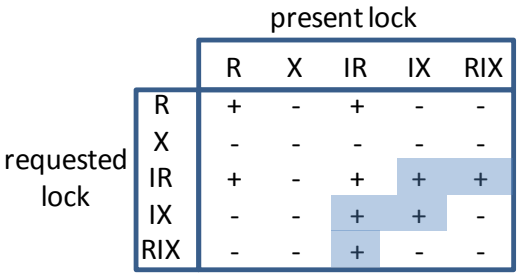


Figure.26 RIX compatibility matrix. I-locks are used on a coarse granularity, e.g. relations, for expressing the intention to perform a certain operation on a finer granularity. Intention locks are used in combination with R- and X-locks. Where potentially incompatible intention locks collide, compatibility is decided by on a finer level of granularity (shaded areas). Finally, locks on different granularities may also be mixed, as RIX acquires an R-lock on a coarse granularity, and an IX lock for expressing its intention to lock individual portions of the relation. Hence, the RIX-lock is used by scans reading a complete relation while modifying only selected portions.

The RAX/RIX lock protocols are used for implementing isolation of diverse qualities (isolation levels) between individual transactions. A low isolation level offer increased concurrency and throughput of the overall system. Higher concurrency is gained by accepting the possibility of certain anomalies, e.g. lost updates, dirty reads, non-repeatable reads and phantom reads. The isolation level of a transaction controls which form of update anomalies are tolerated by a database application. Isolation levels are implemented by controlling the types of applied

locks, and the lock durations. Figure.27 gives an overview over properties and behavior of the ANSI SQL isolation levels.

Isolation Level	Lock Types and Duration		Anomalies			
	WRITE	READ	Lost Update	Dirty Read	Non-rep. Read	Phantom Read
READ_UNCOMMITED	EOT	-	-	+	+	+
READ_COMMITTED	EOT	EOS	-	-	+	+
REPEATABLE_READ	EOT	EOT	-	-	-	+
SERIALIZABLE	EOT	EOT	-	-	-	-

Figure.27 Isolation levels as specified in the SQL standard. The respective behavior is achieved by variation of applied lock types. Write anomalies (lost update/ dirty write) are never tolerated, hence write-locks are acquired for every update operation, and held until end of transaction (EOT). Conversely, data may be accessed without explicit read-lock, causing diverse read anomalies. If read-locks are acquired and held until end-of-statement (EOS), then dirty reads are eliminated. Holding read and write-locks until EOT (2-phase locking) prevents all anomalies.

Each isolation level offers its own benefits and drawbacks, when trading concurrency against update anomalies. Isolation levels allow configuring a DBMS for a specific database application, depending on tolerance and throughput demands of a given task.

Regarding the Access Manager framework, almost the complete locking mechanism described above is made available by the host system. The DBMS possesses a fully-fledged lock manager component, providing the required functionality. The necessity of table locks and their dependencies on secondary access paths is recognized by the SQL compiler by consultation of the data dictionary. The query processor will automatically acquire necessary locks from the lock manager at the beginning of a query execution. Hence, access structures may remain completely ignorant of locking on table granularity. Locking on page granularity on the other hand is intrinsically tied to the page-oriented storage layer. Whenever a page is requested by an access module, this page is attributed with the appropriate lock type. This is accomplished with one parameter that has to be supplied with every access to a page, describing the intention of the page access. These intentions are represented by two options: **`READ_LOCK`** or **`WRITE_LOCK`**. In case of **`WRITE_LOCK`**, a working copy (after-image) for the writing transaction is generated automatically. This form of locking resembles a *pessimistic* locking strategy, where locks are acquired as the transaction proceeds. If locking on page granularity is to be achieved by other means, then the host system's page locking facility can be bypassed by using the **`NO_LOCK`** attribute. When a page is accessed, the system acquires the appropriate locks for the given intention. In addition, it is possible to convert existing locks into more restrictive locks. If, for example, a page is initially accessed for mere reading

activities and the appropriate read-lock is already in place, then a later writing access is still possible. The demand for updating a page with an existing read-lock is expressed by fixing it again with the **`WRITE_LOCK`** option. The read-lock is thereby converted into a write-lock and the routine returns a reference to a working copy (after-image) of the original page. The necessity of making an after-image causes a noteworthy complication of lock conversions. The after-image is naturally stored at a location in main memory that is different from that of the before-image. It henceforth represents the writing transaction's private version of this page, and all other scans belonging to that transaction and operating on the same segment must be able to see modifications on the after-image immediately. The resulting challenge is to redirect all open scans of the writing transaction to the after-image. If a scan of this transaction fixes this particular page in future, the storage manager will automatically supply the correct version, i.e. the after-image. Problematic are only lock-conversions where the page is *currently* fixed by another scan using read-locks. Any references to this page that are currently held by such a scan are still pointing to the before-image. Therefore, the scan causing the lock conversion has to redirect such references of concurrent scans of the same transaction to the after-image. This is accomplished by direct manipulation of the internal structure of concurrent scans. The compensation of side-effects of data modification onto concurrent scans of the same transaction is subsumed under the notion of *scan maintenance.* Scan maintenance cannot be delegated to the host system, since the host system cannot know the internal structure describing the scan status of a custom access method. But the scan responsible for the lock conversion is inevitably of the same type as all other scans on the same segment, making immediate scan maintenance viable.

If lock requests are not immediately grantable, because an object is currently blocked by an incompatible lock of another transaction, then the transaction will wait for a configurable amount of time (lock timeout), until the conflicting lock is released. If this time period expires, and the lock is still occupied, then an error is raised. Finally, if a lock request would generate a deadlock with locks held by another transaction, then the system will detect this situation and resolve it by rolling back the transaction causing the deadlock and thereby releasing all its locks.

At this point, it is important to notice the absence of any provisions for unlocking objects. Also programmatic relaxation of existing locks is not possible, e.g. it is legal to subsequently fix a page with a **`READ_LOCK`** that was previously modified by the same transaction under **`WRITE_LOCK`**, but this will not affect the present lock, i.e. the **`WRITE_LOCK`** will remain in

place. Unlocking is controlled completely by the host system and the duration of locks is invariably managed via the system's isolation settings. Locks will be released automatically at the end of the current statement's evaluation (EOS) or at the end of the current transaction (EOT), depending on the isolation setting. In the latter case, locking is consistent with the 2-phase lock protocol (2PL), a prerequisite for guaranteeing serializable schedules for SQL's highest isolation level. The host system will also convert **`WRITE_LOCK`**s into **`EXCLUSIVE_LOCK`**s in preparation of committing a transaction. Automated lock conversion and unlocking relieves the access method programmer to a large extend from the burden of dealing with locking and concurrency on table and page granularity.

In summary, the storage manager provides pessimistic, hierarchical locking facilities, capable of pursuing configurable isolation levels from dirty reads to serializable transaction schedules. Despite of its simplicity, this interface is sufficient for fully operational locking on page granularity, which is complemented by the storage layer's page shadowing mechanisms required for low contention multi-version concurrency control. As a consequence, explicit calls to the system utility interface's **`Lock()`** and **`Unlock()`** routines are not required when locking on page granularity. Further details concerning locking on finer granularities are discussed in section *4.3.10 Locking & Concurrency* on tuple-oriented access structures.

4.2.4. Transactions & Consistency

For providing the properties postulated by the ACID paradigm, the storage layer must always operate inside a transactional context. Consequently, any scan operation accessing the storage layer is strictly associated with a transactional context. This means that scans can neither operate outside transactions, nor across transactional boundaries. These fundamental prerequisites are guaranteed by the Access Manager framework. It will operate the access methods in such way that open scans are always closed when a transaction ends. In addition, it will automatically call the storage layer's respective interface routine, namely **`Begin()`**, **`Prepare()`**, **`Commit()`**, or **`Abort()`**, for indicating transactional transitions to the storage module. Access method implementations cannot call these routines directly, as they are operated exclusively by the Access Manager framework. They are mentioned here as integral parts of the storage layer interface, and are therefore of relevance for implementing alternative storage layers.

The strict association of all storage manager operations with a transactional context is necessary for ensuring well-defined behavior of the storage layer's multi-version concurrency

control, i.e. every fix and lock operation must be intrinsically tied to a transaction. Therefore, every transaction is labeled with a unique transaction ID (*TAID*). TAIDs are generated by the host system as a sequence of monotonous integer numbers and every transaction obtains a TAID in accordance to the point in time of its beginning. On the Access Manager interface, TAIDs become visible as function parameters. They are generated by the host system and passed on to access method routines, for denoting the transactional context of the currently conducted operation. Access methods are responsible for passing TAIDs on, when calling routines of the underlying storage layer, such that every storage operation is inevitably assigned to the correct transactional context.

The TAID also becomes an integral part of any data page, as the transaction that made the last change in a page is inscribed as the producer (*PID*) in every page header. There it serves as discriminator for existing versions of one page. Whenever a transaction attempts a modification of a page (A-lock), a copy of this page is generated. The before-image remains completely unmodified, including its original producer transaction (PID). The after-image will bear the ID of the transaction that acquired the A-lock as its new producer. For guaranteeing correctness, the host system's lock manager maintains a precedence graph, where all locked objects of all active transactions are registered. This graph serves for detecting conflicting lock sequences and possible deadlock situations. In addition, it allows the lock manager to arrange transaction into a virtual schedule, a prerequisite for guaranteeing serializability. Finally, the lock manager cooperates with the host system's cache manager, which is responsible for presenting a transaction accessing a page with 'correct' version of that page, in accordance to the transaction schedule determined by the lock manager. For this step, it is necessary to correlate page versions and the IDs of requesting transactions, which is technically conducted with TAID and PID. Apart from the necessity of associating every storage layer operation with the corresponding TAID, an access method may remain completely ignorant of these mechanisms. The storage layer will automatically provide correct and consistent versions of requested pages in accordance to the indicated transaction context, as a service of its built-in multi-version concurrency control.

4.2.5. Logging & Recovery

Database systems generally distinguish two forms of recovery mechanisms, namely transaction recovery and disk recovery. Transaction recovery provides the functionality for undoing modifications of one single transaction. During normal operation, transaction recovery is required if a transaction is explicitly aborted or if it runs into a dynamic error, e.g. an integrity

constraint violation, and the transaction cannot complete by reaching a consistent database state. Transaction recovery is also required for crash recovery. If the database system fails because of a severe software or hardware fault, operations stop in mid-transaction. When the system is recovered, all uncommitted operations are undone and the system is restored in its last consistent state before the crash.

Disk recovery is a protective measure against data loss in case of an unrecoverable disk failure of the primary storage device. First a backup of the data (snapshot) is made at an arbitrary point in time, which is stored in a safe place, i.e. not on the same device as operational data. During normal operation, the DBMS maintains a log of all modifications. Logging information and operational data are stored on separate devices. If the disk holding the primary data fails and the operational data is lost, then the now outdated backup copy is restored to a new disk and all operations are redone with the modification log, until the backup finally reaches the last consistent state before the hardware failure.

For transaction and disk recovery, the database has to maintain a log of database operations. This log consists of instructions for physical modifications of individual disk pages. For transaction recovery, the logs are used to undo changes in the database, i.e. they are processed chronologically backwards. Redoing changes, as required for disk recovery, processes a log in forward direction.

Transbase, as the host system for the Access Manager prototype implementation, offers two distinct forms of logging, namely before-image logging and delta logging. Depending on its purpose for disk or transaction recovery, a log may contain either complete copies of pages (before-images) or calculated deltas that describe the transition between two states. Before-image logging is only appropriate for transaction recovery, because before-images only describe the transition of a page to its previous state, which is useful for undoing changes but inadequate for redo operations. A before-image of a modified page consists of the complete page, even if the applied changes are significantly smaller. Before-images of newly allocated pages are not stored, since these pages do not have an original state to be recovered. With their limitation to transaction recovery, before-images become obsolete and may be discarded as soon as modifications on its after-image are committed. This makes before-image logging attractive for mass insertions and extensive updates, since bulky recovery data is stored only as long as strictly necessary. Before-image logging may be used by arbitrary access methods, as it reliably specifies the physical transition of a complete page, regardless of the logical

operation on that page. Therefore, it can be used by the Access Manager for transaction recovery, without any further provisions from the access method that is operating on the page.

Delta logging describes the transition between two consecutive page versions on byte level. A page delta can be applied for transforming a page from an earlier version into a newer version, but also in the opposite direction. Hence, delta logging is suitable for transaction and disk recovery. Delta logging is in general substantially more compact than before-images, since only the modified portions of a page are stored, but its handling and calculation is more complex. Hence, delta logging is particularly well-suited for logging scattered, small-scale modifications, while it shows no advantage over before-image logging for bulk insertions and bulk updates. The compactness of delta-records is achieved by using convenient descriptions of logical page transitions. B-trees, for example, would describe page splits and merge operations with dedicated shift operations. Here the pages are altered by insertion or deletion of one tuple. This triggers moving bulks of adjacent data, although this data remains otherwise unchanged. In a delta log, these operations are optimally described as shift-operations, in conjunction with small delta records describing actually modified data. In general, this means that every custom access method would be able to define its own log records for providing tailored descriptions of page transitions. As a consequence, proprietary log records originating from a diversity of access methods would become intermixed in the database logs.

Despite the advantages such an integration of custom access modules into system recovery would bring, we dismiss it in favor of implementation simplicity and system stability. Explicit delta logging mechanisms in custom access modules will considerably complicate the design and implementation of access methods. Moreover, the integration of external code into the crucial recovery subsystem and the existence of proprietary log entries in the database logs, are likely to jeopardize overall system integrity. As an alternative, the Transbase prototype tries to derive compact delta records by analyzing before- and after-image of a page in the storage manager. Although this is not always optimal in terms of storage complexity, this approach is still able to generate compact log records while upholding system integrity with a sealed off, reliable code basis. With this, logging is removed completely from the scope of access method programmers. It is automatically provided by the host system, without any requirement of explicit support from custom access methods.

4.3. Access Method Interface

This chapter will study the functionality provided by the tuple-oriented access method interface (cf. Figure.21, page 100) of the Access Manager framework. This interface definition and its accompanying interface protocol designate the required functionality and behavior of an access method for integration into a DBMS. As starting point, we consider an extensible DBMS host system, which is exhibiting the Access Manager framework for assimilating custom access methods. These custom access methods are implementing the *access method interface* and each exists initially as a dynamically loadable and linkable library called *access module*. The host system's Access Manager framework allows integration of such modules into the host system's code basis and operates them by calling the libraries' exported access method interface in conformance with the Access Manager protocol. The access modules themselves are considered as black boxes, i.e. they offer a public interface and a known protocol, but assumptions on their internal workings are not permitted. We will now examine various use-cases of custom access method implementations, in order to illuminate valid sequences for calling access method interface routines, as defined by the Access Manager protocol.

All routines provided by an access module are operated by the Access Manager framework, representing the host system's only gateway to access modules. During query evaluation, the access module's routines are called by the *scan operator*, a generic relational operator, encapsulating an arbitrary access module (Figure.28). This operator is a central component of the host system's Access Manager framework. During query compilation, the scan operator is statically bound to a segment $\mathbb{S}$, and consequently also to the access module type $\mathbb{T}$ associated with that segment. In addition, the scan operator is assigned to a specific task, i.e. retrieval, manipulation, or materialization of data. The resulting specialized relational operators are named according to their primary task: *Scan, Insert, Delete, Update,* and *Materialize*. After binding and specialization during query compilation, the following query optimization phase will use negotiation for providing optimal interoperability through additional configuration. The resulting configuration also becomes part of the scan operator, where it is used for calling the routines of the embedded module in accordance to the Access Manager protocol, for accomplishing the assigned task under the given configuration.

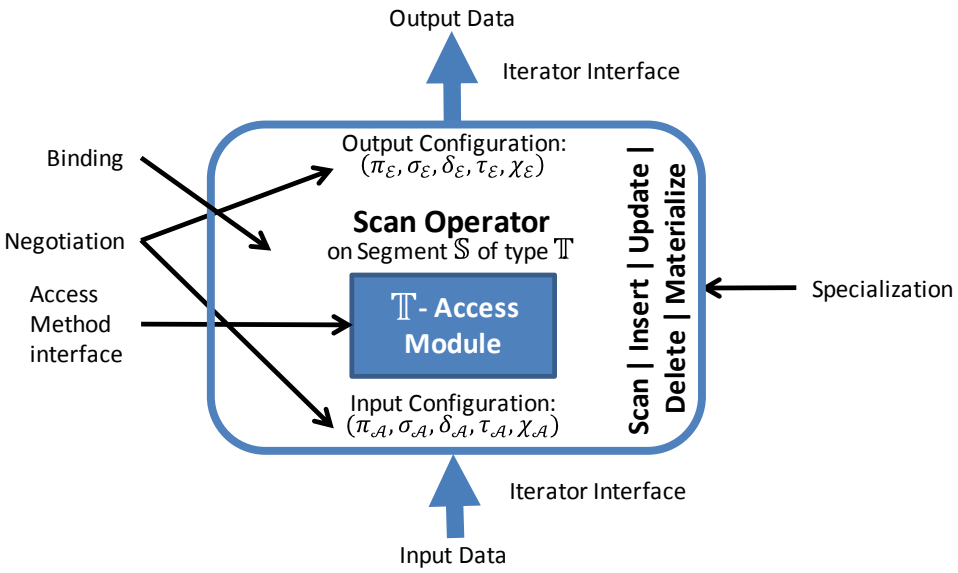


Figure.28 Encapsulation of access modules in generic scan operators. The scan operator provides a compact abstraction from the extensive functionality of an access module implementation. It is bound to a given segment $\mathbb{S}$ of type $\mathbb{T}$ and specialized to one specific task (*Scan*, *Insert*, *Update*, *Delete*, or *Materialize*). Additionally it is configured via negotiation for optimal interoperability within the query plan. The scan operator operates the access module's interface routines in accordance to these settings.

With this, the scan operator holds all information required for correct and efficient operations on the appointed segment, using the corresponding custom access module. Yet the scan operator remains an integral component of the host DBMS. Its configuration represents a detailed and complete description of the operation to be performed by the access module, without making any assumptions on any access module internals. This description makes such operations *plannable*, by recording the scan operator's configuration parameters in the DBMS's query plan representation. Configuration is an exact specification of a prospective operation, and the operator's cost function allows associating it with estimated costs.

Towards its outside, the scan operator provides only the reduced functionality required for negotiation during query planning and for query evaluation based on the Iterator model. This generality allows inserting the generic scan operator throughout a query plan, like any other relational operator.

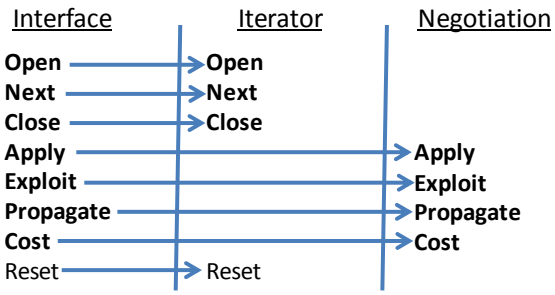


Figure.29 Generic Relational Operator Interface. The scan operator exports the universal interface of relational operators. The mandatory interface functionality (bold print) covers negotiation for query optimization and iteration during query evaluation. The optional **`Reset()`** routine serves for efficiency purposes, which remains to be discussed later.

As DBMS extensibility by integration of custom access method implementations is the primary focus of this work, this section will concentrate on employing access module functionality within scan operators. The following section *4.4 Relational Operator Interface* will discuss further generalization of the scan operator to a generic relational operator for encapsulating custom implementations of arbitrary relational algorithms.

4.3.1. Data Access Module Definition

The host DBMS is extended by integration of a library implementing an access module into the host system. With SQL being the preferred form of interaction with the database system, we also choose this language for this administrative task. To this end, the system's data definition language (DDL) has to be marginally extended to comprise the following statements:

```
CREATE TABLETYPE <ttype_name> FROM <path>

CREATE INDEXTYPE <itype_name> FROM <path>
```

Examples:

```
CREATE TABLETYPE FILE FROM filetab.dll

CREATE INDEXTYPE BITMAP FROM bitmap.dll
```

On close examination, these DDL statements extend the schema of one particular database, rather than globally extending the DBMS. Their purpose is to provide the location of dynamic libraries implementing custom access methods. They also assign unique names (i.e. `ttype_name` or `itype_name`, respectively) to be recorded in the database's data dictionary for referencing this module in subsequent DDL statements. Finally, the distinction between

TABLETYPE and INDEXTYPE describes whether the new module is used for primary or secondary access paths. Secondary indexes have to implement only a subset of the access method interface functionality. The reasons for these differences will be discussed in the course of data access strategies and data maintenance protocols. The information provided by the statements above is stowed away into the system catalog. No interface functions of the access modules are called during this process.

The inversion of this process is also required. Its purpose is the removal of an access method implementation from the database schema. This is accomplished by deleting the corresponding entries from the system catalog, after verifying that there are no active references, i.e. there must not exist any access paths based on the access module to be removed. The corresponding DDL extensions have the form:

```
DROP TABLETYPE <ttype_name>

DROP INDEXTYPE <itype_name>
```

All the prototype modules discussed later are implemented as built-in modules of the Transbase Access Manager prototype. They are statically linked to the database kernel and therefore they are a priori known to the system via predefined names. There is no requirement to publish their names via DDL as described above.

4.3.2. Access Path Creation

Creating tables and indexes based on custom access methods is integrated tightly into the host system's DDL. The corresponding statements differ only in an additional specification of the access method type (i.e. ttype_name or itype_name, respectively) from the corresponding standard SQL statements. As an optional extension, the specification of the module type can be enriched with the additional clause custom_spec. This clause will not be interpreted by the host DBMS, instead it will be passed directly to the access method, where it is parsed and processed. This clause may be used for providing additional configuration parameters to the access method.

```
CREATE <ttype_name> [(custom_spec)] TABLE <tname> (<tablespecification)

CREATE <itype_name> [(custom_spec)] INDEX
      <iname> ON <tname> (<fieldnamelist<)
```

Examples:

```
CREATE FILE ('/tmp/import/employee.txt') TABLE emp (empkey
  INTEGER, name CHAR(*), address CHAR(*), nationkey TINYINT)

CREATE BITMAP INDEX emp_nation_bmx ON emp (nationkey)
```

The effect of these statements is the creation of new data access structures. The example creates a new table of type `FILE`, which functions as a 'virtual' table in the database schema for permitting *direct* retrieval of data from a structured text file, which is residing in the file system outside the database. It uses the `custom_spec` clause for specifying the path to the text file. Additional conventions on the file format (CVS, XML, etc.) or file properties (e.g., delimiters and sort order) could also be declared in the `custom_spec`. This type of tables is particularly useful for directly accessing data exported from other applications, or for importing external data into the database. The data in the file may be queried and the query result can be subjected to arbitrary SQL transformations, permitting extensive SQL-based ETL (Extract, Transform, Load) capabilities. The possibility of creating secondary indexes on such tables, as demonstrated in the example, additionally promotes efficient access to external data.

The first preparatory step in creation of a new segment is conducted by the Access Manager framework by calling the storage layer's **`Create()`** routine for allocation of one single page in the system's internal storage facility. The address of this page (i.e. its page number) is henceforth used for identifying the new segment (**`SegID`**). We refer to this distinguished page of a segment as the segment's *description page* (**`DescPage`**), as it will store information describing essential properties of the segment. Every segment has a description page in the system's internal storage facility, regardless of whether the segment's data is actually stored inside the database's storage or not.

The second step is also conducted by the Access Manager framework. It registers the new access path in the system catalog. During this step, information from the original DDL statement describing the *logical data model* of the new segment is inserted into the system catalog. This information covers the relation's name, column names, column types, check constraints, referential constraints and other information required for handling relations on a logical level. In particular, this information serves for identification of access path candidates during query compilation. The only information on physical properties of the new segment to be recorded in the system catalog is the aforementioned **`SegID`**, which is required when accessing a segment for retrieving the description page. Finally, preparations for segment creation are completed and the Access Manager will call the access module's **`Create()`** routine.

```
Create(TaID, DescPage, SegID, CreateSpec, CustomSpec)
 → ScanContext
```

This method is provisioned with a transaction identifier (**TaID**), which is assigned by the host system and represents the transaction enclosing the **Create()** operation. The parameter binds this operation to the transaction denoted by **TaID**, e.g. two concurrent operations belonging to the same transaction may influence each other immediately, while one transaction is typically isolated from uncommitted effects caused by a another transaction. The **TaID** will also be passed on as parameter for potential calls issued by this operation against the underlying page-oriented storage system, where it is essential for providing effective isolation and locking to this transaction. The second parameter **DescPage** is a reference to the freshly allocated but still empty description page. The remaining parameters describe properties of the segment to be created. While the **TaID** describes an operation's dynamic context, the **SegID** is passed as parameter for defining the physical scope of an operation, i.e. the **SegID** binds this operation to one particular segment. Operations on separate segments are *physically* independent, but there may exist *logical* dependencies between segments, e.g. segments storing redundant data (indexes), referential constraints, triggers, etc. The remaining two parameters contain additional information from the original DDL statement. Relevant information from the standard DDL statement describing the *logical data model* of the new segment is provided in the standardized, structured representation of the **CreateSpec** parameter. This information comprises all logical properties such as column count, column types, column constraints, and primary key specification. Finally, the `custom_spec` clause from the original DDL statement is passed as uninterpreted text in the **CustomSpec** parameter.

The task of the **Create()** routine is to choose data from the information available in the parameters **SegID**, **CreateSpec**, and **CustomSpec**, which needs to be stored in the access method's description page. The information is chosen, depending solely on implementation and on particular requirements of the given access method. The goal is to make the segment self-contained, so that all information required for accessing and operating the segment (its *physical data model*) is readily available at the segment's single entry-point, the description page. All other pages of this segment are reachable (directly or indirectly) from this page. Also external data (e.g. in a `FILE` table), stored outside the database's storage facility, is accessed with information from a description page, which must be located inside the host system's internal storage. The description page's main task is to prevent that the system catalog has to be consulted during access path configuration, data retrieval, or data manipula-

tion, allowing fully autonomous operation of access modules. The system catalog is primarily consulted during query compilation, for mapping relation and attribute names to the correct segments. It is also used during query optimization for identifying alternative access path candidates for index selection. But with the retrieval of the **`SegID`**s of all access path candidates from the system catalog, access path resolution based on the logical data model concludes. From this point access path configuration, cost estimation, and query evaluation are conducted solely with information based on the physical data model from the description pages. As a consequence, it is likely that some information from the DDL statement is stored redundantly in the system catalog, as well as in the description page (Figure.30).

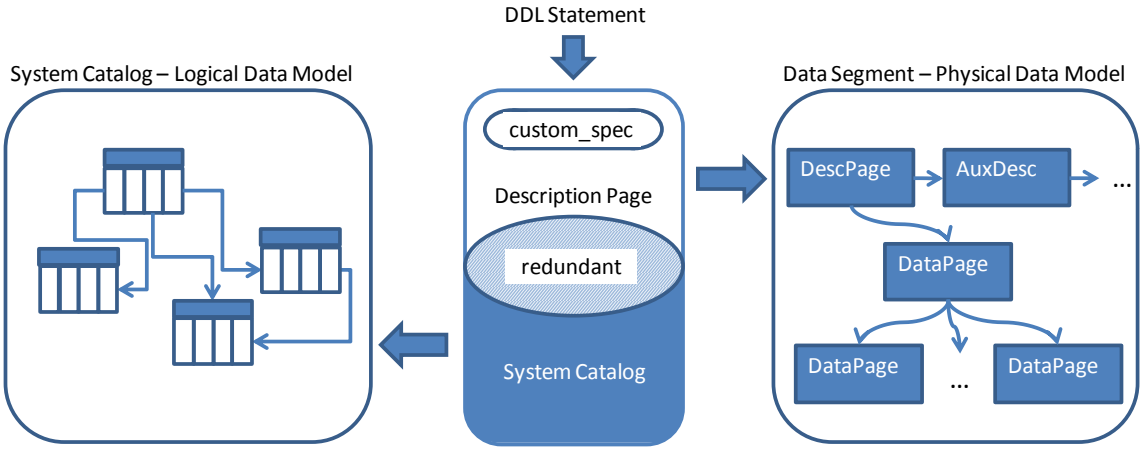


Figure.30 Information dissemination during access path creation. The original DDL statement contains all available information on the new access method. The information representing the *logical data model* of the new access path is automatically inserted into the system catalog. All relevant information from the standard DDL for describing the *physical data model* of the access method, plus the optional, non-standard `custom_spec` are passed to the data access module's **`Create()`** method. This information is stored in the segment's description page (auxiliary description pages may be allocated as necessary). The description page represents the segment's single entry-point and all other pages of this segment are linked to it.

The description page is initially empty, but in a writable state, i.e. it is fixed in the system's cache and it is attributed with a **`WRITE_LOCK`**, since it was recently created. The access module managing a segment may freely choose any suitable organization for the contents of the description page, as well as for any other page of this segment. The only exception is the short page header on every page, used exclusively by the page-oriented storage layer. If the space on the description page should not suffice, it is always possible to allocate additional pages and chain them to an existing page. The proprietary internal organization of pages makes the managing access module the only instance capable of interpreting the segment's pages. As a consequence, the segment's data can only be accessed via the access method's public interface. We emphasize that the host system possesses only a *logical* description of

the segment in the system catalog and the segment's **SegID** for locating the description page. Only the access method that created a segment can interpret and administer pages belonging to that segment.

When the **Create()** routine returns, the new access path is prepared for immediate operation, i.e. the routine establishes an open scan on the new segment. In this state, the scan may have created and accessed an arbitrary number of pages of its segment. All accessed pages, including the description page, are attributed with **WRITE_LOCK**s since they have been created recently, and some may even remain fixed, in preparation for subsequent operation on the segment, when the routine completes. Other accessed pages have been unfixed and released into the system cache. There they are available for subsequent use when accessing the segment. If they are not accessed a second time, they move through the cache's LRU stack until they are eventually swapped out to disk. While in cache, the pages are immediately accessible, but multi-version concurrency control ensures that they are only visible for concurrent scans on the same segment (**SegID**) and of the same transaction (**TaID**), since these pages contain uncommitted changes. At this point, other transactions can access neither pages nor catalog entries of the new segment. The new segment becomes globally visible only when the transaction that created the segment is committed.

The status of the new scan is subsumed in the **ScanContext** data structure, which is allocated, maintained, and eventually returned as result parameter when creating or opening a segment. This central structure is a collection of static and dynamic information to be preserved between individual calls to the routines of the access method interface. The **ScanContext** serves as the pivotal input parameter for all subsequent calls to other access method routines operating on this scan. Its contents are discussed in detail in the following section. The creation of a segment may now be followed by all sorts of operations on its resulting scan, e.g. if the new segment represents an index on a non-empty base relation, it will undergo immediate mass-insertion, for making the index contents consistent with the base relation. Such consolidation is automatically initiated by the host system, based on the system catalog's logical data model and it is executed via the Access Manager framework (cf. *4.3.8 Data Integrity* for more details).

The newly created segment becomes permanent when the surrounding transaction is committed. Then all allocated pages are persistently stored to disk and the transaction's locks are released. Finally the entries in the system catalog become permanent and visible to other

transactions. If the transaction is rolled back, all changes are undone and all pages allocated in its course are automatically released.

`Drop(ScanContext)` → ∅

The inverse activity of dropping a segment behaves exactly contrary. The **`Drop()`** method is called on an open scan. It uses a reference to the scan's **`ScanContext`** as its only input. Now it lies in the access methods responsibility to follow the chaining of all allocated pages belonging to that segment, in order to release them by calling the storage layer's **`DeletePage()`** routine. The underlying storage system does not maintain any data structures redundant to the access method's chaining of pages of one segment. Therefore, when the **`Drop()`** routine returns, the access method must guarantee that all pages it ever allocated have been released. Finally, the Access Manager framework will release the description page and remove all entries associated to the dropped segment from the system catalog. Dropping a segment is initialized by issuing the corresponding standard SQL DDL statement.

```
DROP TABLE <tname>

DROP INDEX <iname>
```

4.3.3. Tuple Identification and Indexing

An inevitable prerequisite for creating alternative access paths (indexes) on an existing segment (base relation), is the ability of the base relation to provide compact tuple identification. Functionally this identification is a bijective mapping between tuples in the base relation and their corresponding, redundant index tuples. This mapping is needed for resolving the corresponding base table tuple when a relation is accessed via a secondary access path (materialization). On the other hand, when updating the base table, the system must be able to find and update corresponding index tuples (index integrity). Finally, tuple identification is also used for associating corresponding tuples from different indexes of the same base relation, i.e. in secondary index intersection and union.

The relational paradigm inherently provides such identification via the base relation's primary key specification. But a primary key may be a rather extensive attribute combination, and the necessity of storing this key as tuple identifier with every index tuple inevitably leads to similarly extensive secondary index tuples. Hence, it is often desirable to have more compact tuple identifier. The practical concept of artificial tuple identifiers (TID, often also called RID for row identifier) is common to many relational DBMSs. There exist many proprietary

solutions to this problem, ranging from TIDs based on physical storage addresses of base tuples, to assigning fully artificial numeric IDs.

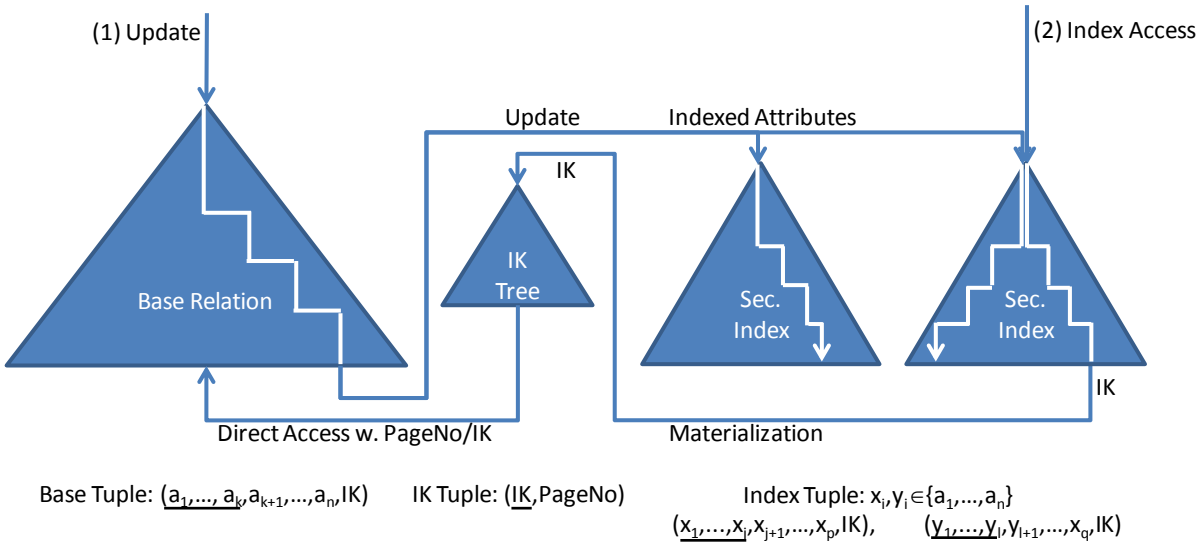


Figure.31 TIDs in Transbase. Use-cases of tuple mappings between base relation and secondary index. (1) In case of an update of the indexed attribute on the base relation, the corresponding tuple in the secondary index has to be found and updated. Technically, such updates on secondary indexes are always conducted as separate delete and insert operations. (2) When accessing a relation via a secondary access path, materialization of the base tuple must be possible. In Transbase this is provided via an indirection over the auxiliary IK-tree.

