Institut für Informatik
der Technischen Universität München

Bewertung des UB-Baums unter Berücksichtigung der Sortierung

Martin Zirkel

Institut für Informatik
der Technischen Universität München

Bewertung des UB-Baums unter Berücksichtigung der Sortierung

Dipl.-Inform. Univ. Martin Zirkel

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

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Martin Bichler

Prüfer der Dissertation:

1. Univ.-Prof. Rudolf Bayer, Ph. D.
2. Univ.-Prof. Alfons Kemper, Ph. D.

Die Dissertation wurde am 16.12.2003 bei der Technischen Universität München

eingereicht und durch die Fakultät für Informatik am 13.05.2004. angenommen.

Für Katja

Danksagung

Diese Arbeit entstand als wissenschaftlicher Mitarbeiter im Projekt Mistral bei FORWISS, dem bayerischen Forschungszentrum für wissensbasierte Systeme. Mein besonderer Dank gilt meinem Betreuer Herrn Professor Rudolf Bayer Ph.D. für die Anregungen und Diskussionen, die diese Arbeit entscheidend beeinflusst haben.
Bedanken möchte ich mich auch bei Herrn Professor Alfons Kemper Ph.D. für seine Bereitschaft das Zweitgutachten für diese Arbeit zu übernehmen.

Meinen Kolleginnen und Kollegen bei FORWISS und am Lehrstuhl von Herrn Prof. Bayer danke ich für die angenehme Arbeitsatmosphäre. Besonders bedanke ich mich bei meinen Kollegen im Projekt Mistral, Dr. Volker Markl, Dr. Frank Ramsak, Dr. Roland Pieringer und Robert Widhopf-Fenk, die mir in dieser Zeit zu guten Freunden geworden sind. Besonders danke ich ihnen für die zahllosen Diskussionen, die diese Arbeit beeinflusst haben, sowie für das Korrekturlesen.

Den ehemaligen Diplomanten Sebastian Hick, Wolfgang Klein, Sónia Antunes Pelizzari, Yiwen Lue und Jörg Lanzinger danke ich für Implementierungsarbeitein und Diskussionen, die zum Inhalt dieser Arbeit beigetragen haben.

Großen Dank auch an meinen Eltern Gudrun und Heinrich Zirkel, die mich immer wieder motivierten und anspornten.

Mein besonderer Dank gilt meiner Verlobten Katja Radtke, die mich während der gesamten Zeit meiner Promotion liebevoll durch Höhen und Tiefen begleitete. Besonders danke ich ihr für ihr Engagement und ihre Geduld meine umfangreiche Arbeit sorgfältig Korrektur zu lesen.

Saarbrücken 15. Juni 2004
Martin Zirkel

Titel der Doktorarbeit:

Bewertung des UB-Baums unter Berücksichtigung der Sortierung

Autor:

Dipl. Inform. Univ. Martin Zirkel

Schlüsselworte:

UB-Baum, mehrdimensionale Zugriffsmethoden, Sortierung, CUBE-Operator, Verbund-Operator, Gruppierung, Massenladen, Anfrageverarbeitung, Anfrageoptimierung, Datenstrukturen, B-Baum, Relationales Datenverwaltungssystem, Data-Warehouse, OLAP, Benchmark, TPC-H-Benchmark

Zusammenfassung:

Der Einsatz der mehrdimensionalen Zugriffsstruktur UB-Baum in komplexen Geschäfts-Anwendungen (z.B. SAP R/3), statistischen Datenbanken, Data-Warehouse und Data-Mining haben gezeigt, dass Leistungssteigerungen bis zum Faktor 10 für die Verarbeitung von Bereichsanfragen möglich sind. Weitere Verbesserungen wurden durch die Integration in den Datenbankkern Transbase von TransAction erzielt. Diese Arbeit untersucht die Möglichkeit der Ausnutzung der Z-Partitionierung des UB-Baums für die Verwendung in der Anfrageverarbeitung und beim Massenladen. Hierzu entwickelte der Autor mehrere neue Algorithmen, die entweder aus der Z-Ordnung effizient eine Zielordnung oder aus einer 1-dimensionalen Quellordnung die Z-Ordnung erzeugen. Dies führt zu einem effizienten Massenlade-Algorithmus sowie zu einem Verbunds- und Gruppierungs-Operator. Die Algorithmen reduzieren die E/A-Kosten in der Sortierphase um mindestens 50% gegenüber dem externen Sortieren. Für die einzelnen Algorithmen wurden Kostenmodelle entwickelt, die durch Leistungsmessungen auf künstlichen Datenverteilungen und durch Teile des TPC-H Benchmark belegt werden. Um die Anwendungstauglichkeit im realen Datenbankumfeld zu zeigen, wurden Messungen auf dem Data-Warehouse der Gesellschaft für Konsumforschung (GfK) gemacht. Die Arbeit schließt mit einer neuen Verarbeitungstechnik des CUBE-Operators und dem Einfluss der neuen Techniken auf die relationale Algebra. Der Autor erstellte die Ergebnisse dieser Arbeit als Mitglied der Mistralgruppe bei FORWISS.