The Access Manager framework also includes an interface for generating TIDs as a service for custom access modules. In Transbase (cf. Figure.31) the TIDs are called IKs (internal keys) and are based on fully artificial numeric values. The IKs for one segment are managed in an auxiliary B-tree, called the IK-tree, which is residing in the same segment as the primary relation. The structure is created automatically at segment creation time and it is capable of generating unique TIDs on request. Moreover, one single IK-tree suffices for handling an arbitrary number of secondary indexes on a given segment. Therefore, it stores tuples of the form (IK, PageNumber), providing management and lookup capabilities for *unique* IKs. In addition, the IK-tree associates each IK with a page number. This page number identifies a page belonging to the base relation segment and storing the base tuple associated with the IK. Finally, corresponding tuples in the base relation and secondary indexes contain identical IK values. This allows direct lookup of the tuple in a secondary indexes corresponding to a given base relation tuple, using index attribute values plus IK value from the base tuple. Conversely, a given index tuple is materialized from the base relation by using its IK value for retrieving the base tuple's page number from the IK-tree. This allows direct access to the page contain-

ing the base tuple. Within the page, the actual tuple is located using a sequential search for the given IK.

The Access Manager TID facility in Transbase is used as follows. If a new tuple is to be inserted into the base relation, the responsible access method implementation determines an existing page in the segment, or allocates a new one, for storing the new tuple. The current transaction's **TaID**, the **SegID** of the base relation and the target page number (**PNO**) are used as parameter for calling the **GetTID()** routine. This routine generates a new unique TID and stores it together with the destination page number in the IK-tree denoted by **SegID**. This modification happens within the context of transaction **TaID**. The new TID is returned by **GetTID()**, and has to be stored in the base relation, together with the new data tuple.

GetTID(TaID, SegID, PNO) → TID

Besides creating TIDs, the interface also allows updates in the IK-tree structure via its **SetTID()** routine. This functionality is required, if existing tuples are moved to another page, e.g. if pages in the primary access path are split or merged. The required parameters for updating the IK-tree are the original **TID** and the new page number **PNO**. The **TID** allows locating the correct entry in the IK-tree, and then the entry is updated using the new page number **PNO**.

SetTID(TaID, SegID, TID, PNO) → ∅

Finally, if a tuple is deleted, its IK value is removed from the IK-tree using **FreeTID()**. The provision of **SegID** and **TID** are sufficient for locating and deleting the correct entry.

FreeTID(TaID, SegID, TID) → ∅

TIDs are provided as an optional service by the host system. Each access module is free to use this facility or to provide a custom TID implementation. As an example for an alternative TID mechanism, an access module might use the access method interface to create and operate an auxiliary B-tree structure resembling the IK-tree. Both approaches are functionally equivalent, but the built-in TID mechanism offers a more convenient interface for this purpose.

How and when TIDs and primary keys are actually used for secondary access paths is discussed in detail in sections *4.3.6 Elementary Navigational Access* and *4.3.7 Data Manipulation* of this chapter.

4.3.4. Opening an Access Path

Before data can be accessed, an access path has to be prepared for operations via its **`Open()`** routine. The process is triggered by the SQL compiler resolving access paths from table names referenced in a given SQL statement. Therefore, the host system's SQL compiler and plan optimizer consult the system catalog on the logical data models of available access paths. These catalog queries serve also for identifying alternative access path candidates. For every primary or secondary access path candidate, the description page number identifying the access path's segment (**`SegID`**) is retrieved from the system catalog. Eventually the Access Manager framework retrieves the corresponding description pages (**`DescPage`**), which is used as parameter for calling the **`Open()`** method of each access path candidate. The **`READ_LOCK`** on the **`DescPage`** and all subsequent operations on this scan are attributed to the current transaction denoted by **`TaID`**.

`Open(TaID, DescPage, mode,` $\left(\mathcal{C}_{\mathcal{A}_i}\right)_{i=1,\ldots,n}$**`,`** $\mathcal{C}_{\mathcal{E}}$**`)`** → **`ScanContext`**

`mode ::= { Scan | Insert | Update | Delete | Materialize }`

The **`mode`** parameter specializes the scan to a particular purpose, namely to retrieval, insertion, modification, deletion, or materialization, as appointed by the defining SQL query. This parameter is provided by the generic relational operator encapsulating the access module. The $\mathcal{C}_{\mathcal{A}_i}$ parameters represent the scan's *applicability* requirements for its *i*-th input stream. Scan operations are nullary for data retrieval and unary for materialization or data manipulation. Still we provide an interface definition for general *n*-ary operators for reasons that will become apparent shortly. The parameter $\mathcal{C}_{\mathcal{E}}$ describes *exploitability* settings of the single output stream. When opening a scan for the first time, i.e. as an access path candidate during query compilation, $\mathcal{C}_{\mathcal{A}_i}$ and $\mathcal{C}_{\mathcal{E}}$ remain vacant, as both specifications are undetermined in this early phase of query planning. They will be established later during the query optimization process, as discussed in the following section.

Open scans have a state of operation at all time, and this state manifests itself in the **`ScanContext`**. This central data structure, which is entirely defined by and fully private to the access module, covers all information on the scan's current state of operation. The **`ScanContext`** is generated each time a segment is created or opened. At that time, information from the persistently stored description page is processed and transferred into this main memory structure, where it is immediately accessible as long as the scan remains open. This step allows unfixing the description page after the scan has been opened, releasing the occupied

cache frame into the host system cache's LRU stack. A **READ_LOCK** will be held on the unfixed description page, guaranteeing that the information in the **ScanContext** remains valid. In addition to this static information from the segment's description page, the scan context will also include information on the current transactional context (**TaID**), which is available as input parameter. This **TaID** defines the scan's affiliation to its enclosing transaction, thereby determining its behavior relative to concurrent scans belonging to the same or other transactions. Consequently, a scan cannot outlast its enclosing transaction, nor is it possible to reassign a scan to another transaction. Similarly, the configuration of the scan is provided as $\mathcal{C}_{\mathcal{A}_i}$ and $\mathcal{C}_{\mathcal{E}}$ parameters of the **Open()** method. As already indicated, these parameters are optional, allowing to open a scan for negotiation without preliminary configuration. During subsequent operations, the **ScanContext** will gather dynamic information, such as current scan position and information on data pages from its segment, that are currently held fixed in the system cache for immediate access. The **ScanContext** serves as the pivotal input parameter for all access method interface routines implementing operations on an open scan. It represents the scan's complete collection on status information and it is also the scan's only opportunity to 'remember' information across individual operations. It will be perpetually read and modified by every operation, reflecting relocations of the scan position and accounting information on currently fixed data pages.

After all suitable access path candidates have been opened, they are ready for general optimization, in particular for index selection. During optimization, all access path candidates are undergoing configuration via negotiation and cost assessment. The optimization strategy is appointed by the host system's query optimizer, while the individual access method implementation contributes only the necessary instrumentation by implementing the negotiation interface. Negotiation will directly influence the **ScanContext** of each scan, i.e. the appointed configuration is recorded within this structure. When optimization finds an access path candidate inappropriate for the current query, it is closed immediately. The **Close()** operation releases all resources that were occupied by the scan. In particular, all fixed pages must be released by the access module. Page locks that have been acquired thus far will be held by the host system's lock manager, as required by the enclosing transaction's isolation level.

Close(ScanContext) → ∅

When query optimization completes, only the chosen access paths remain open. In this state, the scans are fully configured and ready for imminent query evaluation. But between query compilation and query evaluation may pass an arbitrary long period of time. This becomes apparent when considering stored queries. Therefore, the final configurations of the chosen access paths are now recorded in the host system's internal, storable representation for optimized query evaluation plans. The scans are eventually closed, releasing all allocated resources, in order to be reopened only immediately before actual query evaluation commences. When scans are reopened, the **`Open()`** routine is provided with valid $\mathcal{C}_{\mathcal{A}_i}$ and $\mathcal{C}_{\mathcal{E}}$ parameters from the prior negotiation process, as preserved in the query plan. In reopening the scan with full specifications, the original configurations are reinstalled, avoiding the necessity for repeated negotiation and preparing the access paths for immediate operation. When the evaluation of a query terminates, all scans of the query are ultimately closed.

The strict protocol of closing and reopening scans offers full control over the scan status manifested in the **`ScanContext`**, since the initial status of a configured scan is well-defined. But this protocol has one important flaw: perpetually reopening scans is an unnecessary and expensive operation, discarding and rebuilding similar **`ScanContext`**s over and over again. In particular, closing scans after query optimization and reopening them for query execution clearly demonstrates this adverse behavior, but the same applies also to a series of similar queries on one segment within the same transaction. The prospect of reassigning an open scan directly to another operation (task) on the same segment will allow preservation of its scan context to a large extent. Repetitive processing of the description page can be omitted and static information in the scan context will prevail over multiple scan incarnations. Dynamic information concerning the previous scan status can be reviewed for possible reuse. The latter applies in particular to fixed pages, e.g. a B-tree scan, keeping a *fixed* path from its root page through internal pages to the current scan position on a leaf page, may reuse a portion (at least the root page) of fixed internal pages when repositioned for a new task.

For allowing this form of operation, a scan must be able to recognize that its purpose has changed between two operations. But it is not possible to reliably detect the boundaries of independent operations, solely from the normal operation protocol. Database operations often consist of a series of individual but implicitly stateful operations, e.g. interweaved navigation and manipulation on one segment. As an example, the SQL semantics of *update-positioned* allows a database user to interact directly with a relational scan. Such semantics are not visible anymore at the scan's low-level interface. From the viewpoint of an access module,

such operations are just a series of consecutive calls to various interface routines. In particular, it is not clear at which point a scan is reassigned to another task. Only the **`Close()`** operation gives conclusive evidence that a unit of work was completed and the scan context has become obsolete.

To overcome the necessity of closing a scan, the Access Manager offers the optional interface routine **`Suspend()`**. This routine, if implemented by an access method, is used by the Access Manager to inform a scan that a set of operations belonging to one task has been completed.

`Suspend(ScanContext)` → ∅

The Access Manager calls the **`Suspend()`** routine for effectively preserving unused open scans in an auxiliary structure called *scan-buffer*. This buffer contains a limited number of suspended scans, organized in least-recently-used fashion. If the maximum number of such scans is exceeded, then the least recently used scan is removed from the buffer by physically closing it. The scan buffer is maintained by the Access Manager framework in a way that is completely transparent to the access method implementation. In this buffer, a scan may exist across multiple logical incarnations, as readily available and reusable asset. There it may serve for various different purposes during its life-span, but it always remains limited to the one segment on which it was originally opened, as it is bound via the description page it processed when opened. In addition, a scan must not outlast the page locks it relies on. When the locks are released, the description page (or other pages) could change without being noticed in the scan context. Therefore, a scan's life-span also depends on the isolation level of its enclosing transaction. If the transaction's isolation level guarantees serializability, then locks are held until transaction boundaries. Consequentially, an open scan may also exist for the same duration. In case of a lower isolation level, a scan's existence is correspondingly shorter, e.g. only for the duration of one SQL query. The correct behavior is controlled and guaranteed by the Access Manager framework, which will either logically suspend or physically close scans as appropriate. When an access method experiences a call to its **`Suspend()`** routine, it has to initiate operations that are essential for concluding a unit of work, for example freeing resources that will not be reused in a consecutive operation. Otherwise, the scan remains open and fully operational, ready for immediate reactivation. This reactivation is initiated by a second optional Access Manager interface routine:

`Reset(ScanContext, mode,` $\left(\mathcal{C}_{\mathcal{A}_i}\right)_{i=1,\ldots,n}$**`,`** $\mathcal{C}_{\mathcal{E}}$**`)`** → ∅

```
mode ::= { Scan | Insert | Update | Delete | Materialize }
```

Similar to the **Open()** routine, the **Reset()** routine provides complete configuration settings for the upcoming unit of work, but it lacks the **DescPage** parameter, as the description page is already transcribed into the still active **ScanContext**, which is preserved from the preceding operation on this segment. The next call to any routine of this scan will therefore have the same effect, as if called on a freshly opened scan, i.e. a sequence of **Suspend()** and **Reset()** is the lightweight equivalent to a sequence of **Close()** and **Open()**, reinitiating a scan without physically closing it. The resulting scan still belongs the same transaction (**TaID**) and operates on the same segment (**SegID**) on which it was opened originally, but it may exhibit a different specialization (**mode**) and an alternative configuration ($\mathcal{C}_{\mathcal{A}_i}$, $\mathcal{C}_{\mathcal{E}}$), as the scan is about to serve for a different purpose. Conceptually, the **Suspend()** routine is the terminal function call of a cohesive unit of work, while the **Reset()** call starts the consecutive operation.

4.3.5. Negotiation and Optimization

The routines belonging to the operational area of negotiation are used in the query optimization phase. They provide information on capabilities and requirements of custom access methods and other custom relational operators. Based on this information, the DBMS query optimizer is able to integrate such custom operators into query evaluation plans (QEPs) and eventually it chooses the most promising QEP from several alternatives. In cost-based optimization, this decision is supported by cost estimation, which therefore necessitates cost functions for custom relational operators. For providing the required functionality for negotiation and query optimization, the access method interface specifies four routines, namely **Apply()**, **Exploit()**, **Propagate()**, and **Cost()**.

The **Apply()** routine enables substitution of an *n*–ary ERA sub-expression in an algebraic query plan with an algorithmic unit implementing that expression, by establishing all necessary prerequisites of that algorithm. Consequentially, the algorithmic replacement will possess *n* input streams, and for each one it may demand particular input requirements within the scope of *equivalence configuration*. Relational scans exhibit a few special characteristics, distinguishing them from general relational operators. Retrieval scans are always leaf operators in a query plan, possessing no input stream. Such *nullary* retrieval scans are applicable unconditionally, making an **Apply()** routine apparently obsolete for this type of operator. On the other hand, relational scans have input streams, if their purpose is modification of stored

data or when they are used for materialization of a base relation after an index access. The input stream of modification scans represents data to be inserted, or identifies tuples to be deleted or modified. In the latter case, it also describes the modification to be performed. Materialization scans operate on one input stream, originating from a secondary index and providing identification of tuples to be retrieved from the associated base relation. Both manipulation and materialization scans operate on persistently stored relations, each using one single input stream, i.e. they both represent *unary* relational operations. Both nullary and unary scan operators can be adapted through *propagation* and *exploitation* of correlated predicates, as used for direct lookup on the inner relation of a nested-loop join. The unbound variables of correlated predicates represent additional input streams that are also subject to negotiation. For basic customization of a scan operator, allowing optimized interaction with its input streams, the access method interface specifies the following routine, specialized for applicability of n-ary scan operators:

`Apply(ScanContext,` $\boldsymbol{\pi^{in}}$**`, opt)`** → $\left(\mathcal{C}_{\mathcal{A}_i}\right)_{i=1,\ldots,n}$

`opt ::= 0,1,2,3,...`

This routine allows the scan to announce its applicability requirements $\mathcal{C}_{\mathcal{A}_i}$ (cf. Definition.11: Apply function $\boldsymbol{\mathcal{A}}$ on page 38), consisting of an array of configuration ters $(\pi_{\mathcal{A}}, \sigma_{\mathcal{A}}, \delta_{\mathcal{A}}, \tau_{\mathcal{A}}, \chi_{\mathcal{A}})$ for each input stream. Before the **`Apply()`** routine is called, the scan to be configured must be opened, i.e. a **`ScanContext`** is available as parameter, providing substantial information on the physical data model of the scan. The projection $\pi^{in} = \left(\pi_i^{in}\right)_{i=1,\ldots,n}$ describes the permutation of actual attribute positions in the *i*-th data input stream, relative to attribute references in the algorithmic unit's replacements pattern. The **`opt`** parameter selects from an enumeration of optional input requirements. Setting **`opt=0`** establishes minimal applicability requirements for correct functioning of the applied operator. In particular, minimal input requirements should contain only standard representation tives χ_{std}, otherwise the operator is not generally applicable. In addition, the scan operator may allow an arbitrary number of optional input requirement specifications. The optimizer iterates over available **`opt`** settings, starting from zero. If **`Apply()`** is called with an **`opt`** level exceeding the maximum number of available optional input requirements, then the routine returns empty $\mathcal{C}_{\mathcal{A}_i}$ configurations. Whenever returning valid $\mathcal{C}_{\mathcal{A}_i}$ settings, the **`Apply()`** routine also records $\mathcal{C}_{\mathcal{A}_i}$ within its **`ScanContext`** structure, thereby configuring the scan to this new setting. After each iteration, the optimizer may analyze the costs for satisfying the

applicability requirements of the current **opt** setting, potentially using exploitation and propagation on the algorithm's input stream, for actively minimizing applicability costs. The process searches for input configurations offering the best trade-off between good interoperability and low applicability expenses. Eventually the optimizer decides on the input configuration to be used, and a final call to **Apply()** with the chosen **opt** setting makes sure that the corresponding $\mathcal{C}_{\mathcal{A}_i}$ is recorded in the **ScanContext**.

Minimum applicability specifications of scan operators typically comprise only convenient projection of input data. Advanced configurations for more efficient manipulation and materialization often enforce an input sort order that matches the primary linearization of the base relation. As an example, materialization using successive lookup of primary keys in the base relation will perform better, if the search keys are delivered in such order, that every page of the base relations is visited exactly once, even if it contains multiple hits. A more detailed example will be provided in *4.3.7 Data Manipulation*. Finally, non-standard representation also allows tight integration with preceding operations.

The **Exploit()** routine consolidates QEPs that were extended using **Apply()**. It aims for establishing applicability requirements of an algorithm efficiently, by integrating necessary transformations into the algorithm's immediate predecessor. We already demonstrated that this method is particularly effective for constructive query planning based on the *principle of optimality*, because exploitation operates strictly locally, having no implications on the already established applicability requirements of the preceding algorithmic unit. This is also true for exploitation of correlated predicates. An *n*-ary operator, accepting a correlated predicate via exploitation, de facto obtains its *n+1st* input stream in form of a parameter stream feeding values into the predicate's unbound variables. This means that even nullary retrieval scans may factually operate on input streams. The already mentioned locality of exploitation ensures that this additional input stream does not introduce additional applicability requirements, neither on the introduced parameter stream, nor for any other input stream.

Modification scans are always located at the root of query plans, and therefore they have neither parent operators, nor do they produce relational data. Their result is an integer number representing the total number of tuples affected by the modification. Hence, exploitation applies only to scans used for data retrieval and materialization. For these scan operators, *exploitation* allows conducting additional transformations on the fly, while scanning the input set. The **Exploit()** routine (cf. Definition.13: Exploit function $\mathcal{E}$ on page 45) takes the

applicability specification $\mathcal{C}_{\mathcal{A}}$ of the parent operator as input and determines the accepted configuration parameters $(\pi_{\mathcal{E}}, \sigma_{\mathcal{E}}, \delta_{\mathcal{E}}, \tau_{\mathcal{E}}, \chi_{\mathcal{E}})$ of the exploited operator. These accepted exploitable parameters $\mathcal{C}_{\mathcal{E}}$ are immediately adopted into the **`ScanContext`**, making them part of the scan's current configuration. As result, the routine returns $\mathcal{C}_{\mathcal{R}}$, the vector of rejected parameters $(\pi_{\mathcal{R}}, \sigma_{\mathcal{R}}, \delta_{\mathcal{R}}, \tau_{\mathcal{R}}, \chi_{\mathcal{R}})$. Finally, the optimizer will compensate for rejected operations by inserting corresponding standard implementations for $\mathcal{C}_{\mathcal{R}}$, thereby generating a coherent and executable query plan.

`Exploit(ScanContext,` $\mathcal{C}_{\mathcal{A}}$**)** → $\mathcal{C}_{\mathcal{R}}$

The routine **`Propagate()`** is used by query optimization based on algebraic equivalence transformation of a query plan. It pushes configuration parameters downwards through algorithmic units that are permeable for these parameters. For example, establishing a specific sort order either before or after an order-preserving operation is functionally equivalent. Similarly to exploitation, propagation will also extend an *n*-ary operator to *n+1* factual input streams, by accepting a correlated predicate. This happens if a correlated predicate is absorbed by the algorithmic unit in the course of propagation, rather than being properly propagated. In contrast to exploitation, propagation may influence an algorithmic unit's applicability requirements. The query plan optimizer will automatically explore the impact of propagation on applicability requirements, by employing the **`Apply()`** routine as stipulated in the negotiation protocol. Hence, in the presence of correlated predicates, the **`Apply()`** routine achieves true relevance, even for retrieval scans that are originally nullary operators without applicability requirements.

Absorption of a correlated predicate through propagation may have side-effects on a unit's applicability requirements and, in particular, it may introduce applicability requirements for the freshly added input stream. As an example, assume an inner scan participating in a nested-loop join is absorbing the join predicate in form of a correlated predicate (cf. also Figure.10 (b) and (c) on page 51). Then it may request that the data stream of the join's outer loop, which is feeding the correlated predicate, is adequately sorted for improved lookup performance, thereby effectuating a *skip-merge* join algorithm, i.e. a merge join capable of skipping mismatching data on the inner stream. It should be noted that such applicability requirements for correlated predicates represent non-local input directives. In case of a nested-loop join, they introduce additional input directives for the join algorithm's outer loop, although they are established by the inner loop's relational scan. This form of propagation may therefore pro-

duce contradictory input directives. The query plan optimizer is in charge of identifying such conflicts and resolving them, if possible. However, it is recommended that non-local input directives should be used with care, and they should appear only as *optional* input requirements of an algorithmic unit, otherwise conflicting input directives might inhibit absorption of correlated predicates.

Naturally, permeability and therefore also propagation applies only to internal nodes of a query plan, making materialization the primary class of scan operators capable of propagation. Retrieval scans support propagation only when handling correlated predicates. Finally, modification scans at the root of a query plan do not support propagation at all. Similarly to **`Exploit()`**, a call to **`Propagate()`** (cf. Definition.15: Propagate function $\mathcal{P}$ on page 49) takes the applicability specification $\mathcal{C}_{\mathcal{A}}$ of its parent operator as input and returns only the rejected parameters $(\pi_{\mathcal{R}}, \sigma_{\mathcal{R}}, \delta_{\mathcal{R}}, \tau_{\mathcal{R}}, \chi_{\mathcal{R}})$ in its result $\mathcal{C}_{\mathcal{R}}$. Propagation is described by the equation $op_{\mathcal{A}} = op_{\mathcal{R}} \circ op_{\mathcal{E}} \circ op_{\mathcal{P}}$ for every single configuration parameter, hence it implicitly includes exploitation. Similarly to exploitation, propagation directly configures the **`ScanContext`** to locally exploitable parameters $(\pi_{\mathcal{E}}, \sigma_{\mathcal{E}}, \delta_{\mathcal{E}}, \tau_{\mathcal{E}}, \chi_{\mathcal{E}})$. The propagated configuration never becomes explicitly visible. Instead, successful propagation influences the applicability requirements of the operator by directly modifying the scan's configuration in its **`ScanContext`**. Modified applicability requirements necessitate repeated consolidation, which lies within the responsibility of host system's optimizer and is also accomplished via the negotiation interface, i.e. the modified $\mathcal{C}_{\mathcal{A}}$ is retrieved using the scan's **`Apply()`** routine and subsequently satisfied by employing downward propagation or exploitation.

`Propagate(ScanContext,` $\mathcal{C}_{\mathcal{A}}$**)** → $\mathcal{C}_{\mathcal{R}}$

Finally, after the scan is configured, a call to its **`Cost()`** routine retrieves the estimated costs for evaluation, making the effectiveness of alternative QEPs comparable. The general costs of an algorithmic units amount to the sum of local costs plus expenses for generating the necessary input for the algorithm. The local costs assessed for a scan operator display the required expenses for completing a complex unit of work, i.e. the traversal of its persistently stored relation under the given configuration parameters and controlled by the scan operator's input streams. Functionally such a traversal corresponds to a succession of many individual access method interface calls, involving navigation and repositioning within the stored data set, but also retrieval and modification of data. This complex unit of work is described in detail by the scan operator's configuration, consisting of its initial operation **`mode`**, applicability directives

$\mathcal{C}_{\mathcal{A}}$, and exploitation $\mathcal{C}_{\mathcal{E}}$, where the latter two are established during negotiation and are available in the **`ScanContext`**.

```
Cost(ScanContext, InCosts, Stats) → OutCosts
```

For estimating cost in accordance to the recursive cost model presented earlier, the cost function receives the precalculated costs of its input streams in the **`InCosts`** parameter, separated into streaming and blocking cost accounts. In addition, cost estimation of an operation is substantiated with statistical information, namely cardinality estimations, on the operator's input and output streams. This information is generated autonomously by the host system, without any knowledge on the inner workings of custom algorithmic units. Derivation of statistics is based on the observations that boundaries of algorithmic building blocks in a query plan always coincide with the boundaries of the algorithms' algebraic equivalents. This allows maintaining valid statistical information throughout a purely algebraic representation of a query plan, containing arbitrary custom implementations of relational operators. Statistical information on an algorithm's input streams and on its single output stream is provided in the parameter **`Stats`**. The availability of statistics on both input and output is intended for facilitating accurate cost assessment, in particular for algorithmic units implementing extensive algebraic expressions, where interpolation of statistical information might become necessary for accurate cost estimation. Finally, **`Stats`** also includes input statistics of parameter streams feeding correlated predicates.

In contrast to other relational operators, the **`Stats`** parameter of a scan operator is additionally supplemented with the cardinality of the persistent relation on which the scan is operating, as this also represents an actual input stream. For cost estimation, comparison of this input cardinality with the cardinality of the scan's output stream allows inferring the selectivity of the scan's accepted predicates. For being able to provide such information, the host system must maintain general statistics on persistent relations, comprising cardinality and selectivity estimations. These statistics are automatically collected and maintained by the host system, which is able to access and analyze any persistently stored relation. Such access is transparently possible, regardless of the actual access method implementation, via the corresponding access module's access method interface. This allows convenient generation of arbitrary data-centered statistics, such as data distribution, by employing the host system's query processor. The statistics are then stored in the DBMS data dictionary, where they are available for subsequent cost estimation.

If additional information on physical properties of the access structures is required for cost estimation, e.g. the degree of fragmentation or height of a search tree, then such information has to be maintained separately by the access structure itself. This can be accomplished by adequate book-keeping in the description page, whenever the access path is restructured during data manipulation. When the access path is opened, this information is transcribed into the volatile **`ScanContext`** structure, where it is available for cost estimation. Cost estimation returns the data structure **`OutCosts`**, containing a collection of qualitative cost events, strictly divided into blocking and streaming accounts. **`OutCosts`** may serve directly as **`InCosts`** parameter in recursive cost estimation of a parent algorithmic unit. The process of cost estimation is concluded, when the host system eventually maps qualitative cost events to quantitative costs in accordance to the host system's cost model, allowing for flexible and dynamic cost assessment.

4.3.6. Elementary Navigational Access

All data retrieval facilities of the relational scan operator are concentrated in the **`Next()`** routine. This routine is adopted directly from the Iterator model and its operation is tightly interconnected with the scan operator's exploitable configuration parameters $\mathcal{C}_\mathcal{E} = (\pi_\mathcal{E}, \sigma_\mathcal{E}, \delta_\mathcal{E}, \tau_\mathcal{E}, \chi_\mathcal{E})$. If the scan operates on input streams, then the **`Next()`** routine relies also on the input streams' compliance with established applicability requirements $\mathcal{C}_\mathcal{A} = (\pi_\mathcal{A}, \sigma_\mathcal{A}, \delta_\mathcal{A}, \tau_\mathcal{A}, \chi_\mathcal{A})$ for each input stream. These configurations, which are obtained by negotiation during query plan optimization, integrate the scan operator tightly into the QEP, allowing optimized interoperability with adjacent relational operations. The combination of both configurations represents a detailed specification of a planned, iterative traversal through a persistently stored relation. Hence, both configurations are integral parts of the **`ScanContext`**, which is the single parameter of the **`Next()`** routine.

```
Next(ScanContext) → (OutTup, ScanStatus)

  ScanStatus ::= { OnTuple|NotOnTuple|EndOfData }
```

The **`Next()`** call offers rich navigational capabilities in form of relative and absolute scan positioning within the persistent data set. For relative positioning, we define two distinguished scan positions, namely *begin-of-data* (before the first tuple) and *end-of-data* (after the last tuple). Initially, a scan on some relation *T* is by definition positioned on begin-of-data. Relative positioning is effectuated by configuring the scan's $\mathcal{C}_\mathcal{E}$ setting to the trivial selection $\sigma_\mathcal{E} = id$. Every call to **`Next()`** will move the scan relatively to its previous position, i.e. the

scan moves forwards from its current position to the *next* tuple *t* in *T*, with respect to some *chosen linearization* of *T*. A linearization is chosen by supplying a non-trivial configuration parameter $\tau_{\mathcal{E}}$, requesting traversal of *T* in the specified *lexicographical* sort order. In most cases however, traversal will follow the primary linearization of the relation, which is implicitly chosen by configuring the scan to $\tau_{\mathcal{E}} = id$. This allows choosing the primary linearization, even if it is *not* a lexicographical sort order. Hence, relative positioning using **`Next()`** always moves the scan *onto* the next tuple *t*, with respect to the chosen linearization, and the routine will return **`OnTuple`** as **`ScanStatus`**. In addition, *t* is subjected to a projection $\pi_{\mathcal{E}}$ and output in **`OutTup`**, as an immediate result of the **`Next()`** function call. If the current tuple has no successor in the chosen linearization, then **`Next()`** will eventually position the scan to end-of-data and correspondingly return **`EndOfData`** as **`ScanStatus`**. With this, relative positioning allows iterative traversal of *T*, visiting every tuple exactly once.

In addition, navigation using the **`Next()`** routine may be complemented with an optional **`Previous()`** operation. This routine offers the same functionality as the **`Next()`** routine, but it follows a linearization inverse to the chosen one. If the **`Previous()`** routine is not implemented, the host system will try to compensate by using **`Next()`** on a scan configured to the inverted linearization of $\tau_{\mathcal{E}}$. This is only possible, if the chosen linearization corresponds to a lexicographical sort order ($\tau_{\mathcal{E}} \neq id$), otherwise it cannot be formulated as configuration parameter. For non-lexicographical orders, or if an inverted sort order is rejected during negotiation, then the relation will be scanned in forward direction and the result is subsequently sorted using a conventional sort operation $\tau_{\mathcal{R}}$. Similar to forward iteration, the optional **`Previous()`** function is expressed as:

```
Previous(ScanContext) → (OutTup, ScanStatus)

  ScanStatus ::= { OnTuple|NotOnTuple|EndOfData }
```

In contrast to relative positioning, absolute positioning moves a scan onto a selected coordinate of the multidimensional space spanned by the attributes of a relation. The position is specified as a restriction $\sigma_{\mathcal{E}}$, where $\sigma_{\mathcal{E}}$ defines an exact point. For such pinpoint positioning, $\sigma_{\mathcal{E}}$ may provide a full specification of all relation attributes, such that $\sigma_{\mathcal{E}}$ appoints constant values $(c_1, \ldots, c_n)$, determining coordinates in all *n* dimensions of *T*. The minimum specification of an exact point in *T* is represented by restricting any subset of *T*'s attributes constituting a unique *key* in the relational sense. Alternatively, $\sigma_{\mathcal{E}}$ may also select the TID of a searched tuple as surrogate for a unique key. In any case, if a tuple *t* satisfying $\sigma_{\mathcal{E}}$ exists in *T*, then the

scan is positioned onto *t* and the existence of *t* is acknowledged by returning **`OnTuple`** as **`ScanStatus`**. In addition, *t* is subjected to $\pi_{\mathcal{E}}$ and output in **`OutTup`**. Direct access may also position the scan into unoccupied space *between* existing tuples, if no data in the table satisfies the predicate of $\sigma_{\mathcal{E}}$. Hence, no output tuple is available, but the scan is positioned to the spot where the searched data would be located, with respect to the primary linearization of *T*. In this case, the routine returns the status **`NotOnTuple`** and **`OutTup`** remains empty. In contrast to relative positioning, absolute positioning is strictly stateless, i.e. its result is always the same, regardless of the previous scan position.

Finally, scans can be configured to a complex selection $\sigma_{\mathcal{E}}$ defining a set of intervals, or a set of multidimensional query boxes, which are additionally composed with logical interrelations (AND/ OR/ NOT), as expressed in our conception of multi-attribute selection predicates in disjunctive normal form $\bigwedge_i \bigvee_j (\neg)\, [[c_{ij_{low}}, c_{ij_{high}}]]$. In this configuration, the scan will autonomously use absolute positioning for navigating directly onto a query box and subsequently traverse it by using relative positioning, employing the skip-scan technique described earlier. As a result, **`Next()`** and **`Previous()`** will iteratively return all tuples qualifying for the given selection predicate, before finally reaching end-of-data. In conjunction with a linearization $\tau_{\mathcal{E}}$ and the other configuration parameters, this technique allows highly sophisticated traversal of persistently stored data sets through the remarkably simple Iterator interface. We refer to this form of scan operation using a predefined and constant scan configuration $\mathcal{C}_{\mathcal{E}}$ as *conventional navigation*.

Now we will investigate possible interaction of $\mathcal{C}_{\mathcal{E}}$ with an input stream on the example of a materialization scan. The input of a materialization scan provides tuple identification, retrieved from secondary indexes for direct lookup of the associated tuple in the base relation. Hence, navigation is a sequence of point accesses, using absolute positioning, where the predicate of $\sigma_{\mathcal{E}}$ depends on data from the scan operator's input stream. Technically, this is conducted by substituting variables in the predicate with corresponding data delivered by the input stream. This preparatory step is accomplished by the scan operator entity (cf. Figure.28 on page 118), encapsulating an access module implementation. The **`Next()`** routine does not provide for manipulation of the scan's configuration, but the **`Open()`** routine allows direct supplementation of the $\mathcal{C}_{\mathcal{E}}$ parameter. The complete materialization procedure consists therefore of multiple sequences, each starting by substituting variables in $\mathcal{C}_{\mathcal{E}}$ with input data, in preparation for calling **`Open()`**, followed by **`Next()`**, and concluded with **`Close()`**. The $\mathcal{C}_{\mathcal{E}}$ presented to the access method during query evaluation is therefore completely constant, in

contrast to configurations used in query planning, which may contain variables referencing input data or correlated predicates.

Alternatively, such extended navigational capabilities are also accessible via the optional **`Reset()`** interface routine, as an replacement for **`Close()`** / **`Open()`** pairs. In contrast to reassignment of scans to different tasks as discussed earlier, where **`Suspend()`** and **`Reset()`** are called in turns, calling **`Reset()`** on an open scan serves for reassigning a new constant configuration $\mathcal{C}_\mathcal{E}$ to the scan, while the scan remains within the same logical operation. Note that reconfiguration is strictly limited to $\mathcal{C}_\mathcal{E}$, while it is illegal to alter the invariant $\mathcal{C}_{\mathcal{A}_i}$. We refer to this operational mode of relational scans as *input-driven navigation*.

The same basic principle applies also, if $\mathcal{C}_\mathcal{E}$ originally contains correlated predicates. These predicates exhibit unbound variables for iterative substitution with values retrieved from a parameter input stream. In query evaluation phase, all initially unbound variables in correlated predicates are bound to the present values on the corresponding parameter stream. With this, $\mathcal{C}_\mathcal{E}$ becomes constant for the duration of one traversal through the persistently stored data set. This form of navigation is called *parameterized navigation*.

There exists a fourth form of navigation, allowing fully dynamic manipulation of a scan's configuration. For example, assume an open scan that is currently configured to some $\mathcal{C}_{\mathcal{E}_0}$ with some non-trivial selection $\sigma_{\mathcal{E}_0}$ for direct positioning. A subsequent **`Reset()`** is used to install a new configuration $\mathcal{C}_{\mathcal{E}_1}$, removing the selection from the scan's configuration, such that hereafter $\sigma_{\mathcal{E}_1} = id$ holds. Hence, the next call to **`Next()`** will move the scan from its current position to the next tuple, with respect to the currently active $\tau_{\mathcal{E}_1}$. Conversely, after repeated relative positioning, a call to **`Reset()`** may be used to install another selection $\sigma_{\mathcal{E}_2}$, such that the following **`Next()`** will move the scan to a new absolute position. Iterative navigation with intermittent scan reconfiguration constitutes a new form of *dynamic navigation*.

We emphasize that dynamic navigation must be distinguished from input-driven (e.g. materialization) or parameterized (e.g. nested-loop join) scan operations. In all three cases, the scan's configuration changes frequently, yet only in case of dynamic navigation it may change in arbitrary ways. The configuration of input-driven and parameterized access patterns change in uniform ways that resemble mere substitution of variables, and these changes are strictly limited to the selection part of the configuration. Only this limitation to mere variable substi-

tution makes these access patterns plannable, and the availability of statistics on input data and parameter streams enables reasonable cost estimation. These two implications are inevitable prerequisite for effective query optimization. Moreover, static configurations for planned traversals are generated during the negotiation process by the same algorithmic entity to which the configuration eventually applies, guaranteeing that static configurations are both valid and efficient. Dynamic navigation on the other hand uses ad-hoc configurations, assembled outside of the configured entity. This requires detailed knowledge of an access module's internals for devising adequate configurations. Dynamic configurations completely impede cost estimation and cost-based query planning. But, in spite of these drawbacks, there certainly exist practical applications for dynamic navigation, for example in *intermediate access methods* (cf. Figure.20 on page 97) using the tuple-oriented access method interface for exploiting the full navigational capabilities of its auxiliary data structures.

We conclude this section on navigational data access with a brief wrap-up of the fundamental operations of relational scans introduced so far. Any access path may be in one of three possible states. It can be *non-existent*, *operational*, or *closed*, while *suspended* is an optional fourth state, in-between operational and closed. More precisely, any existent access path allows an arbitrary number of scans, where each one is operational, suspended, or closed. With this, all elementary functionality of a generic scan operator for read-only access is covered. The following Figure.32 shows the interaction of all mandatory components of the Access Manager interface. It also comprises optional instrumentations that are necessary for realizing suspension of scans.

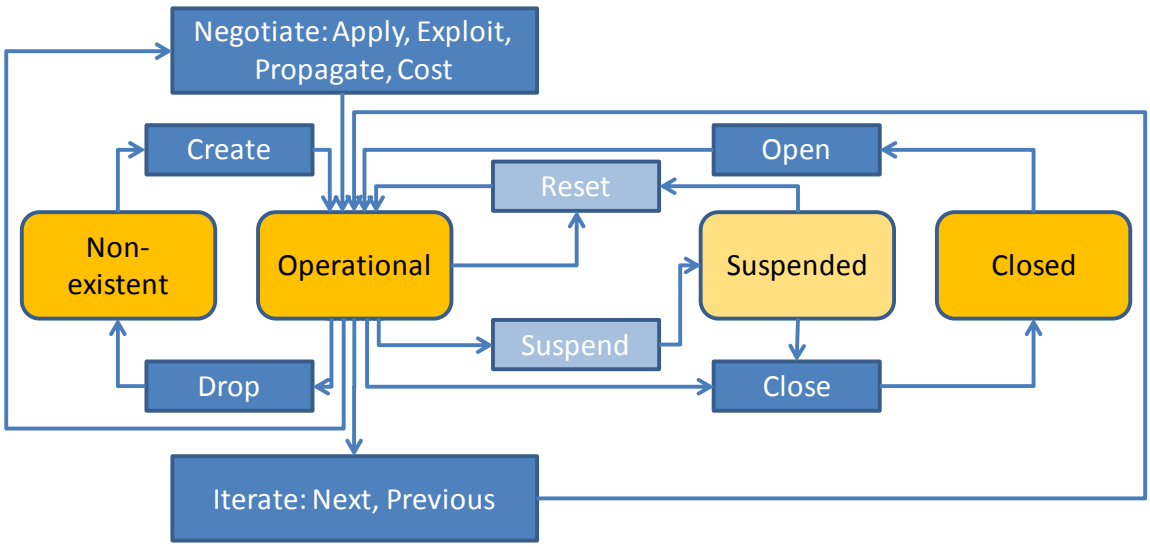


Figure.32 Transition of scan states. Interactions of the minimal set of Access Manager interface components and possible transitions of scan states. Note that any existent access path may allow an arbitrary

number of scans in the states operational, suspended, or closed. The scan state is changed by **`Create()`**, **`Drop()`**, **`Open()`**, **`Close()`** and the optional **`Reset()`** and **`Suspend()`** routines. The routines belonging to the *negotiation* complex apply to operational scans and influence the scan's configuration, while the scan state remains operational. Similarly, the Iterator routines alter the scan position of an operational scan, in correspondence to the scan's current configuration. Finally, **`Reset()`** may also apply to operational scans, allowing scan reconfiguration while the scan remains operational.

After employing any form of navigation mentioned earlier, a scan may be used for subsequent data manipulation at its current position. A detailed discussion of positioning with intermittent data manipulation will be provided in the following section.

4.3.7. Data Manipulation

The linear address space of secondary storage devices requires that data inside a database is stored according to some linearization, i.e. all tuples of a segment are stored in some arbitrary order. A segment's *primary linearization* can be exploited by a diversity of search structures for obtaining efficient access to the data, if it exhibits some functional dependency on the stored data (clustering). Linearization is usually maintained only within page-sized partitions of the physical address space (intra-page clustering). These pages are then logically assembled to the continuous address space of a segment, thereby manifesting coherent clustering on a logical level. Alternatively, inter-page clustering will also sustain physical clustering between individual pages, but this approach is only of secondary importance in the field of DBMS technology, since it generally involves significantly higher maintenance costs for reorganization. Finally, data partitioning represents a third, hybrid form of clustering, where data is functionally associated with dedicated data partitions residing in different physical extents. Thereby related data is physically clustered into one extent, but within that extent only intra-page clustering is enforced. Regardless of employed clustering, the logical address space of one segment may grow through addition and shrink through removal of pages. Additional pages may be inserted conveniently anywhere inside the logical address space, allowing dynamic adaptation to any emerging space requirements, whereas page removal may occur where logical address space becomes unoccupied. This approach allows dynamic management of the address space while preserving the imposed clustering for efficient data access.

For sustained integrity of a data access structure, access methods have to actively maintain the primary linearization whenever data is manipulated. General data manipulation subsumes insertion, deletion, and modification of data. Linearization is preserved by inserting new data into the segment's address space at the 'right place' within the primary linearization. There-

fore, access structures have to locate the correct position for insertion, provide required storage room by reorganizing data or extending the logical address space in the vicinity of the chosen insertion point, and finally insert the new data. When data is deleted, previously occupied space is released by reorganization of the affected pages. If, after deletion, the fill level of a page drops below a given limit, logically adjacent pages are merged and resulting unoccupied pages are eventually removed from the address space.

The access method interface classifies routines for data manipulation as optional, hence explicit read-only access modules may refuse to implement these routines. For achieving full data modification functionality, an access module has to provide at least implementations of the access method interface routines **`Insert()`** and **`Delete()`**. These two routines are able to cover arbitrary data modification and also subsequently necessary access path reorganizations. Both routines will actively move the scan's position either onto the correct tuple to be deleted or to a position where the tuple to be inserted shall be located. Therefore they are using navigational capabilities for repositioning the scan that are similar to those provided by the **`Next()`** routine. Technically this navigational component of data manipulation can be implemented by reusing functionality from **`Next()`**. Since an access method is definitely familiar with its own implementation, it is possible to use effective ad-hoc configuration for dynamic navigation, without going through the negotiation process. If the access module also implements the optional **`Reset()`** routine, then reuse of the access methods public interface for efficient navigation during data manipulation becomes even more attractive. After moving the scan onto the target location on the correct page, the scan will access the page using the **`FixPage()`** routine, thereby acquiring a **`WRITE_LOCK`**. Finally, the contents of that page are modified. If data insertion would exceed the page's storage capacity, then additional space is allocated by using the **`AllocPage()`** method from the storage layer interface and the new page is integrated into the logical structure of the access path. Therefore it might be necessary to acquire additional **`WRITE_LOCK`**s on logically adjacent pages (e.g. parent and sibling pages in tree-like structures), in order to redirect references to the newly allocated page. Finally, data is inserted at the appropriate position. Conversely, data deletion may trigger restructuring and merging of pages according to the logic of the particular access structure. Pages remaining unoccupied after such reorganization are eventually removed using **`DeletePage()`**. Again, references from adjacent pages to a page to be deleted will be updated after fixing the affected pages for acquiring necessary **`WRITE_LOCK`**s.

In addition to **`Insert()`** and **`Delete()`**, the access method interface also provides the optional **`Update()`** routine for direct modification of data. Since every update operation can be simulated by deletion of original data followed by insertion of its replacement, the implementation of the **`Update()`** routine does not introduce additional functionality, yet it may have performance-relevant implications. While insertions and deletions are strictly local operations, an update may cause a tuple to leap from one position in the primary linearization to another, and consequentially the tuple may move from one page to another. This happens if the segment's linearization has a functional dependency on the modified data. As an example, a tuple stored in a B-tree will leap if any of the fields constituting the B-tree's compound key is altered, since the B-tree uses the lexicographical order on the composite key attributes as primary linearization. Hence, we generally distinguish *in-place updates* and *relocation updates*.

All data manipulation routines operate on open scans, using the context of that scan as input parameter. The remaining parameters describe data to be inserted, deleted, or updated. The scan operator entity (cf. Figure.28 on page 118) encapsulating an access path implementation is responsible for retrieving this data from the modification operator's input stream and feed it as parameters to the modification routines.

```
Insert(ScanContext, NewTuple) → InsStatus
  InsStatus ::= { Inserted | Duplicate }
Delete(ScanContext, OldTuple) → DelStatus
  DelStatus ::= { Deleted | NotFound }
Update(ScanContext, OldTuple, NewTuple) → UpdStatus
  UpdStatus ::= { Updated | NotFound | Duplicate }
```

All routines return diagnostics whether the operation succeeded. An insertion may fail because it would lead to duplicates of key, and analogously a deletion may fail if the data to be deleted is not found. The access method merely indicates the result of the operation. The scan operator entity will interpret this status and decide whether it indicates an error situation or if the diagnostic is tolerated or even anticipated. Deletion initiated by a user query may well result in a **`NotFound`** status, if the searched data is not present. On the other hand, if data was successfully deleted from the base relation, then a successive deletion from a secondary index must find and delete the corresponding index tuples. These diagnostics are also used when aggregating the number of inserted, deleted, and modified tuples for returning the set manipulation's result count, a task that is also handled by the enclosing scan operator.

Modification of data in a concurrent environment inevitably entails all sorts of ramifications. *Concurrency* is used here in the sense that multiple scans are open on one segment at the same time. The host DBMS query processor will call access method interface routines of different scans *in turns*. However, the routines are still called in a mutual exclusive manner. Hence, the scans operate *concurrently*, but not in *parallel*. Parallel query evaluation requires additional synchronization, which will be discussed separately in the discourse on advances query evaluation techniques of section *4.5.3 Parallel Query Processing*. The biggest part of ramifications caused by concurrent scans, namely all necessary precautions for enabling multiple transactions to conduct concurrent scan activities on the same segment, are covered by the multi-version concurrency control of the host system. This mechanism provides reliable isolation on page-level granularity for operations of independent transactions, relieving the access method programmer from the necessity of any additional precautions. Therefore, only interactions of concurrent scans of the same transaction require additional attention in an access method implementation. We already mentioned the basic aspects of *scan maintenance*, after acquiring a **`WRITE_LOCK`** on a page. Multi-version concurrency control requires that locking with intention of modification creates a copy of the locked page. This copy henceforth represents a new version of the page, the after-image. It will be used by all scans belonging to the transaction that acquired the **`WRITE_LOCK`**, while scans of concurrent transaction continue using the before-image. As a consequence, the modifying scan has to redirect concurrent scans belonging to the same transaction to the after-image. This is accomplished by direct manipulation of concurrent scans contexts, which becomes possible, since concurrent scans on the same segment inevitably belong to the same access method type, and therefore they are familiar with the semantics of concurrent scan contexts. All scan context adaptations necessary for scan maintenance are performed by the scan that caused the creation of the new version, immediately after acquiring the lock. This strategy of direct scan maintenance allows other scans to continue their work, without consciously recognizing that version, location, and contents of their fixed page have changed. The same strategy is always applied, when the contents of a page are modified. If a concurrent scan is positioned on a page being modified, the modification will insert, delete, or update tuples in the vicinity or at the exact position of that scan. In case of a modification in the vicinity, the tuples in the page may shift positions and the foreign scan has to be adapted in its scan position. On deletion of a scan's current tuple, that scan has to be moved to a position *before* the logically successive tuple according to that scan's effective $\tau_{\mathcal{E}}$ configuration. In case of insertion, scan maintenance has to be

conducted such that recent modifications will become visible to concurrent scans, if those scans move towards the new tuple according to their τ_ε configurations.

This strategy of active scan maintenance is not limited to altered tuples and scan positions, but it applies also to all page modifications effectuated by data manipulation. In B-trees, for example, modification on leaf-level have to be treated in the same way as their side-effects on internal nodes of the index part of the tree, manifesting as page insertions, deletions, splits, or merges. This includes in particular changes in the segment's description page, as these modifications also have to be reflected immediately in the scan contexts of concurrent scans operating inside the same transaction. When a call to a data manipulation routine completes, all tasks concerning modification on that segment must have concluded. The access structure and the contexts of all concurrent scans have to be in a state enabling them to cope with arbitrary subsequent calls to any of their access method interface routines. Manipulation of redundant data stored in separate data structures (indexes) and possible referential constraints to other tables will be discussed separately. This will be subject of the following section *4.3.8 Data Integrity*.

The principles of singleton manipulations can be generalized, when dealing with mass insertion, deletion, and updates. Mass manipulations essentially perform the basic operations described before in a repetitive fashion. Therefore, the scan operator is fed with input streams defining on which data the operation will be performed, i.e. a stream of tuples to be inserted or a data stream identifying tuples to be deleted. The performance of such mass manipulations can be significantly improved, if the input streams controlling the manipulation deliver their data in an opportune sort order. For example, insertion of tuples arriving in a sort order that resembles the primary linearization of the base relation will be significantly faster than random insertion. This sort order facilitates modification in one single sweep over the base relation, allowing 'clustered' modifications, as opposed to perpetual random repositioning within the data structure. This observation applies in the same way to mass deletion and mass updates. The modification scan may actively request an opportune input order during negotiation of its input requirements. In this case, the input sort order has only implication on the operator's performance characteristics, but not on its functionality. Hence, such a supportive sort order would be requested as an *optional* input requirement.

In some cases, mass manipulation has to be conducted under additional precautions. In particular circular references, where the modified relation serves as input for the modification, have to be specially treated (e.g. `INSERT INTO T SELECT a*2 FROM T`). The same holds for self-

references in update statements that cause tuple relocations (`UPDATE T SET a=a*2`), and for sub-queries in delete statements (`DELETE FROM T WHERE a > (SELECT AVG(a) FROM T)`). This form of circular dependencies is automatically recognized by the host system's SQL compiler, considered by the query optimizer, and finally resolved by employing an adequate strategy in the query processor. The resolution of circular references is based on the principle of *deferred updates*, there the modification is conceptually split in two phases. The first phase will compute the complete information describing the planned modification (tuples to be inserted, deleted, or updated) and retain it in a temporary storage area. The table to be modified remains unchanged. The actual modification is performed in the second phase, using input streams from the temporary store and thereby eliminating all circular references. All this is automatically handled by the host system, completely without any support from the affected access module.

4.3.8. Data Integrity

Until now, we examined the effects of data manipulation on one single access path. But manipulation on a base relation may trigger numerous side-effects for maintaining overall data integrity in a database. These side-effects cover maintenance of secondary indexes, checking and propagation of integrity constraints and the execution of automatic database triggers. In the following, we will address each of these three topics separately and finally discuss interactions between these individual tasks.

Indexes

The modification of data in a base relation implicates updates of redundant data stored in secondary indexes. Insert and delete operations inevitably require a corresponding manipulation on all secondary indexes, as indexes always maintain a strict one-to-one relationship between base tuples and index tuples. Row updates on the other hand, do not necessarily affect all indexes, since indexes typically constitute a lean projection of the base relation. If the update modifies only columns that are not part of one particular index, then this index remains unchanged.

There exist two general strategies for maintaining indexes. If a set of tuples is to be modified, integrity can be preserved via *singleton index maintenance*, i.e. each individual base tuple modification is immediately followed by maintenance of all indexes in a one by one fashion. Alternatively, *set-oriented index maintenance* updates all base relation tuples in a first phase, followed by accumulated maintenance of every separate index, each in an individual phase.

Singleton index maintenance allows processing manipulations on-the-fly, without any requirements for temporal storage. The modification of one individual tuple is processed completely on all affected access structures. Data from the input stream describing that modification becomes obsolete and may be discarded as soon as the last index structure has been updated (cf. Figure.33a). This form of index maintenance uses multiple modification scans at the same time, one for the base relation (potentially accompanied by a separate scan on the IK-tree) and one for every redundant segment. These scans are operated in turns, leading to non-locality in index maintenance, which may cause severe performance penalties. We already mentioned that manipulation will benefit if its input data arrives in a convenient sort order, such that modifications on the base relation are performed in form of one single traversal of the base relation, following an adequate linearization. This input sort order is negotiated during query optimization phase and often matches the primary linearization of the base relation. While the negotiated input order can be assumed to be optimal for handling the base relation, it will surely have adverse effects on secondary index maintenance. Base relation and all secondary indexes are inevitably based on pairwise distinct linearizations, otherwise they cannot serve as expedient alternative access paths. Hence, tuplewise modification on indexes provokes a random modification pattern, requiring frequent and potentially wide-spaced repositioning. These access patterns may lead to a significant overhead, if different tuples on the same page are modified in separate and nonconsecutive operations, with intermediate repositioning to other pages. Ultimately, they will cause cache frame thrashing and unfavorable random I/O profiles, where the same page is physically read, modified, and written multiple times. Especially in case of bulk operations on many secondary indexes, this effect is likely to obliterate performance benefits gained through on-the-fly processing.

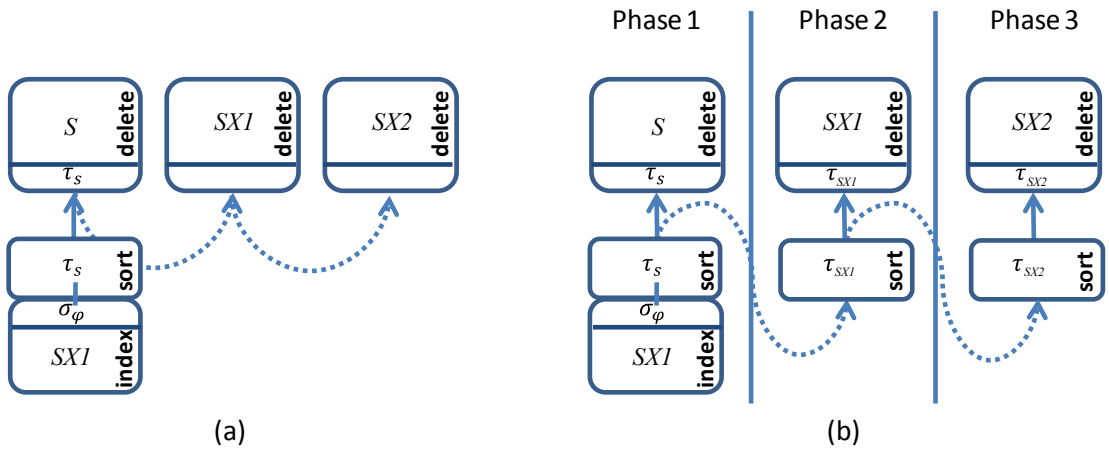


Figure.33 Data integrity maintenance across redundant data structures. A deletion from table *S* with two secondary indexes *SX1* and *SX2* is being conducted using alternative strategies (a) and (b). In both

cases, the input stream describing tuples to be deleted is retrieved from index *SX1*, enforcing some given predicate φ. *Singleton index maintenance* (a) processes immediate deletion for every tuple on all participating data structures. Negotiation will install a beneficial sort order τ_S for modification on the base relation *S*. Indexes are also maintained in the preset order τ_S. *Set-oriented index maintenance* (b) conclusively modifies one access structure at a time, while the input stream is retained in temporary storage, where it is available for modification of redundant access structures in subsequent phases. Temporary storage opens the opportunity to reorder input data. Thus, every access structure may install its preferred insertion order τ_S, τ_{SX1}, τ_{SX2} during negotiation.

In contrast, a set-oriented approach to index maintenance in multiple phases requires temporal storage of the data set defining the planned modification. This allows repetitive processing of identical data in each modification phase. Although temporal storage seems disadvantageous at first glance, it opens the opportunity of reordering data to suit the preferences of a secondary index. Reordering becomes possible at the expense of additional computational complexity, with the prospect of return of investment through optimized locality in data manipulation. The optimal manipulation order for every single index is determined via common negotiation with scan operators on each index segment during query optimization phase. During query evaluation, only one scan is open in each modification phase, using the configuration established via negotiation. This allows manipulating every index in one single traversal, following the index's preferred linearization.

We already discussed scenarios where data dependencies, such as self-references in data manipulation queries, require employment of the *deferred update* mechanism for achieving algorithmic correctness. This inevitably entails temporal storage of the input data set and thereby appoints the strategy to be applied. In all other cases, the choice between the two presented strategies offers a trade-off to be considered by the query optimizer. While tuple-wise modifications show disadvantageous random behavior when maintaining secondary indexes, they are able to conduct all operations without additional temporal storage or additional computational expenditures for reordering data sets. As a variant, it is also possible to combine both approaches, by reordering updates for some secondary indexes, while others are updated randomly. Finally, two indexes may share a common prefix in their preferred lexicographic input order. Using that prefix sort order for both indexes might be sub-optimal for one individual index operation, but it may lead to a superior evaluation plan by reducing overall costs. The optimizer may decide for every single segment which strategy to apply, using locally individualized negotiation with each scan operator and global cost-driven query planning. This decision may be influenced by the number of involved indexes, the amount of

available temporary storage, effective costs for reordering, and finally by the estimated costs of modifications with and without optional input orders.

Hence, all functionality necessary for appointing efficient index maintenance is already provided via standard negotiation. Yet, the access method must be aware that suggesting optional input sort orders to the query optimizer may lead to beneficial alternative strategies. The host system provides all prerequisites for facilitating these strategies, including temporary storage, reordering, and other necessary precautions for guaranteeing correctness. The access module must accept that this strategic decision is made by the optimizer, and support it by providing ample flexibility and reliable cost estimation. During query evaluation 'normal' data manipulation and index maintenance are indiscernible from the perspective of an access module. It will experience a series of repositioning and manipulation on the segment's data structure, using intermittent calls to its interface routines for navigation and data manipulation. The logic controlling index maintenance is completely encapsulated inside the host system's query processor.

Constraints

With the constraint mechanism, SQL provides a variety of instruments for preserving and enforcing relational integrity. *Check-constraints* (often used as domain constraints) serve for restricting the values of a column to a certain subset of the column data type domain. *Key-constraints* and *unique-constraints* guarantee that individual columns, or a combination of columns, do not contain duplicates. *Referential constraints* (also foreign key constraints) ensure referential integrity between individual relations. All types of constraints have the same general functionality. If data manipulation violates a constraint condition, then the responsible data manipulation is cancelled, all its previously completed effects are undone, and the system reports an integrity violation error.

This behavior is conducted by the host system's query execution engine, by interpreting the diagnostics returned as **`ScanStatus`**, **`InsStatus`**, **`DelStatus`**, and **`UpdStatus`** by navigation and data manipulation routines. Besides acknowledging that insertion, deletion, or update has succeeded, these diagnostics can also indicate integrity violations, namely the non-existence of a searched tuple and duplicate of keys. But not every access path is able to recognize duplicates in equally efficient manner. Therefore, an access method is intentionally appointed to the purpose of enforcing a certain constraint and consequentially it must invariably check for violations and report them reliably. If an access structure is not able to enforce

the assigned constraint efficiently, then it may reject that constraint at table creation time. For example, a simple linear access path, organized as linked list of pages, is only able to enforce a unique constraint or primary key constraint by sequentially scanning its complete data set for duplicates, before inserting new data. Such a data structure should reject these forms of integrity constraints, thereby forcing a schema designer to install such constraints either via an adequate secondary access path (e.g. a B-tree index) or to choose a more suitable access method for the primary access path.

Referential constraints are validated by inspecting adequate access paths of referenced tables. Propagation of data manipulation to referencing tables via referential constraints (e.g. `ON DELETE CASCADE`), are accomplished via common data manipulation. All this is conducted by the host system, using available mechanisms of negotiation, navigation, and manipulation. The host system is also responsible for choosing efficient strategies when preserving referential integrity. Checking for integrity violations can involve considerable computational complexity that may even exceed the costs of the actual modification. Hence, a general trade-off exists between an *optimistic* approach, attempting inexpensive lazy constraint checking but entailing possibly devastating undo operations, and a *pessimistic* approach, using early but potentially more expensive constraint checking and thereby avoiding costly undo operations. The strategy is chosen by the host system's query optimizer and an access method implementation may remain unaware of these considerations. An access module may however help confining the effects of necessary undo operations to the current modification operation by implementing the optional savepoint feature.

Database Triggers

Database triggers represent an additional means for preserving the integrity of a database. Similar to constraints, which are validated whenever data is inserted, updated, or deleted, triggers are used to execute procedural code in response to data manipulation events. In contrast to constraints, whose primary scope is enforcement of integrity on a strictly relational level, triggers allow automated enforcement of complex aspects of application logic and self-management of data by execution of application-defined procedural SQL code. In addition, triggers are useful for logging and auditing data manipulations, or for automated data replication.

The SQL standard distinguishes several general trigger types, namely *statement-triggers* that are executed once for a DML statement, and *row-triggers* that are executed once for every

row update, and therefore possibly multiple times per statement. In addition, it is possible to define the exact chronology of data manipulation, by defining triggers that are executed before or after the data is manipulated (*before-trigger* and *after-trigger*). Finally, it is possible to discern the type of data manipulation (insert/ update/ delete) that will activate a trigger. As a consequence, it becomes possible to define triggers for a multitude of data manipulation events, as any possible combination of trigger types is permitted:

$$\{\texttt{row, statement}\} \times \{\texttt{before, after}\} \times \{\texttt{insert, update, delete}\}$$

Database triggers may execute arbitrary procedural SQL statements, including DML statements, which in turn may activate other triggers. Therefore, it becomes possible to devise transitive and even cyclic trigger dependencies. With this, database triggers represent a highly dynamic and complex instrument for database schema designers. But this complexity affects only the host system, which is able to insulate access method implementations completely from the necessity of any special precautions for supporting fully-fledged database triggers. Again, from the perspective of an access method, triggers effectuate just a succession of access method interface calls, navigating and manipulating data. The complex logic of database triggers is encapsulated inside the host system's query processor.

Interrelations

All the mechanisms described above are to be executed in an exact chronological sequence, which is explicitly appointed by the SQL standard. These conventions ensure that possible dependencies between individual tasks are resolved in a deterministic and comprehensible way, as depicted in Figure.34.

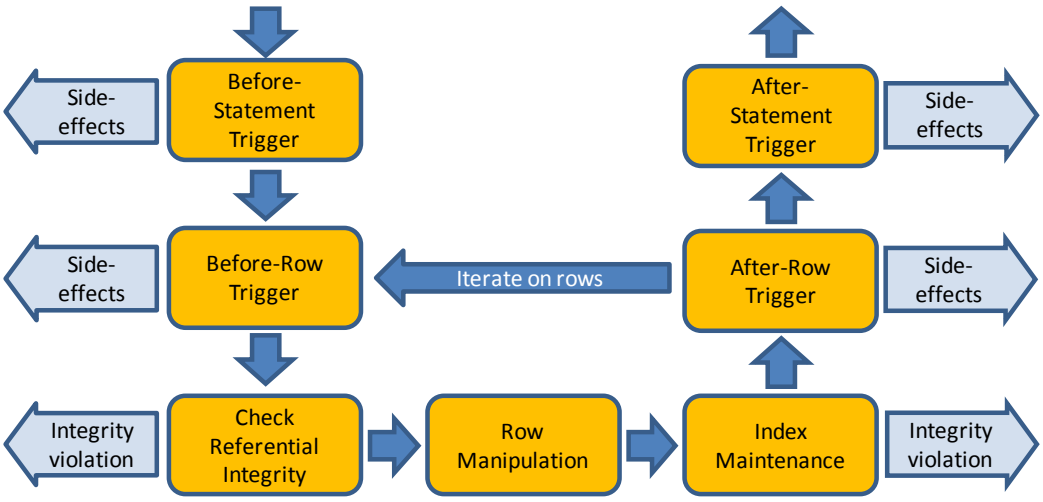


Figure.34 Logical chronological sequence of integrity maintenance. The SQL standard appoints a strict succession for integrity maintenance tasks, where individual tasks may again provoke side-effects, leading

to nested data manipulation. Violation of referential integrity is detected in dedicated checks, while unique constraints are validated during manipulation of appointed primary or secondary access paths that have been put in charge of enforcing these constraints. Occurrences of any form of integrity violation and other dynamic errors trigger exhaustive undo operations revoking any dependant manipulation. Hence, the complete manipulation process becomes one atomic operation.

In spite of this strict definition, the query plan optimizer is free to reorder individual tasks for gaining performance benefits, as long as the outcome of the complete data manipulation process is identical to that of the demanded schedule of operations. Devising valid and beneficial strategies requires sophisticated query planning. The necessary logic is concentrated in the host system's query optimization component, which is responsible for maintaining data integrity on a global scale. Through negotiation and cost estimation, access modules are participating actively in this process and thereby exert a certain influence on significant aspects of the final query plan. According to this plan, the scan operator entity manipulates the individual access modules affected by data manipulation through their access method interface. During this process, the operational scope of access modules does never exceed that of management of local data structures of its own segment, which covers manipulation of the access structure and maintenance of concurrent scans of the same transaction. Hence, from the perspective of an access module, maintaining data integrity is indiscernible from normal data manipulation.

4.3.9. Savepoints

For sustaining data integrity, the SQL standard demands that data manipulation is a strictly atomic operation. The existence of integrity constraints implicates that data manipulations may fail in case of integrity violations. The same applies for other dynamic error situations that may occur during these operations, including arithmetic exceptions, lock conflicts, insufficiency of resources, etc. In these cases, the entire manipulation must be undone, along with all dependant operations that have been triggered during its progression. In the transactional environment of DBMSs, this can be accomplished thoroughly by aborting the transaction enclosing the defective manipulation. However, this crude approach will exceed the aspired goal in many cases, in particular if the affected transaction also covers other extensive operations that have been conducted previously and did complete successfully. In these cases, it is desirable to preserve uncommitted work and undo only the effects of the manipulation that went awry. In order to accomplish such selective undo operations, the host system requires support from all access structures involved into the data manipulation process. As already indicated, this support is provided via the savepoint feature, which is optionally

implemented by an access path module. Savepoints are based on the following lean interface and a corresponding simple protocol.

```
DefineSavepoint(ScanContext) → ∅
ApplySavepoint(ScanContext) → ∅
```

The beginning of data manipulation on an access path is marked by defining a savepoint on an open scan, before applying any changes. Consistent savepoint definition is guaranteed by the host system, which is calling the corresponding access module's **`DefineSavepoint()`** routine. Savepoints preserve the state of a scan before manipulation begins, which essentially is accomplished by retaining a snapshot of the scan context. During the following data manipulation, the scan may navigate freely and conduct a multitude of elementary update operations, where both activities will affect the scan's context.

By definition, any scan participates only in one single data manipulation at any time, and consequentially it has to maintain at most one savepoint. If another savepoint is defined on one particular scan, this denotes the beginning of a subsequent operation and the previous savepoint becomes obsolete. Similarly, a savepoint is discarded when its scan is closed. The host system will define additional savepoints on other access paths as required, as data manipulation progresses and side-effects on other segments unfold. Thereby the host system guarantees that all necessary savepoints are installed in due time. Whenever an access path implementation becomes involved that does not support the optional savepoint feature, then that scan's original state immediately before the manipulation cannot be reinstalled by these means. An exception to this rule is represented by scans that were opened dedicatedly for the current manipulation. In this case, a scan's original state is obviously reinstalled by simply closing it, independently of the scan's support for savepoints. The availability of the savepoint feature for undoing data manipulation is monitored by the host system. It will automatically resort to abortion of the whole transaction, in the adverse case that savepoints are not available and data manipulation fails due to a dynamic error. In other cases, where an error occurs and all participating access modules support savepoints, **`ApplySavepoint()`** will be used to reinstall the preserved scan contexts, thereby resetting all scans to their original states. This particularly includes unfixing pages that are fixed in the current scan context, but were not fixed at savepoint definition time, and reacquiring fixes on all initially fixed pages. The host system's storage layer will actively participate in this operation by supplying pages with the correct contents at savepoint definition time.

Although it is vital that all involved access module implementations support the savepoint feature for its practicability, the bulk of the savepoint functionality is encapsulated in the host system's multi-version concurrency control. At the beginning of a data manipulation, the storage layer is informed via its own **`DefineSavepoint()`** interface that data manipulation is about to begin in the context of a given transaction. Since data modifications are atomic operations, they are serialized in one transaction, and consequentially at most one manipulation per transaction may be in progress at any time. But in contrast to the data access layer, the storage layer must be able to deal with multiple transactions at the same time, and consequently it must be able to support as many savepoints, making savepoints a much more sophisticated feature on this system layer. If a savepoint is applied, then all pages have to be restored into their states before the modification. The storage layer accomplishes this by applying logging information for undoing all page modifications issued by the defective manipulation. Therefore, the essential information necessary for applying savepoints on the storage level is an identification of the first log entry made by the current data manipulation, determining how far the logs have to be processed in reverse until the savepoint is reached.