Inhaltsverzeichnis

Man can learn nothing except by going from the known to the unknown.

***Claude Bernard** (1813–1878), French physiologist.*

1 Einführung

1.1 Motivation

Geht man durch die Geschäftsräume großer Firmen und hört den Gesprächen des Managements zu, wird man immer wieder mit folgenden Themen konfrontiert:

- Wir besitzen riesige Mengen an Informationen (Daten), kommen jedoch an die wichtigen Informationen nicht heran.
- Wir möchten die Daten unter beliebigen Aspekten betrachten.

Firmen, die weltweit operieren, wie z.B. Henkel, NEC oder Microsoft, benötigen aussagekräftige Informationen über die Marktsituation in den einzelnen Ländern, um profitable Produkte zu erzeugen. Hierzu ziehen diese Firmen heutzutage Entscheidungs-Unterstützungs-Systeme (Decision Support Systems, DSS) heran, um die Flut von Informationen auszuwerten. Der Begriff Online-Analyse-Verarbeitung (On-Line-Analytic-Processing, OLAP) beschreibt eine dimensionenorientierte Methode für DSS. OLAP-Systeme bieten Analytikern eine schnelle, konsistente und interaktive Zugriffsmethode, um Informationen, die aus Rohdaten der einzelnen Informationsquellen gewonnen wurden, aus verschiedenen Perspektiven zu betrachten.

Eine Kerntechnologie in Decision Support Systems ist das Konzept des Data-Warehouse (DW). Hierbei betrachten wir DWs, die nach [Kim96] als „dimensional data warehouse“ bezeichnet werden, d.h. die Daten werden bezüglich Dimensionen, wie z.B. „Zeit“, „Kunde“ und Produkt“ organisiert. DWs benötigen eine effiziente Anfrageverarbeitung, um die riesigen Datenmengen zu verarbeiten. DWs werden üblicherweise von „Relationalen Datenbanksystemen“ verwaltet.

Heutige Systeme haben keine Probleme riesige Datenmengen zu verwalten, solange es sich um kurze Transaktionen handelt, d.h. Einfügen (insert), Lesen (read), Löschen (delete) und Aktualisierungen (updates) auf kleinen Datenmengen (Tupelmengen). Diese Klasse wird in der Literatur als On-Line-Transaction-Processing (OLTP) bezeichnet. In OLTP-Systemen werden z.B. Bestellungen oder Banktransaktionen verarbeitet. Die Aufgaben sind wohl strukturiert und werden ständig wiederholt. Daneben handelt es sich üblicherweise um kurze, atomare und lokale Transaktionen, die allgemein unter dem Begriff ACID Transaktion [Gra78] bekannt sind.

OLAP-Systeme, die auf DWs basieren, haben ein vollkommen anderes Anfrageprofil als OLTP-Systeme. Hier dominieren der lesende Zugriff und speziell die Bereichsanfrage sowie die Berechnung von Kennzahlen aus riesigen Datenbeständen. Die Daten werden

üblicherweise aus den verschiedenen Datenquellen eines Konzerns in eine konsolidierte Form gebracht und periodisch z.B. wöchentlich durch Ladevorgänge ins System integriert.

Um diese Anfragen zu beschleunigen, werden Hilfsstrukturen - in der Literatur als Indexe bezeichnet - herangezogen. Indexe basieren auf speziellen Datenstrukturen, die der Anwendung einen wahlfreien effizienten Zugriff auf die Daten ermöglichen. Einige Indexe unterstützen neben Punktanfragen auch Bereichsanfragen, so dass z.B. alle Kunden einer Firma, deren Namen mit Z anfangen, effizient verarbeitet werden können. Der Einsatz von Datenstrukturen kann somit die Antwortzeit von Systemen extrem beschleunigen. Für DWs mit ihrem mehr-dimensionalen Modell hat sich der Einsatz von mehrdimensionalen Zugriffsmethoden bewährt.