Hence, general savepoint information is quite compact, as it consists of snapshots of affected scan contexts and the ID of the manipulation's first log entry. This assumption, and the fact that any scan has to maintain at most one savepoint at any time, allows us to dispense with a separate interface for explicitly releasing a previously defined savepoint. If data manipulation succeeds, its savepoints technically remain installed, but the host system will make sure never to apply such abandoned savepoints. As already discussed, these savepoint are eventually cleared when new savepoints are installed as scans are reassigned to a different manipulation within the same transaction, or when the corresponding scan is eventually closed. This protocol guarantees that at the end of a transaction all its savepoints are cleared, because all scans are closed.

In case of some severe errors, for example if the connection to the client application is disrupted, the DBMS will have no choice but to abort active transactions, even if the savepoint feature is available, since the DBMS requires application logic in order to recover into a consistent state when interrupted. The same applies to lock conflicts, since savepoints are not adequate for resolving such contentions, i.e. even if a savepoint is successfully applied, isolation necessitates to retain all locks acquired during the defective manipulation. Therefore, it is inevitable to abort the complete transaction that caused the conflict, thereby releasing its locks and eliminating the lock contention. As a general rule, the DBMS is free to decide what

actions are to be taken in case of a dynamic error. Consequentially, the savepoint feature is employed only in error situations where it is applicable, where selective undoing is reasonable, and where all participating access modules support the savepoint feature. Otherwise, the complete transaction is aborted. If a savepoint is applied successfully, then the original error is reported to the database application. The application may then choose how to proceed in this situation. Typical strategies include retrying the manipulation, proceeding to some alternative work, or manually aborting the transaction.

4.3.10. Locking & Concurrency

Locking and concurrency settings in a DBMS offer a certain trade-off, influencing lock granularity, concurrency in form of expected conflict rates, and implementation and computational complexity. The following Figure.35 sketches dependencies between these conflicting goals.

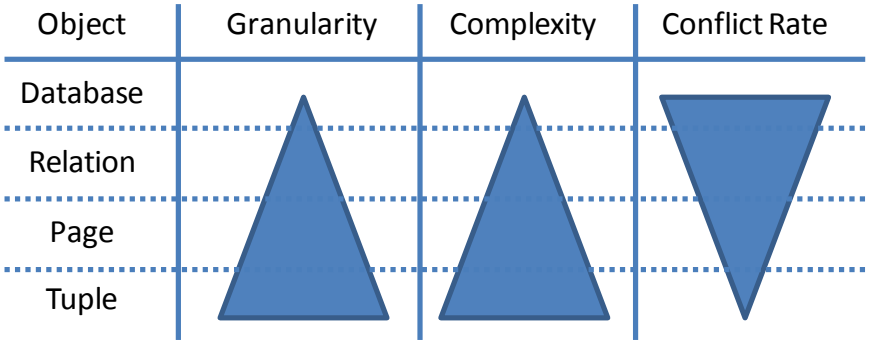


Figure.35 Trade-off in lock granularity. DBMSs generally allow a variety of lock granularity settings, that enable the system to adjust to isolation and concurrency requirements of a particular application. Lock granularities typically comprise global locking of the complete database, table locks managing access to individual relations, page level locks and finally locks of singular data rows.

We already discussed how the host system's storage layer provides reliable locking on page granularity, if the corresponding page fix operations are attributed with lock types describing the intention to read or write during an imminent page access. Moreover, the host system's multi-version concurrency control ensures that every fix operation is provided with the correct version of a page, in accordance to the enclosing transaction's isolation settings. These mechanisms permit coarse table locking and fine granular page locks, while operating without any functional requirements of direct support from access module implementations. If locking on row level is desired, then the access module must actively participate in locking by explicitly operating the host system's lock manager interface.

```
Lock(SegID, PNO, OID, Lock) → ∅
```

```
Unlock(SegID, PNO, OID, Lock) → ∅
  Lock ::= {READ_LOCK | WRITE_LOCK | EXCLUSIVE_LOCK }
```

The lock interface extends the hierarchical locking facilities provided by the **FixPage()** routine. It allows locking of arbitrary objects within a segment denoted by **SegID** and located on the page identified by **PNO**. The access module is responsible to provide unique object IDs (**OID**) for every lockable object in its segment. Any numeric ID may serve as **OID**, for example the tuple's primary key, an arbitrary key surrogate, or simply the **TID**. A call to the lock interface will acquire the necessary locks as required by the RAX protocol (Figure.25, page 110). When locking on row granularity, then **FixPage()** must be operated with the dedicated **INTENTION_READ**, **INTENTION_WRITE**, **INTENTION_EXCLUSIVE** settings for installing the hierarchical intention locks according to the RIX protocol (Figure.26, page 110).

As an alternative to this locking service provided by the host system, an access module may operate the **FixPage()** routine with the **NO_LOCK** setting. This allows the access module to implement its own lock mechanism, thereby completely bypassing the host system's lock manager. The **NO_LOCK** mechanism is merely provided for completeness, but as a general rule it is not advisable circumvent the host system's lock manager.

4.3.11. Transactions & Consistency

In principle, database consistency and transactional isolation is completely sustained by the host system, without any intervention from access module implementations. Scans cannot exist across transactional boundaries and the host system will ensure that all scans are duly closed before a transaction ends. Hence, the transition of transaction contexts is irrelevant for access modules. Still there might exist occasions where an access module wishes to be informed that such a transition is in progress. For this purpose, the access method interface possesses a collection of *optional* interface routines to be used as callback hooks. Whenever the state of a transaction changes and the corresponding callback routine is implemented by an access module, the routine will be called by the host's Access Manager. All routines have the transaction's unique **TaID** as their only input parameter.

```
Begin(TaID) → ∅
Prepare(TaID) → ∅
Abort(TaID) → ∅
Commit(TaID) → ∅
```

A practical example for these callback hooks exists in access modules operating as *data integration layers*. These modules access remote data repositories that are not based on the host system's internal storage facilities. If these repositories support transactions, then transactional transitions must be propagated to remote sites. This can be accomplished in a natural way by utilizing the respective callback hooks.

4.3.12. Logging & Recovery

The Access Manager framework makes no arrangements whatsoever for permitting interaction of access module implementations with the storage layer's logging and recovery mechanisms. Any access method may rely on these system intrinsic services, as long as its persistent storage is based on the host system's internal storage facility. In case of access modules acting as data integration layers and accessing remote data repositories, similar services might be provided by external information systems. If this is not the case, then it might become necessary to provide a custom implementation of logging and recovery functionality inside the access module.

4.3.13. Administrative Tasks

All remaining access method interface routines are optional and serve for diverse administrative tasks. Since access modules physically exist as dynamically loadable libraries that are mapped into the host system's address space, the interface provides routines for initializing and releasing internal structures that might be necessary for the module's internal resource management.

`OnLoad()` → ∅

`OnUnload()` → ∅

As its name indicates, **`OnLoad()`** is called after the library was loaded into the DBMS's address space. It is typically executed when the first access path of a certain type is about to be created or when that access method is accessed for the first time after the database service was started. The inverse function **`OnUnload()`** is called if an operation on a segment terminates. This happens if the last access path of the corresponding access method type is dropped or if the database service shuts down.

Altering access paths

The SQL standard envisions a number of possibilities to change existing access structures.

```
ALTER {TABLE | INDEX} [(custom_spec)] <relation_name>
   {ADD | ALTER | DROP} {COLUMN | CONSTRAINT} <element_name>
   [<new_definition>]
```

Typical examples range from deferred adjustments of data type properties (e.g. precision, scale, etc.), over changing data types and renaming columns, to addition or removal of complete columns in base relations and indexes. Note that the SQL syntax above also allows specification of an alternate `custom_spec` to be passed to the access module for interpretation. The provision of facilities for conducting such modifications is optional, and an access method provides this feature by implementing the **`Alter()`** interface. If an access method does not provide an implementation for this method, the Access Manager framework will simulate it by creating a new access path according to the altered definition, filling it with data from the original structure, which is subsequently discarded. As this approach necessitates twice the storage requirements of an access structure in the database's permanent storage area for a short time period, the Access Manager may alternatively choose to copy the access structure's contents to its temporary storage area, thereby also allowing in-place modification of access structures. All data to be inserted into the new access structure undergoes SQL's automatic *type adaptation*, providing necessary adjustments to an altered table definition. Missing values for recently added columns are filled in by SQL's *default value* mechanisms. In contrast to the aforesaid, modifications of column names however will only affect the data dictionary, but not the actual access structure.

`Alter(TaID, DescPage, SegID, CreateSpec, CustomSpec)`
`→ ScanContext`

The routine's signature is identical to that of the **`Create()`** routine and it also exhibits a very similar behavior. The task of **`Alter()`** is to compare the table definition provided as **`CreateSpec`** and **`CustomSpec`** with the original table definition in **`DescPage`**. It has to detect differences and apply them by altering the access structure correspondingly. Similar to the **`Create()`** routine, the function call results in an open scan on the modified segment, which is represented by the **`ScanContext`** result parameter.

Like any other data manipulation, the whole operation is executed within a transactional context and dynamic errors that might occur in its progress will undo all modifications. In contrast to normal writing manipulation on a segment, the **`Alter()`** operation is conducted

under an *exclusive relation lock* that is automatically established by the host system at the beginning of the operation.

Reorganization & Defragmentation

If a relation is populated using a mass insertion mechanism, then the arrangement of data on the physical address space of a persistent storage device often corresponds to its logical linearization. But all search structures based on page-oriented storage tend to fragment over time, if the contained data is undergoing repeated modifications. Data modifications trigger overflowing of pages which necessitates the allocation of new pages. These new pages extend the logical address space between existing pages, but they are potentially allocated at remote addresses in the physical address space. Also merging and releasing of formerly used pages is a possible consequence of data manipulation. Reusing released pages will eventually cause mingling of pages from different segments. Hence, the logical linearization of data is gradually dissociating from the physical linearization. Although this fragmentation of physical address space does not cause any fundamental functional problems, retrieving data in an ostensibly sequential logical order will actually entail reading from random physical addresses, generating significantly higher I/O costs than sequential reads and ultimately resulting in poor performance of a heavily restructured access path. On conventional hard drives, and without further precautions, a fragmented search structure will be outperformed by one order of magnitude compared to a non-fragmented counterpart containing identical data. This effect is countered by reorganizing (defragmenting) the data structure. Defragmentation involves physically moving data, usually page-wise, but also intra-page reorganization is possible. In addition, the operation is accompanied with updates of potentially complex networks of inter-page references and chaining. These extensive implications make data reorganization a highly costly task, and therefore it is either initiated on explicit request from the system administrator, or as an automatic maintenance task that utilizes hardware resources during idle periods.

Since the host system cannot know how pages and inter-page references of a certain access structure are organized, such reorganization must be provided by the access module itself. This functionality is supported by the storage layer, allowing systematic allocation of new pages through its generic **`AllocPage()`** routine. Defragmentation is triggered by the following DML statement.

```
ALTER {TABLE | INDEX} <relation_name> MOVE <destination>
```

Here the `<destination>` clause defines the target area for the defragmented access structure, either in form of a physical address interval whose boundaries are specified as page numbers or via the name of a logical database *extent*. It is legal to specify a target area that overlaps with the storage area occupied by the fragmented data structure, thereby requesting in-place defragmentation. However, defragmentation is strictly limited to one single segment, i.e. the specification of a target area that is known to be occupied by other access structures will never move those pages out of the way, but it will merely attempt to move the specified segment as close as possible to the target area.

In allusion to the representation of this functionality in SQL, defragmentation is accessible through the optional **`Alter()`** interface routine. Another analogy to altering existing access structures is the identical simulation of this optional functionality with mandatory access method interface routines. Reorganization in absence of an **`Alter()`** routine is achieved by building a defragmented access structure at its new target location and subsequently discarding the original fragmented structure. If in-place defragmentation is requested, then the Access Manager will automatically consider redirecting the access structure's contents to a temporary storage area.

Checking & Reporting

Database systems offer several complementary, autonomous and redundant systems for reliable safekeeping of data. Even in the presence of faulty hardware, data integrity can be sustained to a certain degree. But database systems, like all complex systems, are not completely error-free and both software and hardware defects may ultimately compromise the validity of a database. For a monolithic software system it is possible to maintain a constant quality standard by effectively testing all involved components before releasing the product. An extensible system on the other hand, has to trust in correctness and integrity of subsequently added components. To compensate for this shortcoming, the Access Manager is equipped with an array of mechanisms for constantly verifying correctness, organized in three major stages. As a first stage, the host system will perpetually conduct inexpensive integrity checks during normal system operation. If some discrepancy is detected, the system will immediately report the source of the problem, and depending on the fault's severity, the system will either cancel the currently ongoing operation, or in case of grave errors, it will abort the surrounding transaction. Therefore, this first stage of system verification is able to provide protection against problems that are detectable instantly and inexpensively, and it does also guarantee fail-safety through its ability to undo faulty work. If a problem is detected

after the transaction enclosing some erroneous modification was already committed, then this mechanism provides merely error detection. Monitoring of system activities is performed during normal operation, and to this end, no particular extensions in the Access Manager interface are required. However, each access module may voluntarily contribute to system integrity by performing its own consistency checks during normal operation.

The remaining two stages of system verification operate in the course of dedicated checks, to be invoked via an external maintenance tool. The second stage examines data integrity across different segments by matching data from redundant or dependant access structures, e.g. base relations, indexes, and referential constraints. This test is performed using the Iterator interface of the access structures, and from the viewpoint of an individual access structure, this check represents a normal retrieval operation. Actual integrity checking is performed on a higher level, by the Access Manager framework, where the data from redundant access structures is compared. During this operation, the access structure remains fully operational and normal concurrency precautions apply for ensuring the required isolation from concurrent operations. Only the third stage of checking requires an access module to implement supplemental functionality in form of an *optional* interface routine:

`Check(ScanContext) → Report`

When called, the access method will perform a series of consistency checks that are suitable for validating the integrity of one isolated access tructure. In contrast to the online integrity checks of the first two stages, where only comparatively inexpensive checks are reasonable, the **`Check()`** routine's focus lies on thorough testing, using potentially complex algorithms and accepting corresponding costs. The most important check to be initiated in this stage will search for irregularities in the chaining of pages. This validation is typically accomplished by traversing the search structure via redundant chaining, i.e. brother-chaining and parent-child-chaining in tree structures. Other tests may survey the validity of invariants and assertions of a particular access structure, such as compliance with guaranteed page fill levels, fan-out, acyclicness, or comparison of the data's actual sort order with the expected linearization. Generally, all these tests are executed while the database is fully operational. But concurrent modifications of the data structure under examination may influence the test. Hence, the access method itself may choose whether the access structure is online or offline while these tests are conducted. An adequate measure to protect an access structure from concurrent access during **`Check()`** operations is to establish an exclusive lock on the access structure's description page.

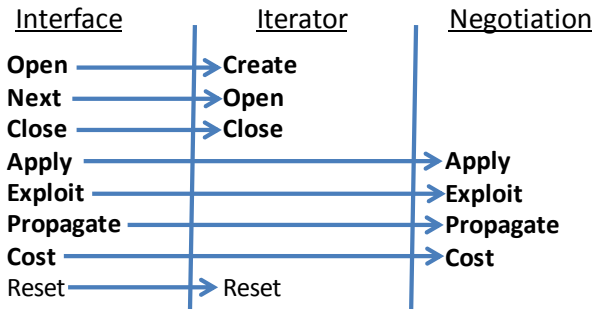
The result of a **`Check()`** invocation is a comprehensive, textual report providing detailed information in human-readable form. In case of uncovered integrity violations, the diagnosis should contain rich information for analyzing the source of a problem and it should also provide useful information for its resolution. The routine may also repair minor deficiencies, if this can be accomplished in a reliable way. Nevertheless, such corrections are to be mentioned in the final report. If no errors are found, this report shall present conclusive information on the status of an access path, e.g. current storage requirements, average page fill level, degree of fragmentation, etc. This information is intended to support schema analysis and refinement during the database design phase or schema revisions. The inefficiency and laborious nature of structural integrity validation makes it generally unattractive for repeated, automated monitoring in a productive environment, hence such reports are typically generated only on explicit request of the database administrator.

4.4. Relational Operator Interface

The preceding section illuminated all important aspects of the *access method interface* for integrating implementations of custom access methods into a host DBMS's query optimization and query evaluation procedures. Access methods represent a specialization of general relational operators and therefore they occupy a unique position among the set of relational operators and their algorithmic implementations. In this section, we will address integration of custom implementations of generic relational operators into the host DBMS. We will show how these implementations are subsumed in the class of *relational algorithmic modules* and we will demonstrate how a lean subset of the access method interface together with an analogous protocol is sufficient for embracing the complete functional spectrum of arbitrary relational operators.

Henceforth, we address this interface of algorithmic entities implementing generic relational operators as the *algorithmic module interface*. Similarly to the access method interface, it is based on the Iterator Model. But 'ordinary' relational operators require no means for access path creation, data manipulation, transaction handling, etc., and therefore their interface consists only of the elementary Iterator routines **`Open()`**, **`Next()`**, and **`Close()`**. This mandatory interface may be augmented with an optional **`Reset()`** routine. For active participation of such algorithmic modules in the query optimization process, the interface also comprises all assets for negotiation, namely **`Apply()`**, **`Exploit()`**, **`Propagate()`**, and

`Cost()`. The following Figure.36 illustrates the common interface for all algorithmic mod-



ules.

Figure.36 Algorithmic Module Interface. This interface provides the required abstraction of specialized functionality of one individual algorithmic module implementing a relational operator. It allows combining arbitrary Iterator-based relational algorithms to complex query execution plans.

We already encountered this particular interface definition in form of the scan operator's external *relational operator interface* (cf. Figure.29 on page 119). Similar to a scan operator encapsulating an access module, any algorithmic unit is embedded in a generic relational operator entity (depicted in Figure.37), which is an integral part of the host system. As the scan operator is a specialization of a generic relational operator, they both export the universal relational operator interface, which allows handling arbitrary algorithmic implementations in a uniform way during query evaluation.

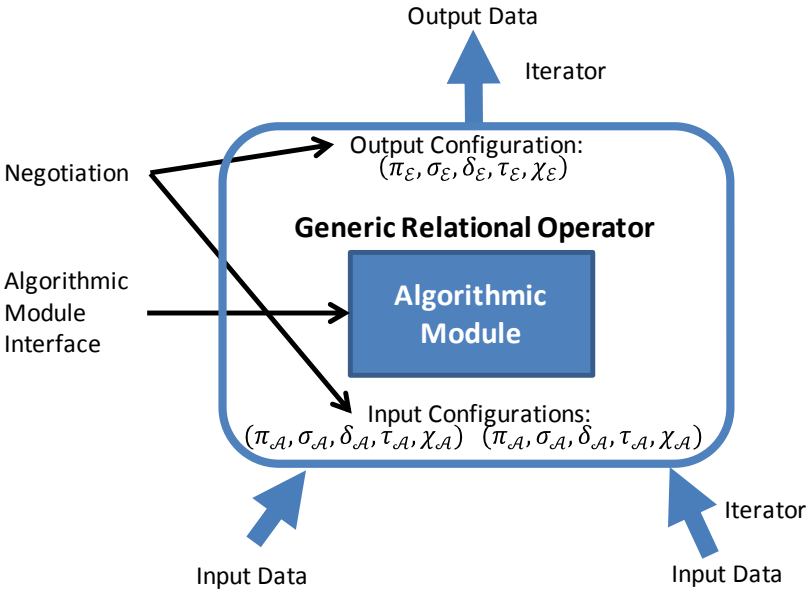


Figure.37 Generic Relational Operators. The generic relational operator encapsulates an extensive functional diversity of access module implementations. It is configured via negotiation for optimal interopera-

bility within a query plan and the resulting configuration represents the detailed description of its planned evaluation. During query evaluation, the algorithmic module's Iterator-based interface routines are operated by the generic relational operator in accordance to this configuration.

Owing to the congruence of algorithmic module interface and generic relational operator interface, the functionality of the generic relational operator embedding an algorithmic module is relatively simple. It has to manage and preserve negotiated configuration settings of the algorithmic unit, forward all Iterator and negotiation calls to the embedded module and contribute all necessary parameters for these calls. Access method interface and algorithmic module interface are operated according to a common protocol. In the following, we will provide separate surveys of the subtle differences between access methods and generic operators during iteration and negotiation.

4.4.1. Iteration

Although the interface routines names for iteration on access method interface and relational operator interface are identical, the limited functionality of the latter allows omitting parameters that are irrelevant for generic relational operators. The remaining interface is specified as:

`Open(`$\left(\mathcal{C}_{\mathcal{A}_i}\right)_{i=1,\ldots,n}$**`,`** $\mathcal{C}_{\mathcal{E}}$**`)`** → **`OpContext`**

`Next(OpContext)` → **`OutTup`**

`Close(OpContext)` → $\emptyset$

`Reset(OpContext,` $\mathcal{C}_{\mathcal{E}}$**`)`** → $\emptyset$

The functionality of the individual routines is very similar to their counterparts in the access method interface (cf. *4.3.4 Opening an Access Path* on page 128). The **`Open()`** routine of an algorithmic module creates a corresponding algorithmic entity associated with the resulting **`OpContext`** and prepares it for evaluation. The configuration parameters $\mathcal{C}_{\mathcal{A}_i}$ and $\mathcal{C}_{\mathcal{E}}$ are optional. Omitting configuration parameters allows opening an algorithmic unit for negotiation. If valid configurations are available from a prior negotiation procedure, then they may be used for opening an algorithmic unit and immediately presetting all input streams of an arbitrary *n*-ary ($n \geq 1$) relational operator and its single output stream. As its result, this routine returns the context of an operator in **`OpContext`**, a private data structure similar to the access method's **`ScanContext`**. This structure preserves an operator's internal state between separate interface calls. In contrast to the corresponding access method routine, a relational operator does not require a **`TaID`** as reference to a surrounding transaction, as it is never

confronted with concurrency issues. It always operates on its private input relation, uses private internal resources, and produces output dedicated to one single subsequent algorithmic unit. Also the parameters **DescPage** and **mode** are irrelevant and therefore absent, as operators possess neither persistent descriptions, nor alternative operating modes. The **Next()** routine is used for iteration, having the **OpContext** as its sole input parameter. The routine iteratively computes and returns the next result tuple as **OutTup**. After the last valid result was delivered, further calls to **Next()** will return an empty tuple, denoting end-of-data and making a separate descriptive result parameter similar to the access method's **ScanStatus** obsolete. Finally, **Close()** terminates the evaluation of an algorithmic unit and releases all occupied resources, thereby invalidating the contents of its input parameter **OpContext**. The optional **Reset()** routine functions similarly as for access methods. It is used for reinstalling alternative configurations $\mathcal{C}_{\mathcal{E}}$ as an inexpensive replacement for **Close()** / **Open()** pairs, while the algorithmic unit remains within the same logical operation. This is particularly useful if the algorithmic unit accepted a correlated predicate as exploitable configuration during negotiation. Note that reconfiguration via **Reset()** is limited to $\mathcal{C}_{\mathcal{E}}$, while $\mathcal{C}_{\mathcal{A}_i}$ is invariant while the operator is active. In contrast, the access method interface's **Reset()** routine also accepts alternative $\mathcal{C}_{\mathcal{A}_i}$ settings when operated in conjunction with the optional **Suspend()** routine**.**

4.4.2. Negotiation

Interface definition and protocol for negotiation in the algorithmic module interface resemble negotiation of access methods very closely. The interface specifications can be derived from each other by replacing the access method's dedicated **ScanContext** with the algorithmic module's generic **OpContext**:

Apply(OpContext, $\boldsymbol{\pi^{in}}$**, opt)** → $\left(\mathcal{C}_{\mathcal{A}_i}\right)_{i=1,\ldots,n}$

  opt ::= 0,1,2,3,...

Exploit(OpContext, $\mathcal{C}_{\mathcal{A}}$**)** → $\mathcal{C}_{\mathcal{R}}$

Propagate(OpContext, $\mathcal{C}_{\mathcal{A}}$**)** → $\mathcal{C}_{\mathcal{R}}$

Cost(OpContext, InCosts, Stats) → **OutCosts**

Otherwise negotiation for access methods and generic algorithmic modules behaves identical, enabling the query optimizer to handle both entities in the same way. For details on negotia-

tion, please refer to the corresponding section on access methods (*4.3.5 Negotiation and Optimization*, page 132).

4.5. Advanced Query Evaluation Techniques

Database management systems often operate in high-end hardware environments, capable of massive parallel computation and attached to broadband storage appliances. And, although an ideal extensible DBMS architecture should avoid any hardware specific considerations in its extension modules, the demand for maximized performance and thorough exploitation of available hardware resources makes it impossible to ignore them completely. Hence, this chapter will focus on suitable strategies for improving interaction between access modules and other custom algorithmic implementations with the host system's query evaluation engine and storage layer for gaining additional performance benefits. We examine three possibilities for effectively speeding up operations by exploiting general properties of modern computer hardware. Prefetching and partitioning are two techniques that incorporate considerations on typical performance characteristics of common storage technology into query processing. The third aspect of advanced query evaluation techniques examines scalability of DBMS throughput by utilization of parallel computation capabilities.

4.5.1. Prefetching

Despite rapidly growing main memory sizes and emerging alternative storage technologies, such as solid state drives (SDD), conventional hard drives are still the prevalent storage facility for persistent databases of considerable size. They represent a mature technology with many advantageous properties but they also have some noteworthy drawbacks. One of their most prominent properties is an asymmetry which is strongly favoring sequential I/O over random I/O. DBMSs have to actively counter this effect by employing adequate I/O strategies. Additional countermeasures have been developed on the hardware side. RAID (Redundant Array of Independent Disks) appliances are nowadays commonplace technology in computer systems running database servers. Their ability to distribute massive I/O workloads on conventional disks working in parallel is capable of achieving a higher throughput, even for random I/Os. However, the original problem is not fully overcome, and sequential I/Os remain substantially faster on RAID systems. In addition, RAID technology demands balanced utilization of all disks in the array, in order to operate effectively, such that the configuration of employed hardware again influences the behavior of the DBMS. SSD (Solid State Drive) technology for random access on secondary memory is advancing quickly in the

DBMS field. But the considerable higher price still makes them unattractive for large databases. In addition, the SSD technology introduces a new asymmetry, favoring reading operations over writing, another hardware property to be a considered in DBMS operations. The most important impact of hardware characteristics on DBMS performance is caused by the divergence of rapidly gaining CPU power and stagnating storage throughput. While disk performance literally remains on the same level since decades, additional CPU power was made available in accordance to Moore's Law. The divergence is not only limited to secondary storage in form of hard drives, but also main memory is affected. The establishment of multi-level hierarchies of hardware and software caches is clearly supporting this assumption.

Although poor I/O throughput is triggered by various reasons, originating from a diversity of employed storage technology, they eventually all lead to the same general effect, namely a memory stall, where the CPU cannot proceed with its tasks, because I/O operations on the diverse memory levels has not yet completed. Memory stalls can be effectively countered with *prefetching* and *write-ahead* techniques. Both techniques are based on the same fundamental premise. They try to request necessary I/O operations as early as possible, instead of deferring them to the latest point in time. This gives the storage system the opportunity to reorder requests within the resulting time interval and schedule them interspersed with requests from other tasks for achieving optimal I/O hardware utilization. Moreover, these techniques allow overlapping of I/O and computational efforts, leading to a superior overall system utilization.

An access module may implement effective prefetching by means of the storage layer's **`FixPage()`** routine. We already discussed how this routine is used during regular operations, where it is attributed with the corresponding lock type for the intended operation (**`NO_LOCK`**, **`READ_LOCK`**, **`WRITE_LOCK`**, **`EXCLUSIVE_LOCK`**). For enabling prefetching, we introduce **`PREFETCH`** as a fifth attribute. This prefetch mechanism of the storage layer will initiate transportation of the page with the requested page number (**`PNO`**) into main memory. As soon as the transport is initiated, the routine will return immediately, allowing the access method to pursue other tasks, before it will eventually access the requested page. Despite the routine's name, the page is not fixed in main memory and no locks are acquired in this particular mode. As soon as the page arrives in main memory, it is released into the system cache's LRU stack. If the requested page is already resident in the system cache when the prefetch is requested, then prefetch call will merely move that page on top of the LRU the stack, into MRU position. In any case, the requested page is henceforth subjected to the cache's normal LRU strategy,

and it begins to sink into the LRU stack while other pages are accessed. But during this period it may be inexpensively fixed by the scan operation that issued the original prefetch request, or by any other scan operation. The loose coupling of prefetching and fixing of pages via the system cache demonstrates the asynchronous nature of the prefetch mechanism. The page fix may be effectively issued within the time period before the page drops out of the cache. If the page is not fixed in a timely manner, the page was prefetched in vain, and the cache frame will be reused for another page.

The presented strategy raises several issues to be considered. The utilization of limited cache resources for buffering prefetched pages must happen in a cooperative way, because without further precautions the cache might be flooded with prefetch requests, making it unusable for its primary purpose. This is prevented by the storage layer that will actively limit the number of prefetched pages in the system. As a consequence, any prefetch request may be declined by the host system's storage layer. This again raises the issue of starvation, i.e. the host system must ensure that prefetching resources are shared in a fair way among concurrent scan operations. Finally, the access method itself must make prefetches in such way that it has a realistic chance that all requested pages are fixed before they drop out of cache. On the other hand, it must make sure that pages are not fixed too early, since a premature fix operation will block until the I/O is completed. Hence, every access method must devise a prefetching strategy that is actively balancing accepted prefetch calls of the storage layer and its own progress in accessing and processing of pages.

The corresponding mechanism for asynchronous write-ahead is also integrated into the storage layer's page access facilities. Whenever a data manipulation scan unfixes a modified page, this page is released into the LRU stack. As soon as this happens, the page is beyond control of the access method, and the page becomes a candidate for an immediate write operation. The host system's cache manager will constantly analyze the current LRU situation, in search of modified pages. For these pages it will initiate required I/O operations generating necessary log records and eventually it will write modified pages through to disk. On some occasions, a modification scan might return later to the same page for conducting another modification. If the first modification was already written to disk, the second modification will necessitate a second write operation. To prevent this form of page thrashing, the host system will employ a lazy write-through strategy, allowing scans to perform several modifications before the systems attempts to write to disk. Ideally, a write operation on a modified page is issued such that the write operation completes approximately at the time

when page reaches the bottom of the LRU stack, making such a page immediately replaceable.

Aside from these considerations, an access method that finds that a modified page should remain in cache, because the page is likely to be modified again, may either fix that page, which prevents its replacement altogether but binds cache resources. Alternatively, the access method may periodically issue prefetch calls on such pages, in order to keep them inside the upper part of the LRU stack, thereby preventing it from becoming a candidate for lazy write-through. Only when a transaction is about to be committed, all its manipulated pages that are still on cache *must* be written through to disk (more precisely, writing the logging information suffices) in preparation of the transaction's termination.

Hence, the host system is able to provide powerful prefetching and write-ahead facilities through the already established interface routines. Any access method may freely choose whether to employ these facilities for possible performance improvements. Moreover, any access method is free to make additional arrangements to improve its throughput by employing prefetching on other levels of the memory system hierarchy. As an example, [Che01] presents such considerations for cache-conscious B-trees. This form of prefetching has to be contemplated within access method implementations, and therefore it is beyond the scope of the access manager framework.

4.5.2. Data partitioning

Apart from prefetching, data partitioning is a commonly used technique for enabling advanced query evaluation and exerting beneficial influence on DBMS I/O and query processing performance. Data partitioning is used for two general purposes. Firstly, data may be distributed across independent storage and processing devices, in order to achieve throughput in an order of magnitude that is a multiple of the maximum throughput of one individual device. Alternatively, related data may be concentrated on dedicated sites, which is particularly useful for spatially distributing data of regional relevance, while allowing location transparent access to global data via federated or distributed DBMSs. Partitioning may be effectively combined with redundancy, either for further promoting efficiency, or for improved system availability and data security, or even for a combination of these two goals.

There exist a broad range of possible strategies for data partitioning. Relations may be *horizontally* partitioned, i.e. individual rows of one relation are kept in different storage areas. Alternatively, relations may be partitioned *vertically*, where projections of a relation are

stored in different places. It is also possible to combine these two partitioning techniques. The actual partitioning may be conducted according to various criteria. Data-driven partitioning is a common technique, where data satisfying a given predicate is grouped together. Typical predicates for this purpose are value ranges, value lists, hash functions, and compositions of the aforementioned, factorizing data into individual partitions, where each partition is assigned to a dedicated storage area. Data-driven partitioning may be conveniently exploited by the query plan optimizer when evaluating predicates that are similar to those used in data partitioning. Alternatively, data may be partitioned implicitly and independently from its actual datum, for example via random or round-robin assignment to different storage areas. Such *data striping* cannot be exploited directly when processing predicates in query evaluation, but partitioning may facilitate parallel query evaluation techniques, e.g. for generating partial query results on independent sites. In contrast to data-driven partitioning, data striping may also be conducted on the hardware-level, e.g. by RAID appliances. Finally, all forms of partitioning, namely horizontal, vertical, data-driven, and data-independent, may be arbitrarily combined, for achieving a desired overall system behavior of joint performance, availability, and security characteristics.

Data partitioning within an access path is controlled to a large extent via the SQL `CREATE` statement's `PARTITION BY` clause. The contents of `PARTITION BY` will be recorded within the system's data dictionary, where they are supportive for the query plan optimizer during query planning, in particular for effective exploitation of data-driven partitioning in predicate evaluation. Partition specification may contain logical names of storage areas (often referred to as *table space* or *data space*) or physical storage areas, e.g. extents, files, etc. Further partitioning directives may be provided by an access method's dedicated `custom_spec` clause (cf. *4.3.2 Access Path Creation* on page 120). Hence, both partitioning specification are available at access path creation time and the access method will integrate them into its description page, where there are permanently available as guidance to be used for conducting data manipulation accordingly.

On the low abstraction level of practical access method implementations, partitioned storage areas are expressed as paired page numbers, denoting the lower and upper boundaries of intervals on the database's physical address space. These intervals correspond to the logical storage areas of the `PARTITION BY` clause. Under these preconditions, partitioning from the perspective of an access module is a fairly easy task that is intrinsically tied to page allocation strategies. After the affiliation of data to a particular partition is determined, the access me-

thod may systematically acquire necessary storage area within the dedicated partition through the storage layer's **`AllocPage()`** interface. In section *4.2.1 Storage* of the built-in storage layer (page 105), we already discussed how this routine allows selective memory allocation, which is employed for preserving physical clustering (*4.3.7 Data Manipulation*, page 143) and for reorganization (*4.3.13 Administrative Tasks*, page 159). Partitioning merely represents another application of this versatile mechanism.

4.5.3. Parallel Query Processing

Until recently, all technical advancements for providing additional computational power were achieved through higher integration of circuits, more sophisticated processing techniques, and higher clock rates of single core CPUs. All existing applications were automatically benefiting from these improvements, without the necessity of any adaptations on their parts. In future, this is likely to change, since fundamental technical limitations inhibit further advancement at present rate in this direction. As an alternative, additional computational resources are likely to be provided in form of multiple CPU cores. This significant change of the basic technical principles of computational hardware profoundly affects all performance-critical software to be run on such systems. The conception of a DBMS as a monolithic service, transparently providing scalable performance, inevitably leads to the conclusion that parallelism has to become an integral part of DBMS design in general and its relational algorithms in particular. In the following, we will present how the Access Manager framework provides all necessary precautions for supporting parallelism.

Conceptually, we distinguish three forms of parallelism, namely *intra-query* parallelism, *inter-query* parallelism, and *inter-transaction* parallelism. The latter two items are very common in multi-user DBMS technology, thus we will not elaborate on these topics and concentrate on intra-query parallelism. This concept is again divided into *intra-operator* parallelism, denoting parallel execution of one particular operator like parallel sorting, and inter-operator parallelism, which describes functional decomposition of a query plan into tasks that may be executed independently. In our conception, these concepts do not operate on granularity of individual relational operators, but rather on granularity of the algorithmic entities that implement these operators.

Transbase, as the host system for the Access Manager prototype, provides parallelism in query processing transparently on the granularity of individual algorithmic units in a query evaluation plan. Parallelism is encapsulated into one single operator, the *async* operator

[Ack08]. This operator has no analogon in relational algebra and serves merely for optimizing resource utilization and improving performance through parallelization. It provides a data buffer, capable of temporarily retaining a small portion of its input data, and an asynchronous thread of execution. This thread is responsible for evaluating the tree portion below the *async* node, which functions as a *producer* in the classical producer/ consumer pattern. Its task is to deliver input data and store it into the *async* node's intermediate buffer. The operator tree above the *async* node runs independently in the parent thread of execution, representing the consumer, which simultaneously retrieves data from the buffer. Naturally, access to the buffer is controlled by synchronization primitives that are also integral parts of the *async* operator. In correspondence to the well-known producer/ consumer scheme, the consumer thread will block until data is available in the buffer, while the producer thread will block whenever the buffer is full. When the lower part of the operator tree is exhausted, i.e. the last tuple was delivered, the producer thread terminates. Figure.38 illustrates possible scenarios for employing the *async* operator.

Parallelization of sequential query execution plans using *async* operators is an integral part of the query planning process and therefore lies in the responsibility of the query optimizer. The optimizer identifies operator tree portions that are suitable for parallelization and separates them from the main thread of execution by inserting an *async* operator. Therefore it examines the possible impact of parallelization by evaluating the `Cost()` functions. But the optimizer is also responsible for managing the costs induced by parallelization, i.e. synchronization, memory consumption for *async* buffers, and overhead for copying tuples into these buffers for exchanging data at thread boundaries. These costs effectively limiting the number of threads in the system. In consequence, query optimization will try to minimize the number of thread boundaries while maximizing the number of parallel operations within a query plan. In addition to efficiency, the optimizer must guarantee that algebraic soundness and algorithmic integrity of the query plan are not compromised by its parallelization efforts. Separated tree portions are then executed independently from the main thread, thereby effectively accomplishing inter-operator parallelism. In addition, the optimizer identifies operator tree portions that represent pipelines of parallelizable operators. Initially these pipelines run independently in one dedicated thread, achieving mere inter-operator parallelism. But they may be replicated dynamically during query evaluation, and a new thread is spawned for every copy, thereby providing true intra-operator parallelism. The pipelines' input stream is partitioned among all existing pipeline instances. This partitioning introduces the necessity to consider order preservation of data within pipelines, in order to comply with present input directives of successive

algorithmic units. *Asyncs* are capable of enforcing order-disrupting, order-preserving, or order-establishing data flow within a pipeline, were each mode is attributed with different performance characteristics. The query optimizer guarantees validity of input directives and enforces them by adjusting *asyncs* to suitable operating modes. Pipelining provides intra-operator parallelism, and its dynamics are able to eliminate congestions in a query plan by adding processing capacities in form of additional pipelines as required. The objective of these dynamics is to achieve a balance among all consumers and producers in a query plan, effectuating load-balancing and self-tuning capabilities that ultimately strive for optimal resource utilization and maximized throughput.

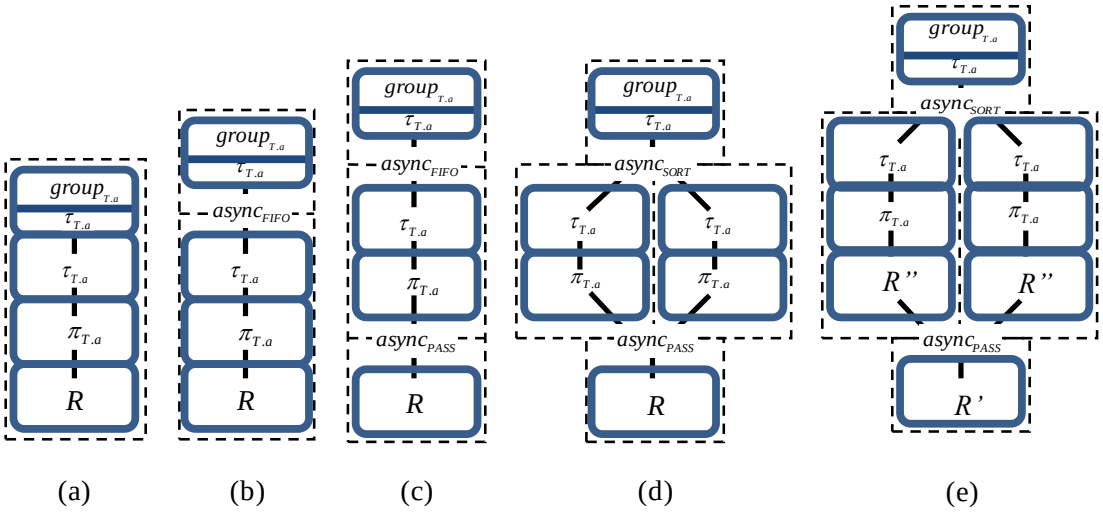


Figure.38 Parallel query execution in Transbase. A sequential query evaluation plan (a) is parallelized by insertion of *async* nodes between algorithmic units, creating new thread boundaries (dashed boxes). All communication between separate threads is conducted solely via the host system's *async* operator. Partitioning of tree portions into threads is conducted such that each algorithmic unit is distinctly assigned to one single thread of execution, thereby effectuating inter-operator parallelism (b, c). Replication of parallelizable plan fragments (pipelines) allows intra-operator parallelism (d). In this case, the *async* operator is responsible for partitioning input data among pipelines and for reintegration of pipeline output. For facilitating preservation of input directives, the *async* node is capable of operating in various modes (PASS/FIFO/HASH/SORT), each influencing order preservation and performance characteristics differently. (e) demonstrates how a relational scan on *R* is divided into two separate algorithmic units *R'* and *R''*, where *R'* represent the sequential portion of the scan module's implementation, while *R''* comprises that part of the code that may be used in pipelines.

Custom algorithmic implementations and custom access modules may participate in these parallelization mechanisms, if they comply with a few basic requirements. The most important property of any algorithmic unit is *thread-safety*, i.e. every algorithmic implementation must assume that it is executed in multiple parallel incarnations within one or more query

plans and that the host system will take no synchronization precautions of its own. Moreover, every module must operate *independently* from neighboring entities and in a fully *self-contained* way, as the optimizer will assume that inter-operator parallelism is generally possible. Only the presence of compelling indicators prohibiting parallelization, such as correlated predicates inducing non-local relationships between operators across thread boundaries, force the optimizer to refrain from parallelization. Otherwise, the optimizer is free to introduce thread-boundaries between algorithmic units as it sees fit. In consequence, it is illegal to maintain any cross-relations between individual algorithmic units. Intra-operator parallelism, on the other hand, is not applied to an algorithmic unit without obtaining the unit's explicit approval via its `Cost()` function. We already indicated in *2.4.6 Cost Function* (page 58) how an algorithmic unit will communicate its *ability to scale* when executed on parallel hardware, by revealing a ratio of parallelizable code portions as compared to its total sequential expenditure via its `Cost()` function. This allows the query optimizer to estimate the algorithmic unit's performance in parallel execution by means of Amdahl's Law. If an algorithmic unit is incapable of parallel execution, either because its algebraic equivalent prohibits parallel processing, or due to an inadequate implementation, it will express this property by returning a parallel cost indicator that will prevent parallelization. It is important to note that the `Cost()` function is always applied to a completely configured algorithmic unit, therefore the resulting cost estimation will include any impact of the configuration on the unit's attainable degree of parallelism.

Figure.38 above also demonstrates one important peculiarity of this approach towards parallelization. Since algorithmic units are strictly assigned to one thread of execution, it is impossible to separate sequential and parallelizable parts of one single algorithmic entity for improving scalability. This limitation is avoided by explicitly allowing any algebraic expression (this explicitly includes primitive relational operators) to be implemented as a composition of autonomous algorithmic entities. Figure.38(e) exemplifies this on a relational scan on *R*, which is composed of two algorithmic entities *R'* and *R"*. Both entities are participating separately in the negotiation process, thereby obtaining the possibility to express differing scaling abilities. In this example, *R'* will use sequential code for performing necessary I/O operations. The consecutive *async* operator will then partition the retrieved data among various instances of *R"*, which represents the parallelizable part of the original scan operator *R*. *R"* will conduct mostly computational work, e.g. decompression, extraction of tuples, or possibly evaluation of predicates that were absorbed via exploitation. This concept may be conveniently combined with prefetching, where *R'* will issue asynchronous I/O requests.

Instead of partitioning retrieved data, prefetch requests are distributed among the pipelines, such that *R''* instances are able to wait for I/O competition and subsequently commence processing.

4.6. Data Integration

Data integration serves as an instrument for incorporating data into a logical database schema, which is physically stored outside the built-in storage layer of the host DBMS. We will present two general approaches to data integration using the Access Manager framework and we will evaluate implications and opportunities arising from each. Both approaches are direct consequences of the Access Manager's primary interface layers, one using the high-level tuple-oriented access method interface while the other is based on the low-level page oriented storage layer interface.

4.6.1. Alternative Storage

The necessity to publish the host system's storage layer interface (Figure.22, page 102) as a basis for building access methods also offers the opportunity to use it for an alternative purpose. If a storage layer is considered as a pluggable software module, similar to the dynamically loadable libraries representing implementations of access methods and relational operations, then it becomes immediately clear that the mere existence of the storage layer interface definition allows to apply a similar approach for devising a concept for pluggable custom storage components, providing page oriented storage facilities that are able to enhance the technical capabilities of the overall system. Similar to the definition of a custom `TABLETYPE` or `INDEXTYPE`, (*4.3.1 Data Access Module Definition* on page 119), it becomes possible to integrate storage module implementations in form of dynamically loadable libraries. A minor SQL extension will allow introduction of such modules into a database schema.

```
CREATE STORAGETYPE <stype_name> FROM <file>
```

Similar to the corresponding access module definitions, this DDL statement extends the schema of one particular database, rather than globally extending the DBMS. Its purpose is to provide the location of the dynamic library implementing a custom storage layer. It also assigns a unique name (`stype_name`) to be recorded in the database's data dictionary for referencing this storage module in subsequent DDL statements. Such references to `stype_name` are used when creating new segments, e.g.

```
CREATE TABLE <tname> <table_definition>
  USING STORAGETYPE <stype_name>[(custom_spec)]
```

This statement will cause the Access Manager framework to redirect all storage allocation requests issued for the segment known as `<tname>` to the storage module referenced by `<stype_name>`. This in particular includes the allocation of the new segment's description page, hence the new segment is completely allocated in external storage, while the database's data dictionary stores necessary information for accessing it. The purpose of the optional `custom_spec` clause is similar as in the custom access path definition. It specifies additional information to be parsed and processed by the storage module, e.g. a file name, connection string, space allocation directives etc.

The inversion of this process is also required. Its purpose is the removal of a storage module from the database schema. This is accomplished by deleting the corresponding entries from the system catalog, after verifying that there are no active references, i.e. there must not exist any access paths based on the storage module to be removed. The corresponding DDL extension has the form:

```
DROP STORAGETYPE <stype_name>
```

The storage layer interface (Figure.22 on page 102) does not provide the same flexibility as the access method interface with its extensive optional interface routines. All routines, apart from the optional savepoint feature, are mandatory. This rigidness maximizes the compatibility and reusability among different storage layer implementations as exchangeable foundations for access method implementations, and it is owed to the completeness of functionality an access module expects from a storage layer. We demonstrated how an access module will rely on caching, multi-version concurrency control, locking, logging & recovery, savepoints, etc., to be implemented in the storage layer, thereby reducing the implementation complexity of the access module. If a custom storage layer is to be used as an alternative to the built-in storage layer, both layers will have to implement congruent functionality. However, the rigidness of the access method interface does not imply that storage layers with limited capabilities are prohibited. It is still possible to implement storage layers that operate in read-only mode, without transactional isolation, without caching, (see also *3.2.5 MySQL* on page 87 for practical examples). But to this end, it is necessary to implement routine stubs for missing functionality that return error diagnostics when called, such that incompatibilities between storage layer and access methods are detected reliably. This will ensure that a func-

tionally limited storage layer will only cooperate with an access module implementation that is aware of these limitations.

Apart from implementing complete replacements for the built-in storage layer, the storage layer interface offers another interesting opportunity. Similar to the concept of *intermediate access modules* (Figure.20 on page 97), it also becomes possible to stack storage layer implementations, for combining their functionalities. As an example, imagine a thin *intermediate storage layer*, which is based on the functionally complete built-in storage layer, but additionally provides custom encryption or compression capabilities on page level.

But given the immense functional complexity of storage layer implementations, we expect that such endeavors are only of little practical relevance for data integration, in contrast to data integration layers that are based on the tuple oriented access method interface.

4.6.2. Data Integration Layer

A data integration layer (cf. Figure.20 on page 97 for a classification in the overall system architecture) is a software artifact that is capable of attaching an external data repository to a database schema that is operated by a host DBMS housing the Access Manager framework. Its main purpose is to bridge any discrepancies between the host system's and an external data source's conception of data. These versatile discrepancies may involve incompatibilities of data models and data representation, geographical distance, connectivity and authentication, etc. Hence, data integration layers function as data access drivers, each dedicated to one particular external data source, for providing a coherent, location transparent view on a collection of remote data. The Access Manager allows extending its host system with such modules, if they implement the access method interface. Consequentially, a data integration module acts as any other access module, providing navigational and optionally manipulation capabilities on data in its responsibility. The flexibility of the access method interface is applied for tackling the possibly broad functional diversity of external data sources and subsuming them in one comprehensible and manageable interface. Ultimately, external data is integrated into the database schema such that the host system may access it as any other ordinary relation though the access module interface. All implementation details are hidden within the integration module controlling the access to that relation.

This encapsulation allows handling integration layers in exactly the same way as an access module. An integration layer manifests itself as a dynamically loadable library, which is used for extending the host system, similar to any other plug-in component implementing an access

module (cf. *4.3.1 Data Access Module Definition*, page 119). Subsequently, an access path to remote data may be created, using the standard routines as described in *4.3.2 Access Path Creation* (page 120). This new access path will be represented by corresponding descriptive entries in the database data dictionary, as a complete specification of the remote data as a table, providing the host system with a *relational view* on this data, regardless of the data source's actual data model. These data dictionary entries are established under the transactional context of the DDL statement, creating a *permanent link* to external data in the database schema. Moreover, the new relation will be associated with a segment located in the database's internal storage area, consisting at least of the segment's description page. The integration module is free to allocate additional pages as required, as a storage area serves for persistently archiving all necessary information for operating on the external data source, e.g. connection information, credentials, data description, etc. The actual data however will typically remain in the remote repository.

After creating such an access path, external data becomes accessible via SQL by referencing the relation's name in a database query. The SQL compiler is able to resolve the name via the system catalogue and associate it with its relational table definition. Eventually a scan is opened on the remote data. Depending on the capabilities of the integration module's implementation and on the availability of optional interface routines, this scan may be used for read-only access, but also for data manipulation. As the host system does not discern between data integration layers and other access modules, it becomes possible to improve access to external data by creating auxiliary internal or external access paths.

As any data integration module implements all assets for negotiation, access to external data will participate in the host system optimizer's global optimization process. As a consequence, it becomes possible to relocate necessary transformations to the remote repository, if that system is capable of conducting relational operations. Such measures are suitable for achieving dramatically improved performance in query evaluation. For example, relocation of restrictions and projections to the remote system is able to minimize the data transfer volume between external repository and host system. Exploitation of pre-sortedness will reduce computational costs and temporal storage requirements. Cost-based query optimization is guided by assessments made by the integration module's `Cost()` function, allowing the optimizer to deal reliably with index selection and join optimization challenges. Index selection chooses one or more access paths from multiple alternatives, considers index intersections, index unifications, and ultimately materialization if necessary. It operates globally, and

evaluates all access path candidates that are registered in the central data dictionary, hence it becomes possible to distribute secondary access paths to the one relation arbitrarily among all attached repositories and the built-in storage facility. Join optimization covers cost-guided selection from arbitrary join sequences for combining data from diverse external repositories and internally stored data. Hence, this approach to data integration allows internally and externally stored data to participate equally in the query optimization process.

But despite these expedient properties of data integration layers, one major shortcoming remains. Limitation of exploitation to the set of *basic unary operators* $\{ \pi, \sigma, \delta, \tau \}$ prevents propagation of more advanced relational operators to remote sites. As a consequence, joins between two relations residing in the same repository must be conducted by the coordinating host system, possibly provoking excessive and expensive shipment of data. Other relational operations, like grouping and aggregation, capable of dramatically reducing the data transfer volume, cannot be applied directly at the remote site. This fact is particularly unsatisfactory, if the remote data repository happens to be a full-blown RDBMS, which would be perfectly capable of handling such complex relational transformations. This limitation represents a considerable handicap of the Access Manager's integration layer compared to some dedicated data integration architectures, which claim to accomplish this task.

Our approach to circumvent this shortcoming capitalizes on the RDBMSs' general ability to rewrite queries, a prerequisite for the prevalent *materialized view* concept. However, we do not capitalize on pre-calculation aspects of materialized views, but on mere query rewriting. As an example, let *A, B* be two relations that are located in an external data repository that is also capable of performing a relational join operation. If we define a relation *AB* as a *remote view* on the result of $A \times B$, and make *AB* accessible via the integration layer, then the host system's query optimizer may choose for arbitrary queries joining relations *A* and *B*, whether to perform the join locally after transferring necessary data from the remote repository, or to access *AB* instead. In the latter case, the join is actually performed on the remote system. This is further supported by the Access Manager's negotiation process, which is admitting relocation of arbitrary predicates to the remote site. As this in particular includes relocation of join predicates from the original user query, this universal approach is capable of effectively reducing the join result size and thereby minimizing communication overhead for arbitrary ad-hoc queries.

The remote view concept may be extended to other relational operations, including predefined aggregations in a remote view definition. But in contrast to the generic Cartesian

product that is effectively refined by adding appropriate predicates, handling of aggregations is restricted to the pre-defined grouping granularity from the remote view definition. Although this may suffice for supporting a known set of aggregation queries through definition of convenient remote views, this concept does not possess to necessary flexibility to cope with ad-hoc analytic queries.

5. Proof of Concept

In this chapter, we will present a collection of plug-in implementations for extending the Transbase RDBMS, which serves as the host system for the Access Manager framework prototype. This survey will exemplify the framework's capabilities for integrating supplemental modules for data retrieval, data storage and data integration into an operational core system. Some of the presented plug-ins are mature access module implementations, which were originally built-in Transbase components, and now have been extracted, modularized and adapted to the Access Manager framework, thereby making them available for interoperability with future plug-in implementations. Based on this reliable foundation, we developed several new access modules, each exhibiting fundamentally different behavior compared to Transbase's innate access methods, yet reuse of aforementioned well-tested components helped significantly accelerating their development process, such that the new plug-ins quickly reached a maturity comparable to their building blocks. Finally, other access methods have been implemented completely from scratch, enriching Transbase with supplemental access and storage technology. Some of these new implementations soon reached industrial strength and, in fact, have already become part of the Transbase product, while others remain in the stage of design concepts for fathoming the possibilities of the Access Manager framework. All presented *primary* access methods generally permit combination with arbitrary *secondary* indexes, and vice versa, even if full support is not implemented in every case. We will provide an overview of interoperability of primary and secondary access paths when introducing the diverse access module implementations.

The concepts of non-standard data models and abstract data types (ADTs) are omitted in this chapter, and we concentrate on basic operations on standard relational data. Since non-standard data has to be transformed into the relational model eventually, in order to be processed with the standard relational operators provided by the host RDBMS, this omission does not compromise the universality of the following study. The technical details of data conversion between different data models are also beyond the scope of this work, but we will discuss conversion of data between standard and non-standard representation within the relational model.

The integration of custom relational operators, which is the secondary objective of the Access Manager approach, will demonstrate tight coupling between access methods and supportive consecutive transformations. We will present how the Access Manager framework will lend

itself to integration of sophisticated solutions for user-defined functions, predicates, aggregates, and other operators, tailored for supporting a specific access method or for specialization of the universal host RDBMS towards a certain application domain.

5.1. Transbase Prototype

The Access Manager prototype was integrated into the commercial RDBMS Transbase. Before discussing the details of its implementation, we will provide a short overview over its designated host system.

5.1.1. The Transbase RDBMS

Transbase [Tra10] is a thriving commercial RDBMS, providing dependable and efficient standard DBMS functionality and is equipped with rich additional features. It implements the typical client-server architecture, and consequentially provides a broad variety of prevalent client APIs. It is compliant with the *ISO SQL Standard* [ANSI99] (SQL-2, entry level) and supports many SQL features which have been categorized as optional in SQL-2, SQL-3 and subsequent revisions. Moreover, it provides unrestricted support for the ACID paradigm in all DDL and DML operations. Transbase's key features are its compactness and high scalability, as its comparatively small footprint allows providing the complete spectrum of DBMS functionality at considerable performance, even under tight resource restrictions, while it is also able to effectively exploit extensive hardware configurations for attaining maximum efficiency. In addition, Transbase has a tradition of incorporating advanced indexing techniques. Its Hypercube index for OLAP, which is based on the UB-tree technology, is tailored for accessing multidimensional and hierarchical data in data warehouse applications. Still, like in most other RDBMSs, the B-tree [Bay72] is the prevalent access method in Transbase, and in fact, it was the only access method for almost two decades. Over this time period, many B-tree properties have established themselves in adjoining system components, like query planning, query evaluation, locking, logging, and recovery. Consequently, Transbase components often silently presumed B-tree peculiarities, e.g. the optimizer derives assumptions on clustering and sort order directly from the B-tree's key specification. For the subsequent integration of the UB-tree [Ram00], these assumptions posed considerable obstacles, making it necessary to explicitly distinguish between these two alternatives. Now, the integration of the Access Manager into Transbase provides an abstraction from the properties of particular access methods, allowing a clean separation of Transbase's two innately built-in access methods. A

more detailed discussion on necessary adaptation of these existing access methods for integration into the Access Manager framework will be provided shortly.

5.1.2. Limitations of the Prototype

The Access Manager prototype is implemented as an optional feature in Transbase. It can be deactivated via a configuration setting at compilation time. This allows switching back to the original implementation by recompilation of the Transbase system. Hence, an unmodified implementation of the original data access layer still exists within the system's code basis. It serves as reference implementation for evaluating functional and performance-relevant aspects in the behavior of the Access Manager implementation. In order to allow switching between these two implementations, the Access Manager interface is hidden behind a facade resembling the original access layer interface towards other system components. In particular, the query processor still calls the access layer's original interface routines. From this fact arise a number of minor limitations in the current state of the prototype implementation, the most important being its restrained negotiation and configuration capabilities. The optimizer assumes uniform *applicability* and *exploitability* for all access method types. These are derived from the system's data dictionary and apply to the original B-tree implementation, the prevalent access method in Transbase. In detail, this means that Transbase assumes that any access method type is able to produce arbitrary *projections* of the attributes associated with a relation in the data dictionary. For insertions and updates, the exact sequence of attributes from the original table definition (DDL) is assumed as de-facto input projection to be enforced by applicability directives. Secondly, Transbase assumes that *restrictions* may be efficiently exploited on the prefix of the primary key, as it is the case for B-trees. Transbase already implements one exception to this rule for UB-tree handling, where restrictions on all key attributes are evenly exploited. Thirdly, Transbase expects a relation's primary linearization to be identical to a lexicographical *sort order* on the relation's primary key attributes, also a property of the B-tree. This sort order is also used as applicable sort order for speeding up set-oriented data insertion and manipulation. Again, Transbase implements a specific exception to this rule for UB-trees, as those exhibit no exploitable sort order. All configuration aspects for exploiting *distinction* and *representation* are missing completely.

Although these limitations appear severe at first glance, they have surprisingly little impact on the evaluation of the prototype access methods described here. This is owed to the fact that all prototypes were implemented in awareness of these characteristics of Transbase. All access methods are equipped with the ability to produce arbitrary projections, they are capable of

effectively enforcing restrictions on their indexed attributes, and they expect the presence of a lexicographical sort order on input data streams controlling modifications. This makes negotiation of projection, selection, and sort order directives obsolete, as all configurations suggested by the host system are invariably accepted. Regarding alternative data representation, Transbase was adapted in selected places with minor modifications, providing required functionality for the introduction of bitmap representation. For instance, instead of implementing separate bitmapped versions of union and intersection operations on sets, the original tuple-oriented operators were adapted for supporting both representations. These algorithmic units choose dynamically between the available alternatives at query execution time, depending on the actual input representation as tuples or bitmaps. This is considered a temporary solution, which is owed to the fact that Transbase is in permanent productive use. Any extensive and fundamental modification, such as the introduction of the Access Manager framework, must be conducted gradually for preserving the functional integrity of the overall system. This workaround will be undone in favor of an implementation conforming to the presented negotiation approach, as soon as the original access layer facade becomes obsolete and the Access Manager framework is fully activated.

Hence, query evaluation of the prototype is fully operational, yet there remain deficiencies in the field of query optimization, where temporary modifications are too extensive and therefore not a viable alternative for a prototype implementation. In these cases, we compensate for missing autonomous query optimization capabilities by using Transbase's ability to process fully specified query execution plans in textual representation. With this measure, it becomes possible to circumvent all shortcomings of missing negotiation in the optimization phase, by predefining optimal configurations in 'handmade' query evaluation plans for measuring the full potential of the access method interface. Whenever limitations of the current preliminary implementation emerge in the following discussion of a specific access method implementation, we will indicate how the limitation was circumvented, together with an evaluation of possible implications of this workaround on the presented results. We want to emphasize, that these limitations will be resolved completely, as soon as the facade currently concealing the Access Manager interface is removed from the host system, and negotiation becomes fully available during query optimization.

5.1.3. Reference Database & System

In order to receive comparable results, we will conduct the following measurements on a uniform data basis. Therefore, we will start with a simple database schema, consisting of one single relation, as defined by the following SQL DDL statement:

```
CREATE TABLE R (a SMALLINT NOT NULL, b SMALLINT NOT NULL,
    c SMALLINT NOT NULL, d CHAR(80), PRIMARY KEY (a,b,c));
```

This table is populated with the reference data set using artificial data, where the three primary key attributes `a`, `b`, `c` are filled elements of the set $\{0, 2, 4, \dots, 508, 510\}^3$, i.e. the complete data set stores $256^3 = 16.777.216$ tuples. The textual field `d` serves as substitute for descriptive information associated with the respective key values. In total, every tuple has a constant payload of 86 bytes, and in Transbase's standard tuple representation each tuple amounts to a fixed size of 89 bytes, and consequently `R`'s total data volume is at least 1.5 Gbyte in standard representation. The sample database uses a page size of 32 kbyte, hence the relation will occupy roughly 45600 pages. We will continuously adapt this schema for incorporating the currently discussed data access method as a separate access path T_i, which is structurally identical to `R`, and load the reference data set. On this relation, we will then conduct a series of scan operations:

```
(a) SELECT COUNT(*) FROM Ti;
(b) SELECT COUNT(*) FROM Ti WHERE a BETWEEN 100 AND 152;
(c) SELECT COUNT(*) FROM Ti WHERE a BETWEEN 100 AND 256
        AND b BETWEEN 100 AND 152 AND c BETWEEN 100 AND 152;
(d) SELECT COUNT(*) FROM Ti WHERE a = 100 AND b = 100 AND c = 100;
```

These queries represent the following operations: (a) 100% full table scan (FTS), (b) 10% interval scan, (c) $30\% \times 10\% \times 10\% = 3‰$ query box, (d) point access. We use the inexpensive aggregation `COUNT(*)` for eliminating the potentially time-consuming transfer of result sets from the query's elapsed time. As of data manipulation, we conduct the following operations in the presented sequence:

```
(e) INSERT INTO S SELECT a, b+1, c+1, d FROM R
        WHERE a mod 6 = 0 AND b mod 4 = 0 AND c mod 4 = 0;
(f) INSERT INTO Ti SELECT * FROM S;
(g) DELETE FROM Ti WHERE a mod 6 = 0 AND b mod 4 = 0 AND c mod 4 = 0;
(h) UPDATE Ti SET b = b-1, c = c-1 WHERE b mod 2 = 1;
```

Statement (e) serves as a mere preparation, as it selects a 'scattered' subset of $33\% \times 50\% \times 50\% = 8{,}25\%$ from the original relation `R`, applies some minor transformations, and inserts it into a temporary relation `S` that will serve as source for the following insertion (f) into the target relation T_i. After conducting the complete sequence of manipulations on T_i, it will hold the same data as before the manipulation, but the access path will have been restructured into a different physical layout.

Of course, these sample queries are only suitable for a very coarse comparison of access methods, and the results cannot be easily transferred to arbitrary real-world database applications with genuine semantic on the stored data, non-uniform data distribution, and variable sized tuples. Nevertheless, they are appropriate for demonstrating the most prominent distinguishing properties of every access method under examination.

All measurements in the following sections were conducted on a Dell PowerEdge T610 server, running a SUSE Linux 2.6.31.5 x86_64 kernel. The system is equipped with 24 Intel Xeon 5650 CPUs operating at 2.67 GHz and 48 Gbyte RAM. Secondary storage is attached via a Dell Perc H700 SCSI RAID Controller, transferring 6 Gbit/s over 2x4 internal SAS connectors. It operates eight 160 Gbyte HDDs at 7200rpm in RAID-0 configuration with a total capacity of 12 Tbyte. The exact version of the employed Transbase database system is V6.9.1.1 (Build 555).

5.2. B-Trees

The B-tree [Bay72] (see also [Bay77a], and [Com79] for a survey), or more precisely the B^+-tree variant implemented in Transbase, is the prevalent data access structure in most DBMSs. It efficiently supports all common usage patterns of access paths in data retrieval and manipulation, namely full table scan, interval scan, point access, singleton data manipulation, positioned update and deletion, and mass-manipulation. Since the basic concepts of B-trees are commonly known, we will henceforth concentrate on the technical aspects of their implementation in Transbase.

The present B-tree implementation is capable of indexing one or more attributes of arbitrary SQL data types. We refer to this subset of indexed attributes as the B-tree's *search key*, which is potentially non-unique and therefore it must not be confused with unique key constraints in the relational sense. If the indexed attributes constitute the relation's primary key, then every occurring value combination must be unique, and the B-tree is capable of efficiently enforcing

this constraint. Alternatively, Transbase's B-tree implementation may also operate in duplicate mode using non-unique search keys. In addition to the search key, the B-tree may store additional information, which is associated to the search key, in auxiliary attributes of arbitrary types as non-key data elements. The internal nodes of the B-tree store separators consisting of prefixes of indexed attributes, stored in a compact representation similar to Prefix B-trees [Bay77a], and serving as orientation when accessing the tree structure via search key lookups. The leaf level contains plain relational data, consisting of index attributes and non-index attributes. Leaf pages also possess a forward chaining, allowing immediate sequential navigation on the leaf level. A sequential scan in reverse direction is also possible, but it will resort to the backwards chaining provided by the lowest internal node level, since an explicit reverse chaining on leaf level is not supplemented.

Although the B-tree technically guarantees a filling degree of 50%, practical experience shows that typical page utilization is over 80%, and space utilization is further enhanced by diverse packing techniques in this implementation. To this end, data is not compressed, but arranged in a compact internal representation, allowing efficient data access and predicate evaluation directly on this representation. Among other techniques, compactness is achieved through *unaligned* storage of data, although some machine data types (e.g. the *double* precision floating-point format) require specific alignment in memory on some hardware platforms, thereby introducing small gaps of unused memory in data structures representing relational tuples. Unalignment is conducted whenever a tuple is inserted into a page for persistent storage, while its inversion is applied to all tuples that are retrieved for query processing. This technique is beneficially combined with *attribute reordering*, i.e. B-trees store an internal projection of attributes that is different from the original table definition. Attribute reordering allows space-optimal rearrangement of attributes, such that a priori alignment is automatically achieved for efficient data access. Reordering also permits arrangement of fixed-sized attributes into continuous memory blocks, simplifying memory management and resulting in further reduction of storage requirements. Similar to head-compression in Prefix B-trees, a special *attribute suppression* mechanism exploits the B-tree's sort order, which forces tuples with common prefixes on indexed attributes onto one page (intra-page clustering). Attribute suppression eliminates the requirement to store redundant tuple prefixes repeatedly.

Data access is organized in concurrent scan operations on B-tree segments, where scans from different transactions are isolated implicitly by the storage layer. Locking of B-tree operations

is conducted on page level, offering a good trade-off between concurrency and locking complexity, but locking on relation level is also possible. Both variants use Transbase's intrinsic locking facility. Naturally, the B-tree implementation is built on Transbase's internal page-oriented storage layer, operating according to a multi-version concurrency protocol (MVCC). Utilization of lock manager and storage layer provide transactional atomicity, consistency, isolation, and durability to all B-tree operations. Consequentially, this B-tree implementation qualifies as a *full access method*, as it bridges the functional mismatch between the page-oriented storage layer and tuple-oriented query processing.

Concurrent B-tree scans belonging to the same transaction use the *scan maintenance* technique described earlier, for adapting the internal status of foreign scans that are influenced by data manipulation in the vicinity of their current scan position. In terms of *negotiable* evaluation techniques, B-tree scans offer full support for arbitrary attribute projections and predicate evaluation on a continuous prefix of the index attributes. The B-tree's lexicographical sort order is fully exploitable for sequential scans in forward and backward direction. Conversely, the B-tree is able to draw significant performance benefits during mass-manipulation from an adequately presorted input stream. Implicit duplicate elimination is available, if the retrieved projection covers all attributes of a key constraint that is enforced by that B-tree. Finally, tuples are always retrieved in standard representation from B-trees. Although Transbase uses rule-based query optimization, it uses *cost estimations* drawn directly from the B-tree access method for justifying optimization decisions. Therefore, it uses a technique that probes the B-tree's index pages with suitable predicates, typically intervals on an index attribute prefix. Depending on the number of examined index levels, this approach allows highly accurate selectivity estimation for predicates on indexed attributes, without the necessity of maintaining separate statistical information. All available features are automatically exploited by query optimization.

Besides these standard access method functionalities, Transbase's B-tree implementation covers the complete functional spectrum of optional access method features for improved performance characteristics. Therefore, it participates in the *scan buffering* protocol, enhancing the basic Access Manager protocol in selected operations for achieving higher efficiency by eliminating the necessity to physically close reusable scans. Finally, it offers full support for the optional *savepoint* feature, augmenting the storage layer's arrangements for concurrency and isolation, by providing graceful fail-safe behavior for revoking defective DML operations.

Transbase employs B-trees as the default access method type for both primary and secondary access paths, i.e. if no explicit index type is given in the DDL statement. For secondary access paths, the necessary reference between index tuples and tuples in the base relation is maintained either via the relation's primary key or via the aforementioned IK-surrogate mechanism. Explicit control over this decision is given to the schema designer via an optional clause in the data definition syntax. If unspecified, a default mechanism will automatically choose the space-optimal option at base table creation time.

In addition to their primary purpose as data access paths, B-trees are also used as default data structure for implementing *unique constraints*, by establishing secondary access paths as unique secondary indexes, and the query optimizer will automatically exploit constraint B-trees for data access, if it finds them appropriate.

In addition, B-trees offer convenient properties for several advanced query evaluation techniques. Their internal index nodes contain a dense accumulation of references to leaf pages, which is highly suitable for effectuating prefetch operations, during full table scans and interval scan operations. In particular, the tree structure allows prefetching on multiple hierarchical levels, thereby providing excellent look-ahead capabilities. The impact of prefetching in B-trees is exemplified in Figure.39 at the end of this section. The B-tree's ability to evaluate predicates to a large extent solely by accessing index pages allows it to identify sets of qualifying leaf pages. These leaf pages can be processed in parallel, by partitioning references to these leaves among worker threads, which eventually retrieve the pages from disk, post-filter any remaining predicates, unpack the qualifying tuples and apply a final projection to result tuples in standard representation. Figure.39 demonstrates how prefetching and parallel evaluation harmonize, as early identification of qualifying pages allows effective prefetching of these pages, while the parallel worker threads handle the laborious task of extracting the final results, thereby effectively overlapping computational and I/O operation for maximized utilization of system resources.

Figure.39 summarizes the B-tree's performance characteristics as a primary access path to the data set introduced in *5.1.3 Reference Database* (page 187). These measurements will be used as a reference for comparison against other indexing technologies. The B-tree index examined here represents a primary access path to said data set, where all three dimensions are incorporated into the B-tree's composite search key, i.e.

```
CREATE TABLE B (a SMALLINT NOT NULL, b SMALLINT NOT NULL,
    c SMALLINT NOT NULL, d CHAR(80), PRIMARY KEY (a,b,c));
```

The B-tree demonstrates its balanced characteristics that endorse its position as the ubiquitous indexing technique in DBMS technology. It is capable of rapid mass insertion (SPOOL), and shows how data retrieval (SCAN) scales with the selectivity of the applied predicate, i.e. 100% for the full table scan (FTS), 10% interval scan, and point access. The only exception is the 3-dimensional query box with a selectivity of $30\% \times 10\% \times 10\% = 3‰$, where the B-tree is able to exploit mere 30% selectivity from the predicate on the first dimension, and consequentially it has to conduct a relatively expensive interval scan on a 30% portion of the table.

As to data MANIPULATION, the B-tree possesses balanced performance characteristics for extensive operations altering about 8% of the data volume. These manipulations are uniformly distributed across all dimensions, and as a consequence of the B-tree's clustering, every page is affected by every operation. The DBMS's query processor actively supports these manipulations by providing the scan operator with input streams in a sort order that is matching the B-tree's primary linearization.

The remaining figures demonstrate the impact of selected implementations aspects on the index performance. The first measurement (VARIANT) aims for quantification of the costs incurred by the additional Access Manager layer during query evaluation. Therefore it compares the original Transbase B-tree implementation without surrounding Access Manager to that of the adapted B-tree access module accommodated in the Access Manager framework. It shows that these costs are negligible, as the B-tree integration into the Access Manager framework has actually a slightly better performance compared to the Transbase's original implementation. This minimal improvement can be attributed to minor simplifications in the B-tree interface that were conducted in preparation of its integration into the Access Manager.

The CLUSTERING group concentrates on the influence of physical inter-page clustering on I/O throughput. It compares a FTS operation on a perfectly aligned B-tree structure, as constructed by the initial mass-insertion process, to a FTS on an artificially fragmented data structure, where every 10^{th} page was moved to a random location within the DBMS address space, simulating a B-tree after heavy restructuring through perpetual data modifications. The diagram displays a massive disturbance in the sequential scan performance, a direct consequence of fragmentation that now prevents a reliable prediction of the DBMS's scan behavior

by the underlying operation system and hardware. This effect is countered by the active prefetching capabilities of the B-tree, such that the average I/O rate on the fragmented segment is significantly adjusted towards sequential behavior.

The final PARALLEL FTS measurement compares single-threaded and multi-threaded query evaluation. This is the only benchmark in this scheme, where data is deliberately retrieved from an appropriately prepared database cache, and not from disk, since it intends to focus on CPU-relevant effects. As the query effectuates a simple FTS, it does not involve any expensive data transformations. The savings in elapsed time are therefore predominantly resulting from parallelization of the tuple-extraction process from B-tree leaf pages by parallel worker threads. In this case, the speed-up of parallelization is not limited by the number of available CPUs, but by necessary synchronization of concurrent access to the database cache and other globally shared resources.

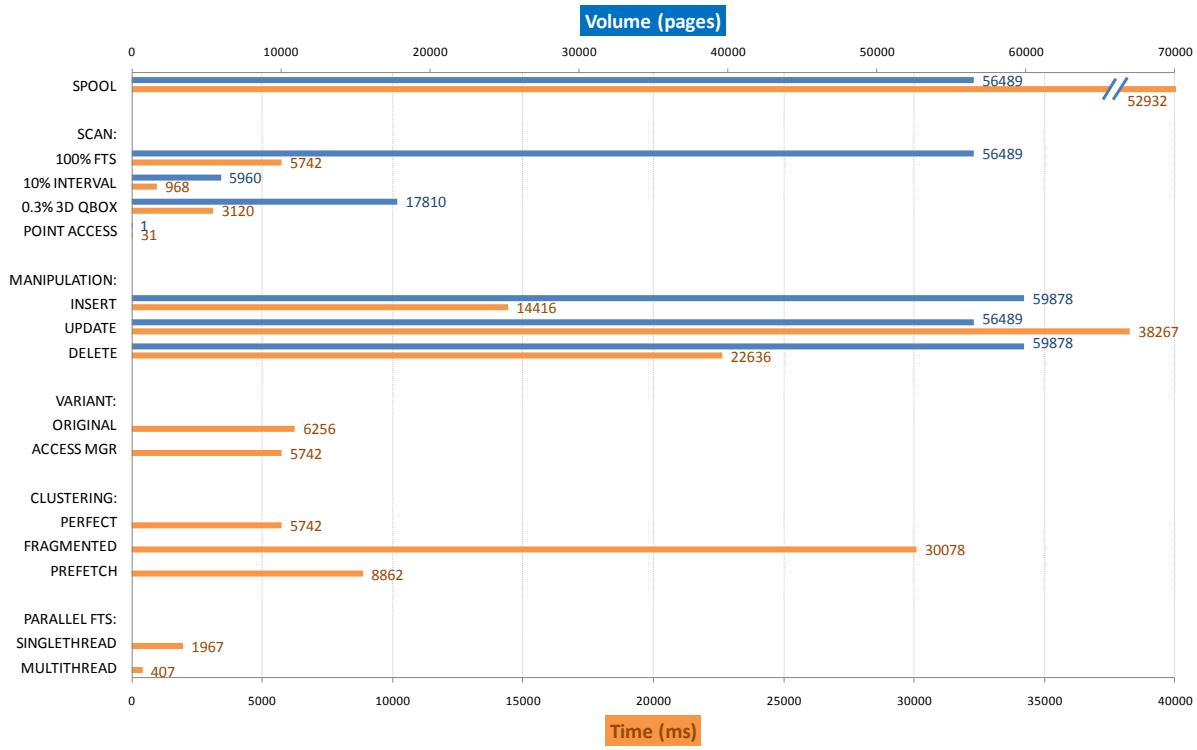


Figure.39 B-tree performance characteristics. The diagram accounts for overall characteristics of the B-tree implementation. All measurements show evaluation times (orange) for the operation denoted on the left, and refer to the lower horizontal axis. Some measurement also exhibit the volume (blue) of data affected by this operation, plotted as number of touched pages, where one page is 32 kbyte in size, and the complete B-tree occupies approximately 1.75 Gbyte. Data volume refers to the upper horizontal axis. Note that values outside the range of the diagram are indicated by a broken bar.

In summary, the B-tree implementation is the most mature and most sophisticated access structure available in Transbase. This makes B-trees particularly interesting for serving as a reusable basis for functional indexes or as auxiliary search structure for custom intermediate access method implementations. Its integration into the Access Manager framework allows rapid adaptation of this technology for novel operational areas.

5.3. UB-Trees

The UB-tree ([Bay96], [Mar99a]) is a generalization of the B-tree for indexing multidimensional data, i.e. multiple attributes of one relation. Its key feature is good responsiveness to a combination of restrictions on arbitrary subsets of indexed attributes, such that the total selectiveness of applied predicates correlates closely with the amount of data to be retrieved from secondary storage. The classical B-tree also supports multidimensional indexing by employing compound keys. However, restrictions on these attributes are not evenly exploited, as an unbalanced prioritization favors index attributes in their order of appearance in the search key. As a consequence, the B-tree performs excellently, if a sufficiently long prefix of index attributes is subjected to highly selective restrictions by a query, but performance will degrade strongly if restrictions on high priority attributes are weak or absent. In contrast to the B-tree, the UB-tree uses a functional mapping of the indexed attributes as search key, which results from interleaving the attribute values' binary representations to the so-called Z-address. By applying conventional ordering to these interleaved keys, the UB-tree clusters its data along the Z-curve or Lebesgue curve (Henri Léon Lebesgue, 1875–1941, French mathematician), a space filling curve that is highly symmetrical in the weightings of individual dimensions. From this symmetry arises the UB-tree's ability to conserve spatial vicinity of multidimensional data in its physical clustering, and its robustness towards selections using conjunctions of arbitrary restrictions on indexed dimensions. This makes the UB-tree the superior indexing structure for multidimensional ad-hoc queries, such as are common in data warehouse and OLAP applications.

Transbase possesses an industrial-strength UB-tree implementation [Ram00] under the product name *Hypercube* index, and while its original integration necessitated several extensions and adaptations in the DBMS's SQL compiler and query optimizer, the implementation of the internal search structure is based on and is strongly coupled with the prevalent B-tree implementation. The UB-tree may be conceived as a classical B-tree using a functional mapping from multiple plain indexed attributes to one singular calculated key. This conception allows

both implementations to share a common code basis, which deviates only in said functional mapping of keys, and a corresponding predicate evaluation mechanism, the so-called *Range Query Algorithm* [Mar99a], that is capable of operating directly on this special key representation. This inseparable integration of the UB-tree access method implementation with the B-tree complicated the former's migration into the Access Manager framework as an autonomous access module. In the current state, the UB-tree still exists as an attachment to the B-tree implementation, yet the UB-tree is accessible via its own access method interface as an independent access structure, ready to be used by other Access Manager modules.

In Transbase, UB-tree access paths are defined using a proprietary extension to SQL. It may be used for defining both primary and secondary access paths, and similar to the underlying B-trees, it is capable of indexing and storing data of arbitrary SQL data types and arity. For optimized effectiveness of spatial clustering in UB-tree indexing, all table definitions must exhibit compulsory *check constraints* on all indexed attributes, reducing the domains of indexed attributes from the SQL base type to the actually used value range, e.g. `longitude NUMERIC(10,7) CHECK (longitude BETWEEN -180 AND 180)`. The corresponding DDL statements have the following form, where `HCKEY` denotes a Hypercube key.

```
CREATE TABLE <table definition with check constraints>
      HCKEY IS <attributelist>

CREATE INDEX <index definition> HCKEY IS <attributelist>
```

In addition to these compiler extensions, the integration of the original UB-tree implementation necessitated several dedicated interfaces to Transbase's optimizer module and query evaluation engine. In contrast to this present integration of the UB-tree, the Access Manager framework allows to incorporate the UB-tree into Transbase in a more generic way, allowing it to share a similar amount of code with the B-tree implementation, but resulting in cleanly separated implementations. The UB-tree's strong dependency on B-tree functionality suggests its implementation as an *intermediate access method* based on the B-tree. This approach implicates that the UB-tree will receive input tuples for insertion or manipulation in standard representation, subsequently transform them using the bit-interleaving technique for calculating the UB-tree's key, and finally forward the modified data to an internal B-tree that is capable of accommodating data in this new representation. Similarly, predicate evaluation using the Range Query Algorithm will process standard restrictions from a given SQL query and translate them into corresponding navigational operations on the internal B-tree's calculated key. The resulting UB-tree is in essence a lightweight implementation of a functional

index, based on the conventional B-tree. Yet it possesses all capabilities for attending multi-dimensional operations efficiently. Integration of the UB-tree via the Access Manager permits using the host system's SQL compiler without specific adaptations, e.g.:

```
CREATE HYPERCUBE TABLE <table definition with check constraints>

CREATE HYPERCUBE INDEX <index definition>
```

The enforcement of suitable check constraints is delegated to the module's **`Create()`** routine, which receives the constraint definitions via its function parameters, allowing it to evaluate and validate them as prerequisites for a UB-tree definition. The query evaluation engine may conduct all necessary operations on the UB-tree via the generic access method interface. The negotiation interface will ensure that all query evaluation capabilities of this access module are exploited thoroughly. The UB-tree is able to produce output tuples of arbitrary projections in standard representation, but in contrast to the B-tree it is able to enforce restrictions [Fen02] in a more flexible and effective manner. A direct exploitability of sort orders is not possible, since the UB-tree uses a primary linearization that is not corresponding to a lexicographical sort order. Transbase uses certain adjustments for speeding up the initial loading phase of UB-trees, where the data spooler module adopts the capability of pre-calculating the UB-tree key values for presorting the input relation and feeding this prepared input set to the UB-trees **`Insert()`** routine. This custom adaptation of the data spooler module for the UB-tree demonstrates the requirement for exploiting arbitrary sort orders and also reveals the deficiency of the query interpreter to provide this functionality for non-lexicographical sort orders. Yet, such custom adaptations as used in this case contradict the generic Access Manager approach. A clean solution will request a lexicographically presorted input stream on one single indexed attribute as input directive for insertion. The host system is able to satisfy this simple requirement, allowing the UB-tree to employ its *TempTris Algorithm* ([Zir99], [Zir04]), which is able to improve the loading process, using a moderately sized temporary storage area. An inverse technique, called the *Tetris Algorithm* [Mar99b], enables the UB-tree to produce a data output stream exhibiting a sort order on one single indexed attribute, for exploitation in a consecutive relational operation. However, neither form of sort order exploitation is currently in use in the present implementation. Alternatively to the Tetris/TempTris approach, the Access Manager allows to resort to non-standard representation for addressing exploitation of sort orders. If a manipulation scan's data input stream is extended with the UB-tree's calculated key, then it becomes possible to apply a conventional lexicographical sort operation on this single artificial attribute for achieving optimal

input sort order for UB-tree mass insertions and manipulations. Finally, negotiation requires a cost function for justifying decisions in query optimization, but although the theoretical foundations of a cost function for accurately estimating the impact of predicate on the scan's result set are available in [Mar00], the rule-based optimizer in Transbase currently decides on employment of UB-tree access paths solely by means of heuristics.

With this, all functionality required for routine operations of the UB-tree is readily available. These basic capabilities allow to employ the UB-tree in highly advanced scenarios, like multidimensional data warehouse applications based on star schemata or hierarchical snowflake schemata. However, when considering such complex application scenarios and their sophisticated query evaluation concepts, especially Transbase's *Multidimensional Hierarchical Clustering* (MHC) (cf. [Pie03] and [Pie01], [Kar02] for related topics), then the limited functionality provided by the access method interface seems insufficient. But if we distinguish between abstract query processing strategies, such as predicate evaluation on dimension tables, fact table access, and residual joins, which are completely independent from the employed indexing methods and the basic access method concepts that implement them, namely interval scans, materialization, and lookups, then it becomes clear that the Access Manager framework is well-prepared to accept these challenges.

The following Figure.40 compares the twin implementations of B-tree and UB-tree, both storing the reference data set described earlier. The UB-tree is defined as:

```
CREATE TABLE UB (a SMALLINT NOT NULL CHECK(a BETWEEN 0 AND 511),
    b SMALLINT NOT NULL CHECK(b BETWEEN 0 AND 511), c SMALLINT NOT NULL
    CHECK(c BETWEEN 0 AND 511), D CHAR(80)) HCKEY IS a,b,c;
```

As expected, both access methods show many similarities with respect to evaluation times and data volume. Bulk loading of an UB-tree is slower because of a difference in both implementations: the B-tree uses the highly efficient *Wiper Algorithm* for constructing the initial data structure from a presorted input stream in one single sweep, using a process resembling an upside-down windshield wiper. The UB-tree, on the other hand, is currently not able to employ this technology on its internal B-tree and therefore suffers a performance penalty in this direct comparison. Similarly, data manipulation on the UB-tree appears slower, but this is caused mainly by different scattering effects of the actual data set on the alternative linearizations. In this special case, the applied manipulations possess a higher locality on the B-tree, which is eventually resulting in its better performance. This effect is likely to tilt in favor of the UB-tree for other data sets, such that in average, both structures will show similar overall

performance. Query processing shows matching performance for FTS and point queries, but a significant difference in handling of 10% interval scan and 3‰ query box processing. While the UB-tree obviously has advantages with the small query box, as it is able to apply the given restriction by retrieving a number of pages that correlates with the predicate's total selectivity, the B-tree is only able to exploit a 30% restriction on the first attribute. On the other hand, the B-tree can clearly outperform the UB-tree in the interval scan. In this case, the B-tree can fully exploit its space filling curve, which is contiguously traversing the requested interval, such that almost all retrieved pages are filled completely with qualifying tuples. The UB-tree's Z-curve, on the other hand, will frequently leap out and reenter the queries interval, leading to a significantly lower hit ratio per page and consequently the UB-tree is forced to retrieve more pages. In addition, the unsteadiness of the Z-curve inhibits efficient sequential scanning, leading to an almost random I/O pattern. This and the higher complexity of the Range Query Algorithm provoke a significantly higher evaluation time. But we must concede that a 10% selectivity is empirically the worst-case scenario for a multidimensional index, and a query optimizer will generally decide in favor of a FTS for restrictions with weaker selectivity. In this special case however, the restriction of the interval scan is very favorable for the B-tree, as it can perfectly exploit restrictions on this particular attribute, but not on any other dimension. In contrast, the UB-tree will always respond similarly to arbitrary 10% restrictions, regardless of their actual composition from restrictions on the available dimensions.

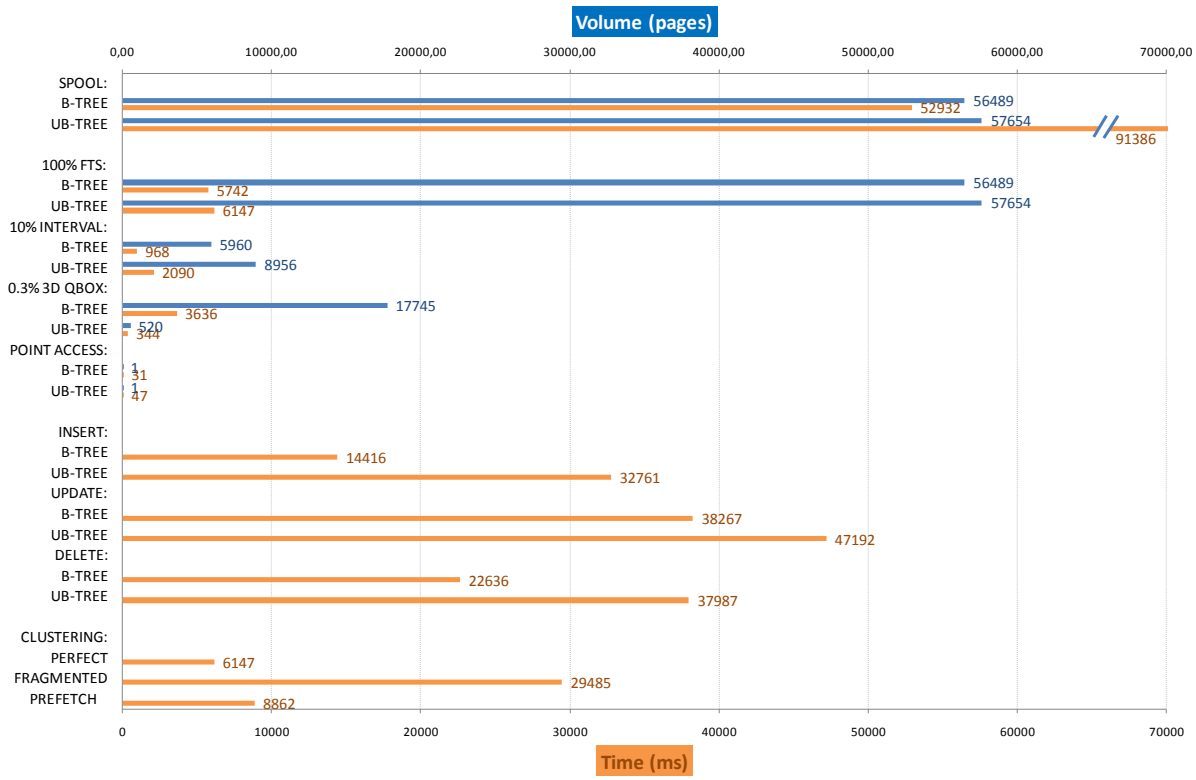


Figure.40 UB-tree performance characteristics. This diagram compares performance characteristics of B-tree and UB-tree implementations. Evaluation times (orange) for the operation denoted on the left refer to the lower horizontal axis. The volume of data (blue) affected by this operation, plotted as the number of touched pages, where one page is 32 kbyte in size, and the complete UB-tree occupies approximately 1.75 Gbyte. Data volume refers to the upper horizontal axis. Note that values outside the range of the diagram are indicated by a broken bar.

In general, the UB-tree is clearly the superior data structure for handling reasonable multi-attribute restriction on vast multidimensional data sets, such as are common in data warehouses and OLAP applications. The final set of measurements in these series demonstrates the UB-tree's capability to counter fragmentation of its physical layout with active prefetching, a property that it actually inherits from the underlying B-tree implementation.

In its current implementation, Transbase employs much of the available knowledge on theoretical properties of UB-trees for taking practical advantage in query processing. In addition, the employment of a cost function for supporting or even replacing heuristic-based query optimization will lead to better query evaluation plans in future. Moreover, the Tetris/TempTris technology is capable of further improving bulk loading and query evaluation in Transbase. A future integration of these currently unutilized resources of UB-tree technology will be significantly facilitated, if conducted as a structured extension via the Access Manager framework.

5.4. Flat Table

The flat table is the first prototype of a Transbase access method that was designed and built entirely for the Access Manager framework. It represents a complete, self-contained, and fully integrated implementation of an access module. After a remarkably short development period, it has reached the status of a technically mature, industrial-strength access method implementation and has already become an integral part of the Transbase product. Although very minimalistic in its original design, it possesses a number of interesting properties that allows it to match the functional and performance-related characteristics of the time-tested and streamlined B-tree implementation in many usage scenarios. It even manages to outperform the B-tree in a number of important aspects.

Originally, the flat table was designed for dealing with massive, time-critical bulk loading processes, especially in data staging and data transformation procedures (ETL), and for inexpensive logging of data manipulations, e.g. conducted through database triggers for documentation purposes or later auditing. Flat tables are organized as doubly-linked lists of pages, stored in the host system's internal storage facility and functioning as containers for tuples in standard representation. The central feature of the flat table is the absence of any search keys and internal search structures. The segment's description page, which functions as a gateway for accessing the data structure, only possesses references (page numbers) to the list's head and tail elements, as depicted in the following Figure.41.

In contrast to the B-tree, which suffers a certain performance penalty when conducting randomized mass-insertion due to necessary expenditure for preserving its linearization and maintenance efforts on its search structure, the flat table simply appends data to the existing data basis. Consequently, it uses the insertion order as its primary linearization, which in general is data independent. The flat table is tailored for efficient insertion, primarily by one single writing scan appending data at the tail of the list, although arbitrary concurrent writing operations (i.e. insertion, deletion, and updates) are generally possible. In contrast to the B-tree's multi-way search capabilities, the flat table does not possess a dedicated search structure, hence it does not offer any direct access capabilities and it is typically accessed by sequentially scanning from the head of the list towards its tail. Beyond that, relative forward and reverse navigation from a given scan position is also possible.

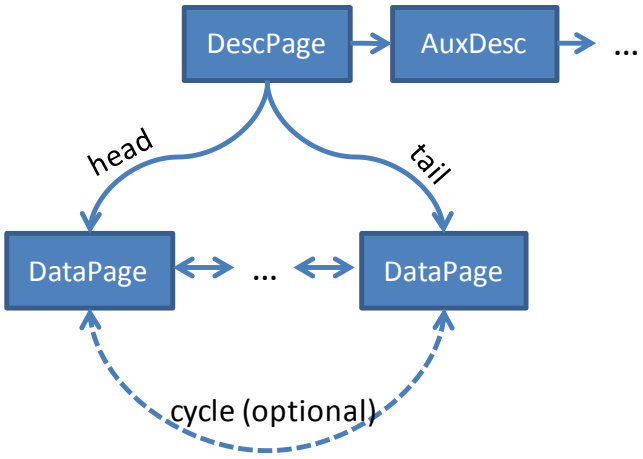


Figure.41 Organization of flat tables. Data is always accessed via the description page. If the page size setting of a database should not suffice for storing all relevant data for accessing the table in one single description page, then the access module will allocate auxiliary description pages and chain them to the data structure. The description page maintains references to the head and tail page of the storage structure, which is organized as a doubly linked list of pages. This data structure supports all necessary reorganization procedures (split and merge), and also allows relative navigation forwards and backwards through the stored data. The ends of the list may be optionally connected, such that the data structure becomes a cyclic list. This variant of the flat table is able to function as a relational FIFO container.

Similar to the B-tree, the flat table stores individual tuples of any arity and arbitrary SQL data types in an *unaligned* representation and uses *attribute reordering* for additional compactness, yet in contrast to the intra-page clustering of the B-tree, the flat table cannot exploit *attribute suppression* for common tuple prefixes. As a consequence of attribute suppression, a B-tree with a compound search key but low selectivity on the key's prefix will exhibit a higher capacity of tuples per page than a flat table storing an identical data set. But on the other hand, the limitation to sequential access permits a significantly simplified page layout for flat tables, leading to a constant reduction in storage requirements of 2 bytes per tuple compared to the B-tree, which becomes significant when storing relatively small tuples. As consequence of these two contrary effects, both data structures typically have approximately identical space complexities for storing identical data.

Reading access to a flat table is granted to an arbitrary number of concurrent scans. Any scan may also manipulate data at its current scan position. Manipulation always preserves the original insertion order, i.e. updates are performed in-place. If the storage capacity of the current page is exceeded after an update, then the page is split by allocating a new page and the original page's contents are divided among the old and new page. Deletions and update operations may reduce the fill level of the concerned pages, thereby degrading storage utilization of the overall data structure. Therefore, flat tables are equipped with the ability for self-

reorganization, guaranteeing an *average* filling degree of at least 50%, and typically preserving a considerably higher fill level. Whenever the fill level of a page drops below 50%, then neighboring pages are probed for potential merging. All these operations strictly preserve the original insertion order, which is crucial for consistently repositioning of concurrent scans of the same transaction when conducting *scan maintenance*. Scans belonging to other transactions are not affected by such reorganization, since they are protected by the storage layer's multi-version concurrency control.

In some cases an implicit functional dependency might exist between insertion order and inserted data, for example in a continuous stream of sensor data exhibiting ascending timestamps. Although such additional semantics are not consciously maintained, verified, or exploited by the flat table or by the host system's query evaluation engine, an application may still exploit this property when performing relative scan operations and positioned manipulation operations on this segment, as long as updates on the order-relevant attributes do not invalidate this correlation.

Input order preservation is also used in a special variant of this data structure, the flat table operates as a relational FIFO (First-In, First-Out) container of limited total size. The data structure initially operates as normal flat table, until it reaches a predefined target size (i.e. a maximum number of data pages) through insertions. At this point, head and tail of the list are chained together, and the structure becomes a cyclic doubly-linked list (cf. Figure.41, page 201). In this state, the flat table will stop allocating further pages and start overwriting the page containing the oldest data with newly inserted data, according to the FIFO strategy. Such a data structure is suitable for storing event logs in a relation of limited total size, where outdated records are automatically deleted. A sequential scan on this log always produces data chronologically, and the host system's query engine allows applying arbitrary SQL transformations for analyzing the log information. One could also imagine another variation, operating a Flat table as LIFO (Last-In, First Out) container, or even as a LRU (Least Recently Used) stack, again exploiting input order preservation, but without the space limiting constraint of the FIFO container. However, the current Flat table implementation supports only FIFO functionality.

The absence of any direct access capabilities renders the flat table inapt for use as secondary access path. Yet as a primary access path, this lightweight implementation is capable of accomplishing all fundamental functionality, including the ability for indexation via suitable secondary access paths. Therefore, the flat table must maintain some unique tuple identifica-

tion for associating base tuples with index tuples. As the flat table itself is not suitable for maintaining an innate primary key in its data, because it cannot verify uniqueness of newly inserted data efficiently, it has to be augmented with a separate data structure for generation of unique keys. The present implementation uses the Access Manager's IK service (cf. *4.3.3 Tuple Identification and Indexing*, page 125), for generating unique surrogate keys, which are stored alongside the corresponding data tuple in the flat table pages. The Access Manager's IK service automatically associates the number of the page inhabited by a new tuple with the tuple's IK value, and stores this pair in the auxiliary IK-tree structure. Finally, the IK-tree allows identification of the flat table page storing the tuple for any given IK, as required for *materialization* of base tuples after data retrieval via secondary access paths.

Flat tables with secondary indexes provide direct access capabilities and efficient enforcement of unique constraints, thereby acquiring two functionalities a bare flat table does not possess. On the other hand, construction and maintenance of secondary indexes and auxiliary IK-tree will severely compromise the flat table's superior performance characteristics for bulk-insertion.

One major handicap of flat tables compared to B-trees is their missing look-ahead capabilities. A flat table, being a linked list, can only know the page number of the immediate successor and predecessor pages relative to its current scan position. This massively impairs its capabilities to exploit parallel asynchronous I/O operations on modern storage hardware, because it cannot issue bulk I/O requests to be handled autonomously by the I/O subsystem. Although the effect is often attenuated, if the flat table pages are arranged in a physically adjacent way, such that a sequential scan operation on the flat table results in a sequential read operation on the underlying file. In this case, the storage system and hardware will be able to recognize this simple access pattern, allowing autonomous prediction and ahead-of-time scheduling of future I/O requests. Thereby prefetching is delegated from the access layer to the underlying operating system and hardware. However, if the address space of the database system is already heavily fragmented, or if data is inserted in numerous smaller chunks over a long period of time, allowing intermixed allocation of pages belonging to foreign segments, or even more common, alternating allocation of base table and index pages belonging to the same relation, then the probability that the operating system is capable of conducting effective prefetch operations is declining rapidly. Especially if the storage system operates in direct I/O mode, i.e. the operating system's file cache is bypassed for I/O operations, then automatic prefetching is undermined completely, resulting in poor sequential scan performance on a flat

table. To compensate these effects, the flat table can be optionally equipped with an auxiliary data structure, the *prefetch list*, providing the required look-ahead capabilities. This data structure is essentially a second, auxiliary flat table, and it is operated via the Access Manager's access method interface, but it is significantly smaller than the main storage structure. Instead of storing data tuples, the prefetch list merely stores page numbers of the main structure in exactly the same sequence as they are linked into the master list. The prefetch list has to be maintained whenever the main structure is allocating or removing data pages during data manipulation. Conceptually this structure is organized as an independent flat table, and it may even have its own prefetch structure, permitting recursive look-ahead functionality, similar to the B-tree. But since the prefetch list stores only page numbers, while the B-tree must also store the corresponding search key separators, the former has a notably higher capacity for page references, which is generally sufficient for conducting adequate prefetch batches.

In Figure.42 we compare performance characteristics of Transbase's B-tree with its flat table. But before we can proceed to measurements, we must make one amendment for guaranteeing fair comparison. By default, mass insertion will fill a B-tree's pages to a level of 80% of their total capacity, such that subsequent insertions will not immediately lead to split operations. On the other hand, all insertions into flat tables are conducted as append operations. Hence, it is reasonable to use all available capacity of flat table pages when conducting mass insertion, thereby reducing the overall size of the storage structure. As a consequence, direct comparison between B-trees and flat tables must compensate for different page utilization. In the following, we circumvent this discrepancy by forcing the B-tree to fill its leaf pages completely.

The measurements in Figure.42 demonstrate the performance characteristics of a flat table storing the reference data set, defined as:

```
CREATE FLAT TABLE F (a SMALLINT NOT NULL, b SMALLINT NOT NULL,
    c SMALLINT NOT NULL, d CHAR(80));
```

Obviously, the flat table is not suitable for efficient query evaluation, as it always has to process the complete relation, independent of any selectivity of applied predicates. In fact, a higher complexity of the applied predicate will necessitate more expensive post-filtering and thereby increase query evaluation time. On the other hand, the flat table demonstrates good robustness against massive restructuring operations, as it is able to outperform the B-tree in initial spool operation and subsequent bulk insertion, where the B-tree suffers a performance

penalty for reordering the input set according to its primary linearization and for building and maintaining its index structure. All insert operations on the flat table operate strictly locally at the tail of its list. Similarly, update operations on a flat table are performed in-place, allowing it to complete all required operations with one single sweep over the base relation, while the B-tree is forced to process the update in two phases, since data must be moved as a consequence of search key updates.

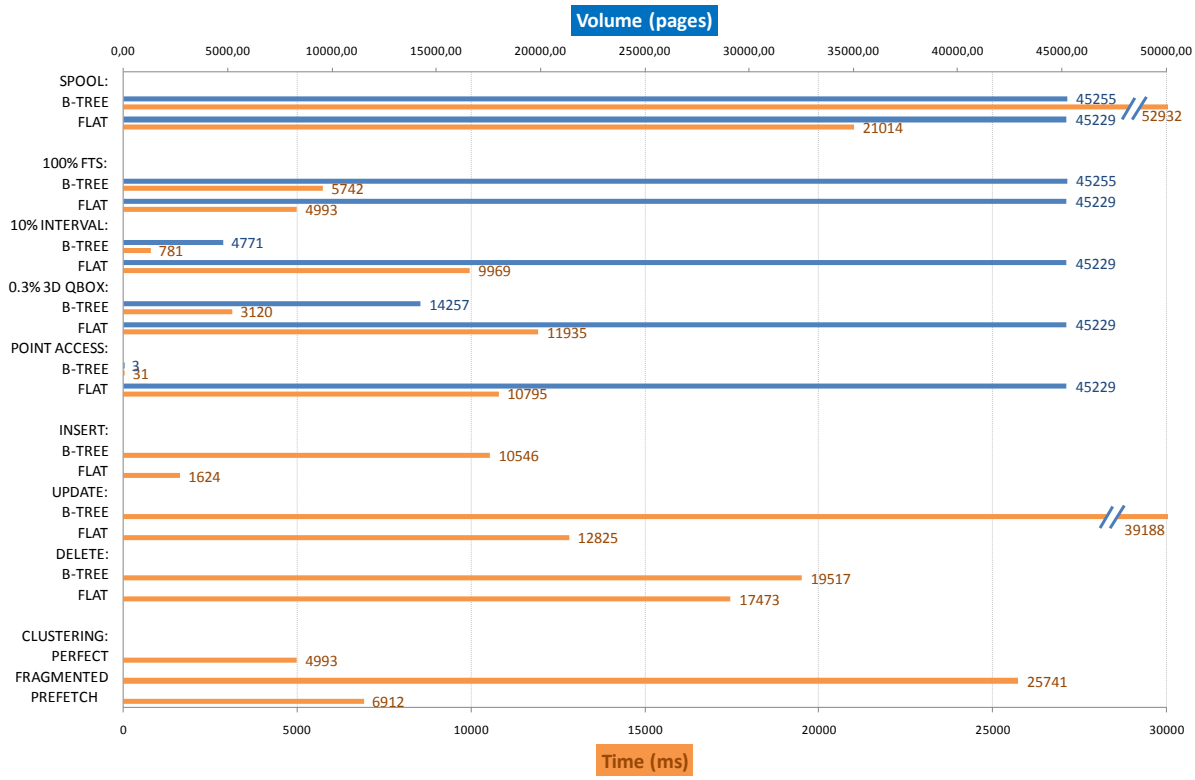


Figure.42 Flat table performance characteristics. This diagram compares performance characteristics of B-tree and flat table implementations. Evaluation times (orange) for the operation denoted on the left refer to the lower horizontal axis. Some measurement also exhibit the volume (blue) of data affected by this operation, plotted as number of touched pages, where one page is 32 kbyte in size, and the complete flat table occupies approximately 1.5 Gbyte. Data volume refers to the upper horizontal axis. Note that values outside the range of the diagram are indicated by a broken bar.

Finally, even the delete operation on the flat table has a minor performance advantage, because in contrast to the B-tree, it has no index to maintain, only a very compact prefetch list. The final measurement demonstrates the effectiveness of this prefetch list for providing valuable look-ahead capabilities on a fragmented flat table segment.

In summary, flat tables are appropriate containers for volatile data, or as a temporary storage container for data that has to undergo some extensive refinement process. After the manipulation process has concluded, the contents of a flat table can either be transferred into a primary

index structure, or alternatively a secondary index of arbitrary type may be built directly on the flat table.

5.5. Bitmaps

Bitmap indexes are the prime example for the construction kit principle of the Access Manager framework and its distinguished ability for encouraging and cultivating reuse of existing components. Similar to flat tables, bitmap indexes reside completely inside the storage area controlled by the host DBMS, but contrary to flat tables, they never interact directly with the storage layer. They are entirely built on top of the DBMS B-tree layer, as an *intermediate* access module. All required storage and data retrieval functionality is attended by inserting, deleting, or accessing data from an auxiliary B-tree structure that serves as permanent storage container for bitmaps. Hence, the B-tree functions as a mediator for mapping the tuple-oriented storage requests of the bitmap index onto the page-oriented storage layer.

Using this approach, bitmap indexes can be implemented as a comparatively thin layer. As an example, the bitmap module does not implement any search structures of its own. This functionality is completely provided by its B-tree component. From the perspective of the object-oriented programming paradigm, the bitmap index *extends* the B-tree structure, thereby *inheriting* the bulk of its necessary functionality. Nevertheless, the bitmap module represents a highly specialized access method implementation that shows fundamentally different behavior compared to the properties of its central building block.

Before we start examining the properties of the present bitmap implementation, we will establish the general concepts of bitmap indexes. A bitmap index is capable of storing and efficiently retrieving data of the form $(v_1, \dots, v_n, k_1, \dots, k_m)$, where v_i, k_j are attributes of a base relation T. We call $v = (v_1, \dots, v_n)$ the indexed attributes on which the bitmap index will provide rapid predicate evaluation and direct access capabilities, while the attributes $k = (k_1, \dots, k_m)$ constitute a relational key on T and serve as identification for locating the base tuple corresponding to each index tuple. The prevalent form of bitmap indexes uses *equality-encoded bitmaps* (cf. [Cha98] for more details on this and other encoding schemes). In other words, for every distinct value of indexed fields $t_v \in \pi_v(T)$ that is actually appearing in the base relation T, the index will store one bitmap b_{t_v} that is associated with t_v. These bitmaps provide a bijective mapping for every unique key $t_k \in \pi_k(T)$ to a bit position pos_{t_k} in each bitmap, such that for every row in T with key t_k that exhibits the value t_v on the indexed

fields, the bit pos_{t_k} in the bitmap b_{t_v} is set to 1 (1-bit). Otherwise it remains 0 (0-bit). If T contains *n* tuples and the indexed fields contain *c* distinct values t_v (including SQL NULLs), then the bitmap index consists of *c* separate bitmaps, each of a length of at least *n* bits. If the mapping $t_k \rightarrow pos_{t_k}$ is not dense, then the resulting bitmaps are correspondingly longer. Using equality encoding, the sets of 1-bits in any two bitmaps are disjoint. Otherwise T would contain two tuples with key t_k, exhibiting two different values of t_v at the same time, thereby violating the key constraint. Also, the total number of 1-bits in all bitmaps equals the number of tuples in T. The following figure exemplifies these relationships.

$\pi_k(T)$	$\pi_v(T)$	b_{NULL}	b_0	b_1	b_3
1	1	0	0	1	0
2	3	0	0	0	1
3	NULL	1	0	0	0
4	0	0	1	0	0
5	3	0	0	0	1
6	1	0	0	1	0
7	1	0	0	1	0
8	3	0	0	0	1

Figure.43 Equality encoded bitmaps. This example shows projections of a relation T with 8 rows, having a dense relational key k, and a non-key column v with four distinct values. An equality encoded bitmap index on v requires four bitmaps of length 8, in this case using the identity for mapping $t_k \rightarrow pos_{t_k}$.

Our implementation stores bitmaps in an auxiliary B-tree. As a B-tree arranges its data in fixed sized pages of the DBMS's storage layer, e.g. 8 kbyte, and one B-tree tuple may not exceed the size of a data page, bitmaps have to be compressed and eventually split-up for permanent storage. Therefore bitmaps are partitioned into fixed length bitmap *chunks*, each chunk covering 1 Mbit. Empty chunks containing only 0-bits are not stored but discarded immediately, since they can be easily reconstructed on demand. The remaining chunks are subjected to a multi-stage compression mechanism. The first stage uses an inexpensive run-length encoding scheme, relying on low-level routines for optimal hardware support. It offers best compression results for very sparse and very dense bitmaps. The second stage uses a dictionary-based compression algorithm for further size reduction. Finally, if the compressed chunks are still not fitting into pages, they are split into page-sized *fragments*. These fragments are finally inserted into the auxiliary B-tree as tuples, having the general structure $\left(\underline{v}, \underline{pos_{base}}, bulk\right)$. Again v represents indexed attributes from the base relation to be stored redundantly in the secondary index. These attributes have the same types as in the base relation and they possess the highest weight in the storage B-tree's compound key. The field

pos_{base} is an integer number, describing the position of this bitmap fragment's first bit in the bitmap. This field has also the lowest weight in the B-tree's compound search key. Finally, '*bulk*' stores a binary array containing the fragment of an encoded and compressed bitmap chunk as a descriptive field of the B-tree. The maximum size of the *bulk* fragment is chosen such, that the complete tuple occupies at most the maximum payload of one data page. Due to variable compression rates, the sizes of tuples may vary, yet the B-tree will guarantee a worst-case page utilization of 50%.

This implementation possesses all typical properties of equality-encoded bitmap indexes. It provides its highest compactness, if the indexed attributes exhibit a low selectivity. Owing to bitmap encoding and compression, the compactness of bitmap indexes is substantially higher than that of the corresponding standard B-tree, storing identical information in standard tuple representation. But, in contrast to the underlying B-tree, bitmap indexes are not suitable for storing additional, descriptive attributes besides v and k.

In SQL, a *bitmap index* is created using the standard DDL syntax for secondary index definition. The current implementation does not support creating *bitmap tables* as base relations, because it lacks mandatory primary access pats functionality, i.e. it supports insert and delete operations, which are necessary for secondary index maintenance, but it supports neither searched nor positioned update operations. As soon as this functionality is implemented, it will become possible to create bitmap tables.

```
CREATE BITMAP INDEX <indexname> ON <tablename>(<attributelist>);
```

This statement will create a secondary index for a base relation `<tablename>` on the columns specified by `<attributelist>`. Generally, the host system will automatically incorporate the base relation's primary key attributes into the bitmap index definition. Alternatively, it will resort to the IK-surrogate mechanism, if the base table provides IK information.

Bitmap indexes are particularly well suited for indexing columns of low selectivity, hence they are typically used for indexing single attributes representing some coarse classification. Due to their native representation as bit arrays, bitmap indexes support efficient logical bit-operations, in particular intersection and union of bitmaps, realized as low-level coded, hardware-aided binary AND/ OR operation. As a consequence, building separate bitmap indexes on several attributes offers an alternative approach for processing multidimensional restrictions, where inexpensive combination of arbitrary bitmaps allows flexible mixing and

matching of multidimensional predicates in ad-hoc queries. Possible strategies for exploiting bitmap operations in query processing will be discussed separately.

Equality-encoded bitmaps are capable of efficiently evaluating queries of the form:

```
SELECT k,v FROM T WHERE v = (const_1,…,const_n);
```

The specification of the underlying B-tree immediately suggests how search operations on this storage structure are conducted. Predicates on v are applied unaltered to the B-tree's search key. Therefore, the B-tree provides full support for equality queries, as well as for arbitrary interval queries on v. A minor additional arrangement in the bitmap index implementation allows extending the predicate to points or intervals on k, e,g.

```
SELECT k,v FROM T WHERE v=(const_1,…,const_n) AND k=(const_n+1,…,const_n+m);
```

This is accomplished by applying the aforementioned bijective mapping $t_k \rightarrow pos_{t_k}$ to the given predicate and subsequently retrieving all qualifying bitmap fragments from the B-tree. Obviously, such translation of predicates can be accomplished easily, if the mapping $t_k \rightarrow pos_{t_k}$ is monotonous, as it is the case in this implementation. Since the B-tree stores coarse bitmap fragments, the result set retrieved from the B-tree is a superset of the actual query result, and has to be post-filtered for compliance with the original predicate, i.e. 1-bits representing keys that do not match the requested interval are purged from the result bitmap. With this extension, the presented bitmap implementation offers efficient support for an extensive class of predicates on all stored attributes.

Besides equality-encoding, a number of alternative bitmap encodings exist, with range-encoding [Cha98] being the most popular among them. A range-encoded bitmap b_{t_v} exhibits 1-bits at position pos_{t_k}, if the corresponding base tuple is smaller or equal to the bitmap's associated value t_v. Range-encoding responds efficiently to interval queries of the form

```
SELECT k,v FROM T WHERE v BETWEEN (const_1,…,const_n) AND (const_n+1,…,const_2n);
```

The result is calculated by using an equivalent term that is easily mappable to efficient bitmap operations:

```
SELECT k,v FROM T WHERE NOT v≤(const_1,…,const_n) AND v≤(const_n+1,…,const_2n);
```

Hence, range-encoded bitmaps can answer simple interval queries by retrieving no more than two bitmaps, while equality-encoded bitmaps must retrieve one bitmap for every value falling

into the query interval and combine all bitmaps into one conjunct bitmap. On the other hand, range-encoded indexes have to retrieve two bitmaps for answering a common point query. In addition, maintenance of range-encoded bitmaps is expensive, because several bitmaps have to be updated when one new tuple is inserted or deleted, making range-encoding less suitable for frequent updates than equality-encoding.

The concept of range-encoding can be driven further towards bitmap binning [Rot04], suitable for indexing attributes of high cardinality domains. Instead of storing a separate bitmap for every single t_v, bitmap binning partitions the domain of v into intervals (bins) and stores only one bitmap per bin, thus reducing storage complexity. Consequently, the results produced by such bitmaps contain potential hits in bitmap bins at the margins of the query interval. For these potential hits, the exact values of t_v have to be materialized from the base-relation for post-filtering. Although the current implementation supports only equality encoded bitmap, the Access Manager framework is fundamentally capable of coping with the peculiarities of the presented alternate bitmap encodings and it is suitable for their implementation.

Bitmap encoding and compression offer massive savings in storage requirements for bitmap structures. At the same time, these techniques are the main reason why bitmaps respond poorly to data manipulation and consequentially have acquired their reputation as read-mostly search structures. This effect is caused by the non-locality of the bitmaps representation. An update operation altering one single bit in the bitmap has to retrieve a compressed chunk from the disk and expand it in main memory. Now the required operation is conducted, manipulating the single bit in the uncompressed bitmap representation. Afterwards, the chunk has to pass again though the compression cycle and ultimately the compressed chunk is split into page-sized portions and stored in the B-tree. Although this procedure describes roughly the flow of operations in this particular implementation, we presume that some general conclusions drawn from this example can be transferred to arbitrary bitmap index implementation. Besides being already expensive in terms of I/O and CPU-intensive operations, this procedure inevitably triggers additional performance penalties for concurrency, logging, and recovery. For example, a bitmap implementation could simply rely on locking and concurrency precautions provided by the underlying B-tree. But one must be aware that a one-bit-manipulation will lock a complete chunk, potentially leading to locks on several B-tree pages. Moreover, a lock on one chunk corresponds to locks on one million tuples, i.e. this form of locking on bitmaps is very coarse and therefore clearly insufficient for data manipulation in high-concurrency environments. As compression and encryption are conceptually equivalent,

changing one single bit in the uncompressed bitmap will possibly cause extensive changes in the compressed bitmap representation, owing to *diffusion*, a property of encryption algorithms that also applies to dictionary-based compression. Such diffusion of a small local change across a complete chunk will inevitably have adverse effects on the performance of the DBMS logging facility, where logs of page deltas are generated and retained as recovery precaution.

These properties of compressed bitmaps make them unattractive for frequent modifications. Without further precautions, mass-updates on relations with bitmap indexes may become that expensive that dropping the bitmap indexes before performing the updates and rebuilding them from scratch afterwards represents a viable strategy. To compensate for this shortcoming, this bitmap implementation comprises its own caching facility. The goal is to gather as many update operations as possible and perform them inexpensively on an uncompressed bitmap chunk in main memory. Only if the update operation is completed or if the cache memory is exhausted, modified chunks are eventually pushed to permanent storage. This internal cache is organized as a LRU structure on bitmap chunks. In order to maximize memory efficiency, this LRU is partitioned into four levels. Most recently used chunks are held uncompressed, offering maximum affinity to updates. Chunks that have not been touched for some time are compressed using inexpensive run-length-encoding. The next stage applies additional dictionary-based compression, before the chunks are finally forced into a temporal storage segment on disk. This persistent temporal storage segment is organized as a temporary B-tree, which is structurally identical with the B-tree used as permanent storage facility. Therefore temporal storage offers the same efficient lookup capabilities as the permanent structure and both variants share a common code basis. The fundamental difference between the two segments is the substantially lower I/O costs on the private temporary segment, without concurrency, logging, and disk recovery provisions.

We already discussed how bitmap indexes are able to process predicates on all available attributes. In addition, this implementation is able to produce arbitrary projections of present attributes as result sets when operating in standard representation. But the bitmap indexes' unique feature is their natural ability to operate on bitmap representation, and this internal representation opens several new opportunities in query evaluation. For example, a bitmap index is capable to produce data in standard representation in ascending or descending lexicographical sort order on v, a property inherited from the underlying B-tree. Alternatively, it is also able to inexpensively combine the final set of qualifying tuple identifiers into one result

bitmap and then bitwise extract result tuples in ascending or descending pos_{t_k} order. As the bijective mapping $t_k \rightarrow pos_{t_k}$ is monotonous, the result is correspondingly sorted on t_k. In addition, bitmap indexes are predestined for `DISTINCT(`v`)` queries, as this aggregation is used internally for partitioning the stored relation into bitmaps. Finally, and probably most important, bitmap indexes are able to deliver their result in bitmap representation χ_{bm}, such that algorithmic units accepting bitmap input are able to conduct further transformations in this representation. These output configurations are negotiable during the query optimization phase via the Access Manager protocol. Negotiation is supported by a simple cost function for bitmap indexes, which mainly relies on the underlying B-tree's assessment for delegated scan operations and finally adds local costs, e.g. for bitmap post-filtering or for conversion to standard representation.

Some algorithmic units are able to accept input in bitmap representation, thereby reducing costs for representation conversion between bitmap scan and successive algorithmic units. Among these units, implementations of set intersection and set union conducted on bitmaps are particularly promising, as they allow rapid calculation of conjunction and disjunction of partial query results from different secondary bitmap indexes on the same base relation. Consequentially, bitmap-enabled implementations of these two operators were added to Transbase's pool of algorithmic units for exploiting bitmap representation. They distinguish themselves from their standard counterparts in several important ways. First of all, they do not require presorted input, but operate on streams of bitmap chunks in arbitrary order. This becomes possible as a consequence of high compression rates, allowing to retain complete bitmaps in the previously presented LRU structure for bitmap chunks. During query evaluation, it suffices to decompress only the currently processed chunks into run-length encoding, as many bitmap operations are conducted directly on this compact format. This form of bitmap intersection and union proved itself highly efficient, so the bitmap algorithms were extended to accept unsorted data streams in standard representation. In this mode, the algorithmic unit will construct and manipulate bitmaps on-the-fly. Similar to the bitmap scan, a bitmap operator is able to enforce selections, projections, distinct, and sort operations on its internal bitmaps, before producing its result in standard representation. Of course it may also produce its result in bitmap representation for further processing, e.g. in cascades of bitmap intersections and unifications. Apart from these two implemented operators, bitmap representation allows for other interesting operators, for example aggregation of compact bitmaps,

where the frequently used `COUNT(*)` operator seems particularly promising, but also set difference, set equality, etc are candidates for future implementations.

Figure.44 compares bitmap, UB-tree, and B-tree secondary indexes on the reference data set. In each case, the base relation is a dedicated flat table storing IK surrogates for indexing. The indexes cover all three dimensions of the reference data set in anticipation of arbitrary ad-hoc queries, therefore we use three bitmap indexes and three B-trees, each one handling one single attribute, while one UB-tree index is sufficient for handling all three dimensions. The resulting B-trees occupy approximately 112 Mbyte each, adding up to a total of 336 Mbyte, the single UB-tree uses 224 Mbyte, and the three bitmap indexes are compressed to a total of 11 Mbyte. The index creation times are dominated by the sort operation establishing the input sort order for optimal mass insertion. Consequentially, the UB-tree benefits from its ability of using one single index structure requiring only one sort operation.

In the FTS and interval scan operations, the bitmap index is able to outperform the two other candidates slightly in terms of evaluation time. Therefore it reads the smallest of three available indexes completely, which has a compressed size of only 3 pages. This unnaturally high compression rate is a consequence of the homogeneous structure of the artificial reference data set. However, the bitmap index is only able to retain a comparatively small overall performance benefit, because the complete data set has to be converted from bitmap representation into standard representation for further processing. The 3‰ query box and point access use-cases reveal the bitmap index's strengths for processing multidimensional queries. Although the query processor has to retrieve intermediate result sets from three bitmap indexes, it is able to combine them efficiently using direct bitmap intersection. The query processor essentially uses the same strategy for B-tree indexes, but it has to use the more expensive sort-merge algorithm for intersecting intermediate results in standard representation. Hence, the ability of the bitmap index to dispense of this sort operation allows it to get close to the UB-tree's performance in the latter's primary field of application.

The final two measurements shall give an impression of the various secondary indexes' response to massive data manipulation. The first measurement demonstrates how mass insertion performs, if all indexes are maintained with an input set that is appropriately presorted to match the indexes' internal linearizations. Mass deletion, on the other hand, was deliberately conducted in the input order corresponding to the base relation's linearization, leading to randomized deletions on secondary indexes. The bitmap index shows its ability for good performance, if updates are applied in conveniently presorted batches, while it exhibits

massive performance degradation when operated in random mode, as it is likely to happen in concurrent multi-user environments operating on transactional data.

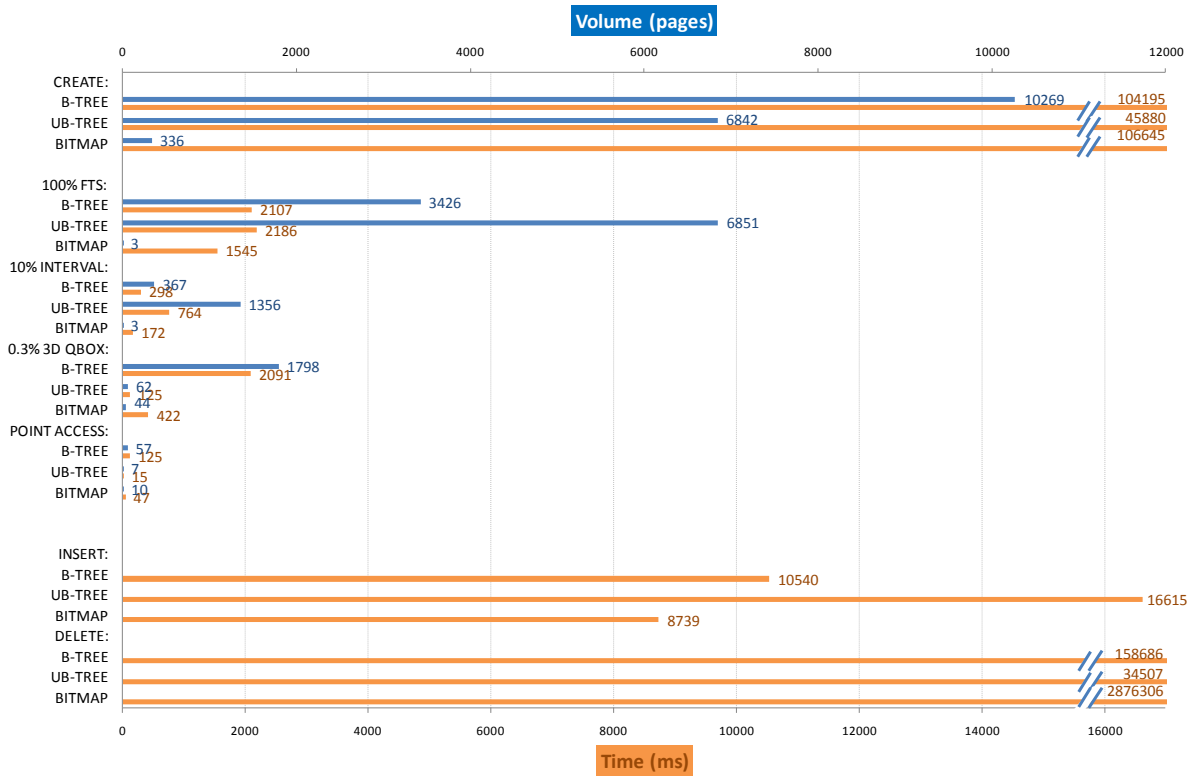


Figure.44 Bitmap performance characteristics. This diagram compares performance characteristics of three bitmaps, one UB-tree, and three B-trees, each index type operating as secondary indexes on three dimensional data of separate flat tables. Evaluation times (orange) for the operation denoted on the left refer to the lower horizontal axis. The volume of data (blue) affected by this operation is plotted as the number of touched pages, where one page is 32 kbyte in size, and three bitmap indexes occupy approximately 11 Mbyte while three B-tree indexes use 336 Mbyte. Data volume refers to the upper horizontal axis. Note that values outside the range of the diagram are indicated by a broken bar.

Bitmap indexes show good performance in multidimensional query evaluation, comparable to that of UB-trees, although UB-trees exhibit superior query evaluation times for a moderate number of dimensions. But as the number of dimensions increases, they will ultimately suffer from the '*course of dimensionality'*, where multidimensional clustering becomes ineffective. This is the area where bitmap indexes have a true advantage over UB-trees, but only under the prerequisite that all participating dimensions are appropriate for bitmap indexing.

5.6. File Table

The file table access module provides direct read-only access to external data, located in text files outside the database. Therefore, the file table module qualifies as a simple data integra-

tion layer. It integrates plain files containing relational data, formatted in a predefined way, e.g. rows are stored in one line and column values are separated by a delimiter character. CSV (comma-separated values) is a popular format that is suitable for this purpose.

Due to its read-only limitations, the file table allows for a surprisingly simple implementation. The module is in essence a scanner/parser, capable of reading text files and converting them into data in standard representation. The current implementation offers two different parsers, one for a delimiter-separated value (DSV) format and one for structured XML documents. The formats are discerned using a `FORMAT` specification clause in the table definition statement:

```
CREATE FILE(<path> [FORMAT DSV|XML]) TABLE <tablename>
    (<FieldDefinitions>)
```

The current implementation uses textual representation for data, i.e. numeric values are represented as character strings, which in general is considerably less compact than the corresponding binary representation. Using a binary format could be easily accomplished, as it only requires the provision of an appropriate parser and a corresponding specification for binary file formats as discriminator in the `FORMAT` clause.

As the file table possesses no search structure allowing for direct access capabilities, data is always accessed sequentially in storage order. Consequently, no primary key declaration is allowed in its DML statement. With file tables being read-only tables, there is no need to check a primary key constraint for inserted or modified data. Instead a primary key constraint serves as an assertion of a data property towards the database system, to be used by the query optimizer for supporting planning decisions. In order to guarantee correctness of optimization decisions and resulting plans, the system has to validate this assertion at table creation time. However, the current implementation does not validate the data in any way, so it deliberately chooses not to accept primary key declarations, i.e. the file table's **`Create()`** method will actively decline a primary key declaration. Even if such a check was in place, its validity could be compromised by editing or replacing the data file after table creation. There is no way to prevent such tampering with the file, but on the other hand, data integrity is always at risk if any database files are modified via non-standard methods. Therefore an external file has to be considered an integral part of the database system from the moment it is integrated into the database's system catalog until its removal via a standard drop table DML statement.

Having no direct access capabilities to individual tuples also impedes the creation of secondary indexes on file tables, because materialization is not feasible. Also, for supporting secondary indexes, an unambiguous tuple identifier is required, such as a primary key or an IK, to map every index tuple to exactly one tuple in the base relation. But neither primary keys nor direct access is available for file tables, and the employment of an IK table would require the storage of IK values with every tuple in the base relation, which is impossible since the relation is stored in a read-only file.

But file tables still possess one unique feature, which is not yet exploited. Even though files do not possess search structures for direct access to individual tuples, they still allow random access to their data. This capability can be used in two ways: The byte-offset of the beginning of every line is a unique numeric identifier for each tuple. This recognition finally enables creation of secondary indexes on file tables. Technically, these offsets are used as key surrogates, providing the identification and direct access functionality similar to IK values. As a specific peculiarity of file tables, these values do not have to be stored explicitly in the base relation, as they are embedded in the physical structure of the file. In addition to saving storage space, these physical tuple addresses also relieve the file table from maintaining dedicated IK tables, offering additional savings on storage and maintenance complexity over traditional indexing. But exploitable physical tuple addresses in file tables are depending on the file table's read-only restraint, as it guarantees constant tuple offsets. However, technically this approach can be sustained for append-only file tables and a similar approach is also feasible for flat tables, if these tables are operated in read-only or append-only mode.

The second consequence of random access to files is the possibility to perform binary searches on the data. For this, the original DDL statement has to be augmented with an ORDER BY specification in the file table's custom clause.

```
CREATE FILE(<filename> ORDER BY <FieldNames>) TABLE <tablename>
    (<FieldDefinitions>)
```

Similar to a primary key declaration, this specification represents an assertion of a data property, which can be exploited by the file table module for performing binary searches on its data, thereby permitting random access capabilities with logarithmic complexity. But since the primary scope of file tables is data import rather than data retrieval, neither secondary indexes nor binary search on file tables have been implemented yet.

Figure.45 compares the file table's behavior with the conventional storage structures B-tree and flat table. The latter structures need to be created with a DDL statement and populated via a spool process, before data becomes accessible. In this example, data is spooled from an external ASCII file (1.6 Gbyte) that will also serve as data basis for the file table. Hence, creation of these conventional structures involves reading the external data file once and construction of data access structures. In case of the B-tree, data is additionally sorted for rapid insertion. After the process concludes, both flat table and B-tree occupy approximately 1.5 Gbyte of the DBMS address space. At this point, the source file usually becomes obsolete and may be deleted. Even so, this load process has momentary storage requirements of 3.1 Gbyte, and in case of the B-tree an additional temporary storage area of up to 1.5 Gbyte is required for sorting. Creation of a file table on the other hand, does not move any data. The DDL statement only allocates one single page in the DBMS's internal storage area for its description page. During FTS operations, the conventional access structures perform significantly better than the file table, but if the file table is accessed only once, even this superior performance will not suffice for compensating for the expensive creation process. Only if data is likely to be accessed repeatedly, then internal storage structures are able to amortize themselves.

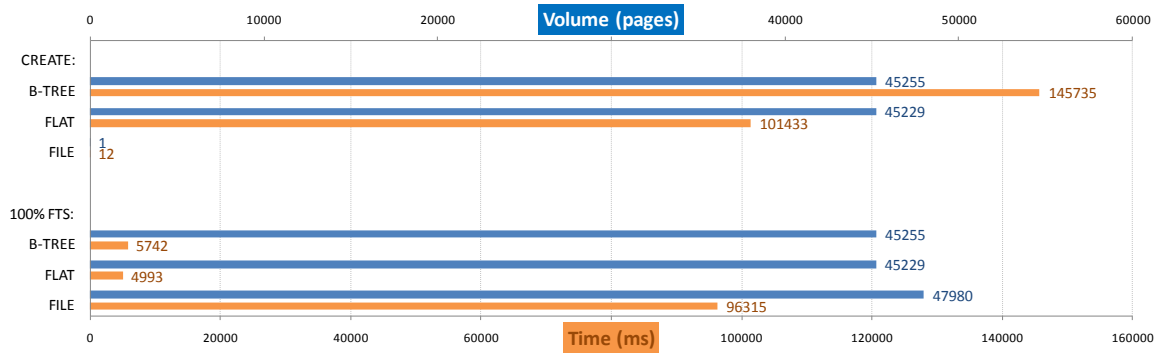


Figure.45 File table performance characteristics. This diagram compares conventional data import from an external file into internal B-tree and flat table structures with direct access to data residing in a file outside the database. In the latter case, data is retrieved via the *file table* data integration module. Evaluation times (orange) for the operation denoted on the left refer to the lower horizontal axis. The volume of data (blue) affected by this operation, plotted as the number of touched pages, where one page is 32 kbyte in size. B-tree and flat table occupy approximately 1.5 Gbyte each, while the file table in ASCII format uses almost 1.6 Gbyte on the file-system, which is equivalent to 47980 pages. Data volume refers to the upper horizontal axis. Note that values outside the range of the diagram are indicated by a broken bar.

Although full table scans on file tables are significantly slower than similar operations on internal structures, file tables represent an adequate instrument for alleviating and improving data import, as they allow to process external data with the full functional scope of SQL. This

makes it possible to load data, apply necessary transformations, and conduct on-the-fly integrity checks on the imported data in one single step.

As a consequence of these measurements, a general revision of the ASCII scanner implementation is planned, in order to clarify its apparently poor performance. But as possible improvements will affect spooling internal storage structures and reading external files equally, all previously made conclusions will remain valid. An alternative scanner/parser implementation using a binary data format will further diminish the present performance deficiency of the ASCII variant and it will also allow for more compact external files. Both ASCII and binary variants can be effectively augmented with additional data compression and data encryption.

5.7. Generic Functional Indexes

In its current state, Transbase offers no support for functional indexing, and index definition is only possible on the plain attribute values. To a certain degree, the UB-tree represents an exception to this rule, as it uses an internal mapping from plain attributes to a calculated search key. Still the UB-tree does qualify as a functional index in the classical sense, because externally all restrictions operate strictly on the plain representation of indexed attributes, and not on a functional mapping as defined by a selection $\sigma_{f(x)=c}$. Before functional indexing can be realized in Transbase, the host system must provide extensive support for such predicates. This involves extensions to the SQL compiler for formulating functional index definitions and it also necessitates precautions in the query optimizer module for deducing the applicability of an available functional index to a given predicate. As soon as the host system complies with these preliminaries, the Access Manager framework is able to support development of functional indexes, resulting in a dramatic reduction of the expected implementation effort. Therefore, we conceive functional indexes as lightweight alternative to intermediate access modules, where the main differences between two implementations are the functions providing necessary mappings. Many common tasks of an access module, like storage, retrieval, and manipulation are independent from the actual mapping, and therefore it is desirable to provide a generic, reusable implementation for these recurring tasks.

A possible solution will supply reusable functionality as an *intermediate access module*, which implements the complete access method interface, but allows for integration of two, optionally three, user-defined functions. This first function provides the actual mapping of input data to calculated values. These values are then stored in an auxiliary search structure,

e.g. a B-tree or any other available access module. The second function is responsible for translating a given predicate from a user query into one or more predicates to be applied to the auxiliary search structure. In some cases, the translation of search predicates might be inexact and retrieve a superset of qualifying data from the auxiliary search structure. As an example, consider bitmap indexes as functional indexes storing encoded bitmap chunks, where the underlying B-tree will always produce complete bitmap chunks. Then an optional third function is required, for post-filtering the intermediate result and delivering the exact result to the query processor for further evaluation.

```
CREATE FUNCTIONAL(<custom_spec>) INDEX <indexname> ON (<expression>);
```

Example:

```
CREATE FUNCTIONAL(SOUNDEX.DLL ON B-TREE) INDEX emp_sndx
    ON employees(soundex(name));

SELECT * FROM employees WHERE soundex(name)
    BETWEEN soundex('SMITH') AND soundex('SMYTHE');
```

Functional indexes represent a lightweight approach, allowing very fast adaptation of a universal DBMS towards an application domain. Only if the scope of functional indexes should not suffice for satisfying specific requirements, then resorting to implementation of an intermediate or full access module provides the necessary flexibility.

5.8. Main Memory DBMS

All presented search structures are designed for operating on large data sets residing on slow secondary storage devices. But today's computer hardware possesses gigabytes of fast main memory, and DBMSs are bound to incorporate these resources in effective query planning and adjusted query evaluation techniques. This cannot be accomplished by simply using bigger main memory caches, containing mere images of persistently stored data, but through reorganization of data for efficient access in main memory. Pure main memory database systems (MMDBs) and hybrid database systems therefore convert data that was retrieved from disk into dedicated main memory search structures, such as binary trees, heaps, hash tables, etc. The presented bitmap index implementation is an example for a hybrid access structure, as it organizes its bitmap chunks in compact compressed representation, which is appropriate for the DBMS's page-oriented cache and permanent storage, but at the same time it uses run-length encoded or inflated bitmaps with fast direct access capabilities. Finally,

other approaches are based on the observation that the volatile levels of today's memory hierarchies, i.e. main memory with its numerous caches, are in essence block I/O devices, just like hard disks. Even though access times to main memory are considerably faster, and it possesses random access capabilities, memory transfer is conducted in blocks of up to 512 bytes, depending on the next cache level's cache-line size. Therefore Chen et al. [Che01] argue, that B-trees are optimal main memory search structures, if their node size is adequately dimensioned or if an appropriate prefetch mechanism is installed for preventing cache-misses while navigation through the search structure. The Access Manager provides the necessary flexibility for experimentally investigating such assumptions in the context of a complete DBMS. Moreover, the bitmap example proves that the access method interface is capable of effective accommodation of memory-aware access modules.

5.9. Data Partitioning

Horizontal and vertical data partitioning are popular instruments for improving DBMS performance. Both allow physical distribution of one logical storage structure over several physical storage devices, thereby reducing average access times and enhancing data transfer rates. Data-dependant horizontal partitions also open new opportunities for parallel data processing, while vertical partitioning introduces additional degrees of freedom to physical data clustering. Preliminary experiments with both forms of partitioning, realized as intermediate access modules operating on several auxiliary B-trees, validate that the Access Manager offers appropriate flexibility for implementing partitioned access methods. Especially its modular approach allowing for quick and easy reuse of existing storage modules facilitated rapid implementation of these access module prototypes. Yet, at the time of this writing, these approaches have not been pursued any further than to a general study of feasibility.

6. Conclusion

This work presented the Access Manager framework as an approach for systematic extension of a universal host RDBMS with supplemental access method implementations and auxiliary algorithmic units for query evaluation. Its goal is to overcome and invert the prevalent status quo, where DBMS applications from a vast spectrum of specific application domains, like customer relationship management, business intelligence, e-commerce, engineering, scientific applications, etc., had to adapt to the peculiarities and functional capabilities of a standard DBMS, instead of customizing the DBMS to the applications' needs. DBMS customization inevitably involves the requirement of incorporating new, non-standard data types like video, images, audio, documents (structured and unstructured) and text data, but also handling of standard data with rich semantics, like spatial, temporal, genetic, ecological, meteorological or geological information, describing complex interrelations in huge data sets, and necessitating specialized access methods for efficient exploration. There is unquestioned need and potential for such DBMS extensibility, as [Gae97] describes over 50 alternative index structures for spatial indexing alone, while only a comparatively small number of access structures is enjoying a significant acceptance in common DBMS technology. A database system must be prepared to meet the particular requirements of an application domain, not only by providing tailored storage and retrieval functionality, but also by supporting analytical functionality resembling natural operations on the entities modeled in the database schema, eventually resulting in specialized forms of transformation, selection, aggregation, and other operations. The primary objective of the Access Manager framework is to obtain a generic solution for meeting customization requirements from diverse application domains. In addition, the Access Manager approach provides direct access to external data sources for importing data, or, if that is not possible, e.g. in a heterogeneous information infrastructure of a global enterprise or for WWW data, then it permits integration of complete external data repositories and combination of their particular search capabilities 'under one roof' of a common logical database schema.

The currently prevalent approach to DBMSs, where one manufacturer is providing a complete, monolithic system, is clearly not suitable for achieving the aspired goals, because the task is far too complex for a satisfactory one-size-fits-all solution. The resulting system would be cumbersome and overloaded with functionality, expensive because of surplus features, and difficult to maintain and administrate. Moreover, the presumable lack of expertise of one single manufacturer for addressing the huge field of possible application domains will inevit-

ably lead to insufficiencies in functionality and performance of the resulting system. Therefore an operational DBMS with basic functionality and optional piecemeal extensibility within a defined scope for embedding specialized functionality, to be provided by experts of a particular application domain, seems a far more promising approach. But this approach also has its pitfalls in form of possible functional and efficiency bottlenecks in the extensible infrastructure, or risk of compromised system integrity through undesirable side-effects between different extensions. It also may provoke a possible domino-effect, where extensibility of one DBMS module entails extensions to other modules, ultimately resulting in an overly complex extension interface.

6.1. Achievements

The Access Manager framework promotes modular DBMS extensibility and addresses potential pitfalls of this approach with effective countermeasures. Therefore it opens the architecture of an operational standard DBMS in a few selected areas, by introducing a concise yet flexible and powerful interface to the DBMS's components. This DBMS will function as a host system for accommodating application domain specific plug-ins, allowing well-defined adaptation and customization of the host DBMS by introduction of alternative primary and secondary access methods into the system, supported by custom algorithmic units implementing auxiliary operations, and a powerful data integration layer. This interface particularly emphasizes thorough integration of extension modules into the host system's intrinsic query optimization process, by devising a fully-automated, negotiation-based technique for constructing, transforming, and assessing efficient query evaluation plans containing external modules. This negotiation process promotes flexible substitution of partial query evaluation plans in algebraic representation with their implementations as encapsulated algorithmic units provided as opaque extension modules. It also allows an algorithmic unit to demand adaptation to be applied to its input for optimizing the algorithm's internal operations and minimizing its implementation complexity. On the other hand, negotiation permits configuration and exploitation of additional functionality an algorithmic unit provides beyond its primary purpose. Finally it allows propagation of basic relational transformations across opaque algorithmic units. All these activities aim for improving interoperability between independent algorithmic units, where each unit may encapsulate the functionality corresponding to an arbitrary algebraic expression. Every form of negotiation operates on the same uniform functional scope at the joints between autonomous algorithmic entities, such that the resulting

query evaluation plans are constructed from opaque algorithmic units that are bound together using a common set of functionally limited connectors.

Basic query evaluation through the Access Manager framework uses a very concise interface, thereby simplifying the development process of extension modules for elementary operations. But the framework is also adaptive to operations of higher complexity and capable of meeting performance challenges by provision of optional interface functionality. It particularly promotes development of extension modules in short iteration cycles, by encouraging modularity and reuse, and by providing supportive testing and debugging facilities. As a secondary goal, the framework protects the host system components from mutual intrusion. It cleanly separates autonomous extension modules and coordinates and supervises all interactions between these independent components and the host system. Thereby it safeguards the stability and consistency of the overall system and enforces data integrity by guaranteeing that all operations are carried out in a consistent way, within a transactional context, in strict adherence of a global concurrency control mechanism, and in accordance to present access privileges.

The interface functions are tied together using a comprehensible yet flexible protocol. In its basic scope, the protocol supports fundamental operations, such as query optimization and query evaluation, for controlling essential configuration, navigation, and manipulation capabilities of access method implementations. In addition to this rudimentary functionality, the protocol permits alternative sequences of basic interface calls and incorporation of optional interface functionality for achieving higher efficiency and for accomplishing more advanced concepts. This alternative protocol allows perfecting the functional capabilities of an otherwise complete access method implementation and maximizes its efficiency.

The Access Manager framework demonstrates its flexibility and universality in its ability to consistently integrate data residing in heterogeneous information systems outside the host DBMS's storage system into a homogeneous database schema, and thereby providing a uniform, location transparent view on distributed data. In this conception, the Access Manager framework accommodates pluggable extension modules, implementing access methods that function as gateways to other information systems. The access module wraps an external data repository and exposes only the Access Manager's abstract access method interface. The ability of this interface for participating actively in query optimization permits global optimization across a heterogeneous infrastructure. Efficient query evaluation is achieved through systematic exploitation of query evaluation capabilities of remote sites, and the Access Manager ensures data integrity by integrating the remote repository into its transactional context.

Practical experience with multiple access method implementations proves that the Access Manager approach is capable of substantiating its functional claims and its pretension for good performance with convincing results of available prototype implementations. Moreover, the introduction of the Access Manager into Transbase, as its host system, confirms that the additional layer does not induce significant costs. In this special case, the host system using the Access Manager even shows slight performance benefits over the original system, which can be attributed to the revision and unification of access method interfaces. Yet the most significant approval for this approach represents its almost completed transition from a mere research prototype to an intrinsic component of a productive DBMS.

In its present state, the Access Manager supports a remarkable subset of the extensive functionality that the academia postulated for a hypothetical CDBMS [Dit01]. Still there remain several interesting starting points to be pursued in future.

6.2. Future work

In accordance to the presented approach, the main focus for future work on the Access Manager is likely to address several query optimization problems. The most pressing of which is certainly the transfer from the current prototype, cooperating with a rule-based optimizer, to cost-based query optimization. Transbase's optimization heuristics are clearly apt for generating query evaluation plans of high quality. Moreover, rule-based optimization is often superior to cost-based optimization for queries of low or moderate complexity, where heuristics alone are sufficient for finding the optimal plan. Thereby, it allows evading all additional costs for collecting, maintaining and evaluating statistical information. However, rule-based optimization occasionally shows some vulnerability towards necessary code maintenance of the optimization module, or adaptations for new functionality. Integration of new heuristics into an extensive rule system is an intricate task, as it is likely to introduce conflicting rules or other unwanted side-effects. These effects are generally hard to predict and automated testing is often inappropriate for detecting them. Hence, cost-based validation of delicate rule-based decisions, and purely cost-based optimization for queries of high complexity, seems a promising approach. Moreover, it would open the unique possibility to confront the Access Manager framework with both optimization techniques at the same time.

Another interesting approach would examine the implications of integrating further unary ERA operations into the negotiation process. In particular, when dealing with data integration,

where fully-fledged DBMSs are serving as remote sites, then grouping and aggregation is a very attractive candidate for reducing the data volume to be transferred. A more thorough examination of the general problem of delegating joins on remote relations to the source repository is also desirable. Perhaps this leads to a universal concept for partitioning query execution plans into arbitrary fragments for distributed evaluation (cf. Garlic in [Haa97]). Alternatively, a further pursuit of the *remote view* approach, as sketched in *4.6.2 Data Integration Layer* (page 179), may open new query optimization opportunities.

Apart from query optimization, query evaluation is another interesting field for further improvements. The most promising concept for a future implementation is probably presented in *5.7 Generic Functional Indexes,* (page 218). This concept is able to lower implementation complexity for many application-domain indexes, since it represents a highly reusable framework that will effectively avoid repeated implementation of many reoccurring tasks when building functional indexes as *intermediate access modules*. Generic functional indexes stringently require a generalization of predicate evaluation in the host system, possibly involving new query optimization challenges for resolving functional dependencies. But if these preconditions are met, then a skeleton for generic functional indexes will be implemented as a mere pluggable access module, rather than as integral part of the Access Manager framework. Yet, this generic *intermediate access module* will possess a unique feature, allowing it to choose from all available *full access modules* the one implementation that is most appropriate for providing storage and retrieval functionality to a given functional index, thereby putting the interchangeability of access method implementations and the robustness of the Access Manager protocol to an ultimate test.

7. References

[Ack05] Ralph Acker, Roland Pieringer, and Rudolf Bayer, "Towards Truly Extensible Database Systems," in *Database and Expert Systems Applications (DEXA), 16th International Conference*, Copenhagen, Denmark, 2005, pp. 596-605.

[Ack08] Ralph Acker, Christian Roth, and Rudolf Bayer, "Parallel Query Processing in Databases on Multicore Architectures," in *8th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2008)*, Cyprus, 2008, pp. 2-13.

[ANSI99] American National Standards Institute (ANSI), "Information Technology - Database Langunages - SQL," American National Standards Institute (ANSI), ANSI/ISO/IEC 9075, 2008.

[Aok98] Paul M. Aoki, "Generalizing "Search" in Generalized Search Trees," in *ICDE*, Orlando, 1998, pp. 380-389.

[Bay72] Rudolf Bayer and Edward M. McCreight, "Organization and Maintenance of Large Ordered Indices," in *Acta Informatica*, 1972, pp. 173–189.

[Bay77a] Rudolf Bayer and Karl Unterauer, "Prefix B-Trees," *ACM Transactions on Database Systems (TODS)*, vol. 2, no. 1, pp. 11-26, 1977.

[Bay80] Rudolf Bayer, Hans Heller, and Angelika Reiser, "Parallelism and recovery in database systems," *ACM Transactions on Database Systems*, vol. 5, no. 2, June 1980.

[Bay96] Rudolf Bayer, "The Universal B-Tree for multidimensional Indexing," Technical University of Munich, Technical Report TUM-I9637, 1996.

[Car95] Michael J. Carey et al., "Towards Heterogeneous Multimedia Information Systems: The Garlic Approach," in *5th International Workshop on Research Issues in Data Engineering-Distributed Object Management (RIDE-DOM'95)*, Taipei, Taiwan , 1995, pp. 124-131.

[Cat00] R. G. G. Cattell and Douglas K. Barry, *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann, Ed., 2000.

[Cha98] Chee-Yong Chan and Yannis E. Ioannidis, "Bitmap Index Design and Evaluation," *SIGMOD Rec.*, vol. 27, no. 2, pp. 355-366, 1998.

[Cha99] Surajit Chaudhuri and Kyuseok Shim, "Optimization of Queries with User-defined Predicates," *ACM Transactions on Database Systems (TODS)*, vol. 24, no. 2, pp. 177 - 228, June 1999.

[Che01] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry, "Improving Index Performance through Prefetching," in *ACM SIGMOD Conference*, Santa Barbara, CA, USA, 2001, pp. 235-246.

[Cod70] Edgar Frank Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, June 1970.

[Com79] Douglas Comer, "The ubiquitous B-Tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121-138, 1979.

[Dit01] Klaus R. Dittrich and Andreas Geppert, Eds., *Component Database Systems*.: Morgan Kaufmann, 2001.

[Döl02] Mario Döller, "Enhancement of Oracle's Indexing Capabilities through GiST-implemented Access Methods," University Klagenfurt- ITEC Institute, Klagenfurt, TR/ITEC/02/2.09, 2002.

[Fen02] Robert Fenk, Volker Markl, and Rudolf Bayer, "Interval Processing with the UB-Tree," in *Proc. of IDEAS*, Edmonton, Canada, 2002, pp. 12-22.

[Fuh99] You-Chin Fuh, Stefan Dressloch, Weidong Chen, and Nelson Mendonca Mattos, "Implementation of SQL3 Structured Types with Inheritance and Value Substitutability.," in *VLDB*, Edinburgh, Scotland, 1999, pp. 565-574.

[Gae97] Volker Gaede and Oliver Gunther, "Multidimensional Access Methods," *ACM Computing Surveys*, vol. 30, pp. 170-231, 1997.

[GiST90] GiST Indexing Project, University of California, Berkeley. (1990) [Online]. http://gist.cs.berkeley.edu/

[Haa89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh, "Extensible Query Processing in Starburst," in *SIGMOD Conference*, Portland, 1989, pp. 377-388.

[Haa97] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang, "Optimizing queries across diverse data sources," in *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, 1997, pp. 276-285.

[Hel93] Joseph M. Hellerstein and Michael Stonebraker, "Predicate Migration: Optimizing Queries with Expensive Predicates," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, D.C., 1993, pp. 267-276.

[Hel95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer, "Generalized Search Trees for Database Systems," in *21th International Conference on Very Large Data Bases*, Zurich, Switzerland, September 1995, pp. 562-573.

[IBM02a] IBM Corp., "DB2 SQL Reference, Version 8," IBM Corp., CT17RNA, 2002.

[IBM02b] IBM Corp., "DB2 Spatial Extender User's Guide and Reference, Version 8," IBM Corp., CT19HNA, 2002.

[IBM03a] IBM Corp., "IBM Informix Virtual-Table Interface, Version 9.4," IBM Corp., CT1TDNA, 2003.

[IBM03b] IBM Corp., "IBM Informix Virtual Index Interface, Version 9.4," IBM Corp., CT1TCNA, 2003.

[Kar02] Nicos Karayannidis et al., "Processing Star Queries on Hierarchically-Clustered Fact Tables," in *28th International Conference on Very Large Data Bases, VLDB*, Hongkong, China, 2002, pp. 730-741.

[Kle03] Carsten Kleiner and Udo W. Lipeck, "OraGiST - How to Make User-Defined Indexing Become Usable and Useful," in *Database Systems for Business,*

Technology, and the Web, Leipzig, Germany, 2003, pp. 324–334.

[Kor00] Marcel Kornacker, "Access Methods for Next-Generation Database Systems," University of California, Berkley, Ph.D. Thesis 2000.

[Kor97] Marcel Kornacker, C. Mohan, and Joseph M. Hellerstein, "Concurrency and Recovery in Generalized Search Trees," in *SIGMOD*, Tucson, 1997, pp. 62-72.

[Kor99] Marcel Kornacker, "High-Performance Extensible Indexing," in *25th International Conference on Very Large Data Bases*, Edinburgh, Scotland, 1999, p. 699/708.

[Loh88] Guy M. Lohman, "Grammar-like Functional Rules for Representing Query Optimization Alternatives," in *SIGMOD Conference*, Chicago, 1988, pp. 18-27.

[Mar00] Volker Markl and Rudolf Bayer, "A Cost Function for Uniformly Partitioned UB-Trees," in *International Database Engineering and Applications Symposium, IDEAS*, Yokohama, Japan, 2000, pp. 410-417.

[Mar99a] Volker Markl, "Mistral: Processing Relational Queries using a Multidimensional Access Technique," Technische Universität München, Munich, Dissertation 1999.

[Mar99b] Volker Markl, Martin Zirkel, and Rudolf Bayer, "Processing Operations with Restrictions in RDBMS without External Sorting: The Tetris Algorithm.," in *ICDE 1999*, Sydney, Australia, 1999, pp. 562-571.

[Obj95] Object Management Group, "CORBAservices: Common Object Services Specification," 1995.

[Ora02] Oracle Corp., "Oracle9i Data Cartridge Developers Guide, Release 2 (9.2)," Oracle Corp., A96595-01, 2002.

[Ora03] Oracle Corp., "Oracle Database SQL Reference, 10g Release 1 (10.1)," Oracle Corp., B10759-01, 2003.

[Ora10] Oracle Corporation. (2010) MySQL. [Online]. http://www.mysql.com

[Pie01] Roland Pieringer, Volker Markl, Frank Ramsak, and Rudolf Bayer, "HINTA, A

Linearization Algorithm for Physical Clustering of Complex OLAP Queries," in *Design and Management of Data Warehouses, DMDW*, Interlaken, Switzerland, 2001, pp. 11-22.

[Pie03] Roland Pieringer, "Modeling and implementing multidimensional hierarchically structured data for data warehouses in relational database management systems and the implementation into Transbase," Fakultät für Informatik, Technische Universität München, Munich, Dissertation 2003.

[Pir92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan, "Extensible/Rule Based Query Rewrite Optimization in Starburst," in *SIGMOD Conference*, San Diego, 1992, pp. 39-48.

[Ram00] Frank Ramsak et al., "Integrating the UB-Tree into a Database System Kernel," in *Proceedings of the Conference on Very Large Data Bases, VLDB*, Cairo, Egypt, September 2000, pp. 263-272.

[Ram02] Frank Ramsak, "Towards a general-purpose, multidimensional index: Integration, Optimization, and Enhancement of UB-Trees," Fakultät für Informatik , Technische Universität München, 2002, München, Dissertation 2002.

[Rot04] Doron Rotem, Kurt Stockinger, and Kesheng Wu, "Efficient Binning for Bitmap Indices on High-Cardinality Attributes," Berkeley Lab, Berkeley, USA, Technical Report LBNL-56936, 2004.

[Sel79] Patricia Griffiths Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price, "Access path selection in a relational database management system," in *ACM SIGMOD International Conference on Management of Data*, Boston, Massachusetts, May 1979, pp. 23-34.

[Sto03] K. Stolze and T. Steinbach, "DB2 Index Extensions by example and in detail," *IBM Developer works DB2 library*, December 2003.

[Tor99] Mary Tork Roth, Fatma Ozcan, and Laura M. Haas, "Cost Models DO Matter: Providing Cost Information for Diverse Data Sources in a Federated System," in *VLDB'99, Proceedings of 25th International Conference on Very Large Data*

Bases, Edinburgh, Scotland, 1999, pp. 599-610.

[Tra10] Transaction Software GmbH. (2010) Transbase DBMS Documentation. [Online]. http://www.transaction.de/news/downloads/documentation

[Ube94] Michael Ubell, "The Montage Extensible DataBlade Achitecture," *ACM SIGMOD Record*, vol. 23, no. 2, p. 482, June 1994.

[Zir04] Martin Zirkel, "Evaluation of the UB-Tree with consideration of sorting," Technische Universität München, Munich, Dissertation 2004.

[Zir99] Martin Zirkel, Volker Markl, and Rudolf Bayer, "Exploitation of Pre-sortedness for Sorting in Query Processing: The TempTris-Algorithm for UB-Trees," in *IDEAS 2001*, Grenoble, France, 1999, pp. 155-166.

Appendices

Quick Reference

∀ n-ary $f, g \in$ ERA:

$$f \overline{\leq} g \Leftrightarrow \exists\, \mathcal{C}_O^{std}, \mathcal{C}_{I_1}^{std}, \ldots, \mathcal{C}_{I_n}^{std} \in \text{ERA}:\ \mathcal{C}_O^{std} \circ f \circ \left(\mathcal{C}_{I_1}^{std}, \ldots, \mathcal{C}_{I_n}^{std}\right) \equiv g \qquad 30$$

Corollary.2: **Equivalent implementation.** Supposing that $f, g \in$ ERA with $f \overline{\leq} g$, and there exists an algorithmic implementation $[f]$ and implementations $\left[\mathcal{C}^{std}\right] \in$ [ERA] for trary $\mathcal{C}^{std}$, then there also exists an *equivalent implementation* of all representatives of $\overline{g}$, because

$$\begin{aligned} g &\equiv \mathcal{C}_O^{std} \circ f \circ \left(\mathcal{C}_{I_1}^{std}, \ldots, \mathcal{C}_{I_n}^{std}\right) \\ &\equiv \left[\mathcal{C}_O^{std}\right] \circ [f] \circ \left(\left[\mathcal{C}_{I_1}^{std}\right], \ldots, \left[\mathcal{C}_{I_n}^{std}\right]\right) \end{aligned} \qquad 30$$

Definition.9: Sets of Connectors $\mathfrak{C}^{std}$ and $\mathfrak{C}$. A *standard connector* $\mathcal{C}_{\mathcal{S}}^{std}$ is an ERA expression composed of the basic unary ERA operations $(\pi_{\mathcal{S}}, \sigma_{\mathcal{S}}, \delta_{\mathcal{S}}, \tau_{\mathcal{S}}, id_{\chi_{std}})$, operating according to specification $\mathcal{S}$, but strictly in standard representation. The set of all standard connectors is $\mathfrak{C}^{std}$. A *generic connector* has the form $\mathcal{C}_{\mathcal{S}} = (\pi_{\mathcal{S}}, \sigma_{\mathcal{S}}, \delta_{\mathcal{S}}, \tau_{\mathcal{S}}, \chi_{\mathcal{S}})$, and the set of generic connectors is $\mathfrak{C}$. It follows that $\mathfrak{C}^{std} \subset \mathfrak{C} \subset$ ERA. 36

Definition.10: Applicability. We introduce three distinct qualities of applicability for arbitrary $f, g \in$ ERA:

(1) An arbitrary algorithm implementation $[f]$ is *strictly applicable* in a query evaluation plan, iff it possesses no input requirements, i.e.

$$f \circ g(R) \xrightarrow{apply\,[f]} [f] \circ g(R)$$

(2) $[f]$ is *regularly applicable*, iff its input requirements are satisfied by applying standard connectors $\mathcal{C}^{std} \in \mathfrak{C}^{std}$ to a given relational input. The operator '$\star$' denotes composition in the presence of non-trivial input requirements. The host system is capable of supplying *strictly applicable* implementations $[\mathcal{C}^{std}]$ for every $\mathcal{C}^{std} \in \mathfrak{C}^{std}$.

$$f \circ g(R) \xrightarrow{apply\,[f]} [f] \star g(R) \equiv [f] \circ [\mathcal{C}^{std}] \circ g(R)$$

(3) $[f]$ is *weakly applicable*, iff it has input requirements $\mathcal{C} \in \mathfrak{C} \backslash \mathfrak{C}^{std}$. The host system cannot provide an immediate implementation for $\mathcal{C}$.

$$f \circ g(R) \xrightarrow{apply\,[f]} [f] \star g(R) \equiv [f] \circ \mathcal{C} \circ g(R) \qquad 37$$

Definition.11: Apply function $\mathcal{A}$. For $f, g, h_1, \dots, h_n \in$ ERA, the algorithmic implementation $[f]$ is applied in a query plan as replacement for an n-ary f with the $\mathcal{A}^{[f]}$ function. The mapping of attribute positions in input stream tuples to the attribute references in f is provided as vector $\pi^{in} = \left(\pi_i^{in}\right)_{i=1,\dots,n}$. The function parameter $opt \in \mathbb{N}_0$ is an integer choosing an operational mode for f from an enumeration of available modes. Each opt settings effectuates different *optional input requirements* for f. The result of $\mathcal{A}^{[f]}$ are connectors $\left(\mathcal{C}_{\mathcal{A}_i:f_{opt}}\right)_{i=1,\dots,n}$ defining input requirements for each input stream. The formal definition of the $\mathcal{A}^{[f]}$ is:

$$\mathcal{A}^{[f]}: \{\pi\}^n \times \mathbb{N}_0 \to \mathbb{C}^n,$$

$$\mathcal{A}^{[f]}\left(\pi^{in}, opt\right) = \left(\mathcal{C}_{\mathcal{A}_i:f_{opt}}\right)_{i=1,\dots,n}:$$

$$\forall g, h_i \in \text{ERA}: g \circ f \circ (h_1, \dots, h_n) \xrightarrow{\mathcal{A}^{[f]}} g \circ \left[f_{opt}^{\pi^{in}}\right] \star (h_1, \dots, h_n)$$

$$\equiv g \circ \left[f_{opt}^{\pi^{in}}\right] \circ \left(\mathcal{C}_{\mathcal{A}_i:f_{opt}} \circ h_i\right)_{i=1,\dots,n}$$

where $[f_{opt}^{\pi^{in}}]$ is the implementation of f, configured to the current input mapping π^{in} and optional input requirement setting opt. 38

Corollary.3: **Coalescence and Decomposition.** Two arbitrary $\mathcal{C}_1, \mathcal{C}_2 \in \mathbb{C}$ can be coalesced into one single $\mathcal{C}_3 \in \mathbb{C}$, such that $\mathcal{C}_1 \circ \mathcal{C}_2 \equiv \mathcal{C}_3$. In particular, for any $\mathcal{C}_1^{std}, \mathcal{C}_2^{std} \in \mathbb{C}^{std}$ exists a $\mathcal{C}_3^{std} \in \mathbb{C}^{std}$, such that $\mathcal{C}_1^{std} \circ \mathcal{C}_2^{std} \equiv \mathcal{C}_3^{std}$. Decomposition describes the inverse operation. 40

Application & Exploitation: $op_{\mathcal{A}} \equiv op_{\mathcal{R}} \circ op_{\mathcal{E}}$ 42

$$\forall op_{\mathcal{A}}, op_{\mathcal{E}}, op_{\mathcal{R}}, op \in \{\pi, \sigma, \delta, \tau\} \begin{cases} op_{\mathcal{A}} = id \Leftrightarrow \text{no output specification is requested,} \\ \quad \Rightarrow op_{\mathcal{E}} = op_{\mathcal{R}} = id. \quad \text{(trivial case)} \\ op_{\mathcal{E}} = id \Leftrightarrow \text{requested output specification was rejected,} \\ \quad \Rightarrow op_{\mathcal{A}} = op_{\mathcal{R}} \neq id. \quad \text{(worst case)} \\ op_{\mathcal{R}} = id \Leftrightarrow \text{requested output specification was accepted,} \\ \quad \Rightarrow op_{\mathcal{A}} = op_{\mathcal{E}} \neq id. \quad \text{(ideal case)} \\ else \text{ requested output specification was partially accepted,} \\ \quad \Rightarrow id \notin \{op_{\mathcal{A}}, op_{\mathcal{E}}, op_{\mathcal{R}}\} \quad \text{(general case)} \end{cases} \quad 42$$

Definition.12: Exploitability. We introduce two distinct qualities of exploitability for arbitrary $[f], [g] \in$ [ERA]:

(1) An algorithmic implementation $[g]$ is *fully exploitable* towards a connector $\mathcal{C}_0$, iff $[g]$ allows integration of $\mathcal{C}_0$, such that

$$[f] \circ \mathcal{C}_0 \circ [g] \xrightarrow{exploit\,[g]} [f] \circ [\mathcal{C}_0 \circ g]$$

(2) An algorithmic implementation $[g]$ is *partially exploitable* towards a connector $\mathcal{C}_0$, iff $[g]$ allows decomposition of $\mathcal{C}_0 \equiv \mathcal{C}_1 \circ \mathcal{C}_2$, such that an implementation $[\mathcal{C}_1]$ exists in [ERA] and $\mathcal{C}_2$ can be integrated into $[g]$:

$$[f] \circ \mathcal{C}_0 \circ [g] \xrightarrow{exploit\,[g]} [f] \circ [\mathcal{C}_1] \circ [\mathcal{C}_2 \circ g] \qquad 44$$

Definition.13: Exploit function $\mathcal{E}$. Let $f, g \in$ ERA, implemented as $[f] \circ \mathcal{C}_{\mathcal{A}} \circ [g]$. The connector $\mathcal{C}_{\mathcal{A}} \in \mathfrak{C}$ represents coalesced transformations required for substitution and application of $[f]$ and $[g]$. The complexity of such a query evaluation plan can be reduced by integrating functionality from $\mathcal{C}_{\mathcal{A}}$ into $[g]$, using the $\mathcal{E}^{[g]}$ function. The result $\mathcal{C}_{\mathcal{R}:g}$ of the $\mathcal{E}^{[g]}$ function represents the part of $\mathcal{C}_{\mathcal{A}}$ that was rejected by $[g]$:

$\mathcal{E}^{[g]}: \mathfrak{C} \to \mathfrak{C}$,

$\mathcal{E}^{[g]}(\mathcal{C}_{\mathcal{A}}) = \mathcal{C}_{\mathcal{R}:g}$:

$$\forall f, g \in \text{ERA} \begin{cases} \mathcal{C}_{\mathcal{R}:g} = id \;\Leftrightarrow \text{fully exploitable} \\ \qquad \Longrightarrow [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \xrightarrow{\mathcal{E}^{[g]}} [f] \circ [\mathcal{C}_{\mathcal{A}} \circ g] \\ \mathcal{C}_{\mathcal{R}:g} = \mathcal{C}_{\mathcal{A}} \;\Leftrightarrow \text{not exploitable} \\ \qquad \Longrightarrow [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \xrightarrow{\mathcal{E}^{[g]}} [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \\ \text{else} \qquad \text{partially exploitable} \\ \qquad \Longrightarrow [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \xrightarrow{\mathcal{E}^{[g]}} [f] \circ \mathcal{C}_{\mathcal{R}:g} \circ \left[\mathcal{C}_{\mathcal{E}:g} \circ g\right] \end{cases} \qquad 45$$

Exploitation & Propagation: $\forall op \in \{\pi, \sigma, \delta, \tau\}: \; op_{\mathcal{A}} = op_{\mathcal{R}} \circ op_{\mathcal{E}} \circ op_{\mathcal{P}}$ 48

Definition.14: Propagation. We introduce two distinct qualities of propagation for arbitrary $[f], [g], [h] \in$ [ERA]:

(1) A connector $\mathcal{C}_0$ is *fully propagatable* through an algorithmic implementation $[g]$, iff a decomposition $\mathcal{C}_0 \equiv \mathcal{C}_1 \circ \mathcal{C}_2$ exists, such that

$$[f] \circ \mathcal{C}_0 \circ [g] \circ [h] \xrightarrow{propagate\,[g]} [f] \circ [\mathcal{C}_1 \circ g] \star [h]$$

$$\equiv [f] \circ [\mathcal{C}_1 \circ g] \circ \mathcal{C}_2 \circ [h]$$

(2) A connector $\mathcal{C}_0$ is *partially propagatable* through an algorithmic implementation $[g]$, iff $[g]$ allows decomposition of $\mathcal{C}_0 \equiv \mathcal{C}_1 \circ \mathcal{C}_2 \circ \mathcal{C}_3$, such that an implementation $[\mathcal{C}_1]$ exists in [ERA] and $\mathcal{C}_2$ can be integrated into $[g]$, i.e.

$$[f] \circ \mathcal{C}_0 \circ [g] \circ [h] \xrightarrow{propagate\ [g]} [f] \circ [\mathcal{C}_1] \circ [\mathcal{C}_2 \circ g] \star [h]$$
$$\equiv [f] \circ [\mathcal{C}_1] \circ [\mathcal{C}_2 \circ g] \circ \mathcal{C}_3 \circ [h] \qquad 48$$

Definition.15: Propagate function $\mathcal{P}$. Let $f, g, h \in$ ERA, implemented as $[f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \circ [h]$. The connector $\mathcal{C}_{\mathcal{A}} \in \mathfrak{C}$ represents coalesced transformations required for substitution and application of $[f]$ and $[g]$. The complexity of such a query evaluation plan can be reduced by propagating functionality from $\mathcal{C}_{\mathcal{A}}$ through $[g]$, using the function $\mathcal{P}^{[g]}$, where $\mathcal{C}_{\mathcal{R}:g}$ represents the part of $\mathcal{C}_{\mathcal{A}}$ that was rejected by $[g]$:

$$\mathcal{P}^{[g]}: \mathfrak{C} \to \mathfrak{C},$$
$$\mathcal{P}^{[g]}(\mathcal{C}_{\mathcal{A}}) = \mathcal{C}_{\mathcal{R}:g}:$$

$$\forall f, g, h \in \text{ERA} \begin{cases} \mathcal{C}_{\mathcal{R}:g} = id \ \Leftrightarrow \text{fully propagatable} \\ \quad \Longrightarrow [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \circ [h] \xrightarrow{\mathcal{P}^{[g]}} [f] \circ \left[\mathcal{C}_{\mathcal{E}:g} \circ g\right] \star [h] \\ \mathcal{C}_{\mathcal{R}:g} = \mathcal{C}_{\mathcal{A}} \ \Leftrightarrow \text{not propagatable} \\ \quad \Longrightarrow [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \circ [h] \xrightarrow{\mathcal{P}^{[g]}} [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \circ [h] \\ \text{else} \qquad \text{partially propagatable} \\ \quad \Longrightarrow [f] \circ \mathcal{C}_{\mathcal{A}} \circ [g] \circ [h] \xrightarrow{\mathcal{P}^{[g]}} [f] \circ \mathcal{C}_{\mathcal{R}:g} \circ \left[\mathcal{C}_{\mathcal{E}:g} \circ g\right] \star [h] \end{cases} \qquad 49$$

Definition.16: *Streaming* Cost Calculation. The cumulated costs of an *n-ary* algebraic expression, concluded by a *streaming* algorithmic implementation of g, calculate as:

$$\text{cost}_{\text{B}}([g] \circ ([f_1], \dots, [f_n])) = \sum_{i=1}^{n} \text{cost}_{\text{B}}([f_i])$$
$$\text{cost}_{\text{S}}([g] \circ ([f_1], \dots, [f_n])) = \text{cost}_{\text{L}}([g]) + \textstyle\sum_{i=1}^{n} q_i \cdot \text{cost}_{\text{S}}([f_i]) \qquad 62$$

Definition.17: *Blocking* Cost Calculation. The cumulated costs of an *n-ary* algebraic expression, concluded by a *blocking* algorithmic implementation of g, calculate as:

$$\text{cost}_{\text{B}}([g_n] \circ ([f_1], \dots, [f_n])) = \text{cost}_{\text{L}_{1\text{st}}}([g]) + \sum_{i=1}^{n} \text{cost}_{\text{B}}(f_i) + q_i \cdot \text{cost}_{\text{S}}(f_i)$$

$$\mathrm{cost}_{\mathrm{S}}([g_n] \circ ([f_1], \ldots, [f_n])) = \mathrm{cost}_{\mathrm{L}_{\mathrm{rest}}}([g]) \quad 62$$

List of Figures