Diese Arbeit verwendet UB-Bäume [Bay96] als mehrdimensionale Zugriffsmethode. Basierend auf dem UB-Baum werden effiziente Algorithmen für Bereichsanfragen in Kombination von Sortieren, Massenladen und Gruppierung entwickelt und analysiert. Diese Arbeit behandelt vor allem den Einsatz dieser Technologie im DW. Da Punktanfragen die Kernmetrik für OLTP-Systeme sind, werden sie in dieser Arbeit nicht weiter behandelt. Für grundlegende Aussagen über das Verhalten des UB-Baums für Punktanfragen verweisen wir auf [Mar99] und [FriM97].

1.2 Decision Support System

OLAP-Systeme werden üblicherweise als Mehrbenutzersysteme auf einer Client/Server-Architektur implementiert. Die Antwortzeit soll möglichst kurz sein, ganz gleich, wie groß die Datenmenge oder die Komplexität ist. Eine kurze Antwortzeit ist somit das primäre Ziel.

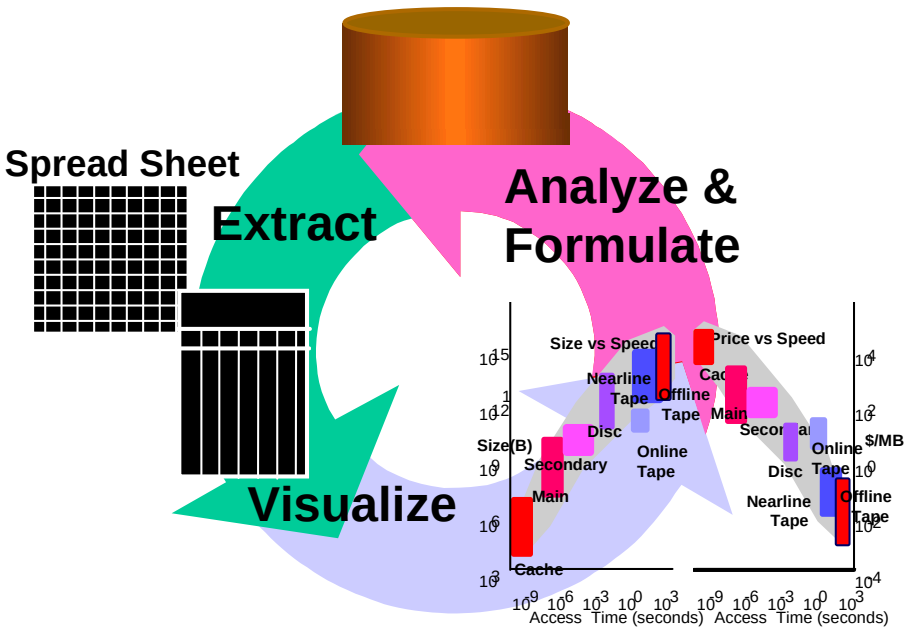


Abbildung 1-1: Der OLAP Prozess nach [GraCB+97]

Die Struktur dieser Anwendungen kann wie folgt beschrieben werden [GraCB+97]:

- Formulieren einer Anfrage, die die relevanten Daten aus einer großen Datenbank extrahiert.
- Darstellung der Ergebnisse in einer graphischen Anwendung, wie z.B. Excel.
- Analyse des Ergebnisses und, basierend auf dem Ergebnis, Stellen einer neuen Anfrage.

Visualisierungswerkzeuge, wie das MDX Sample (siehe Abbildung 1-2) von Microsoft, benutzen ein mehrdimensionales Modell (MDM), um komplexe Analysen zu visualisieren und somit Tendenzen und Aussagen für den Benutzer besser greifbar zu machen.

Ein Teil dieser Arbeit beschäftigt sich mit der Einsatzmöglichkeit multidimensionaler Zugriffsstrukturen im Datenbanksystem (üblicherweise relational) sowie mit dem Vergleich kommerziell verfügbarer Methoden in diesem Bereich.

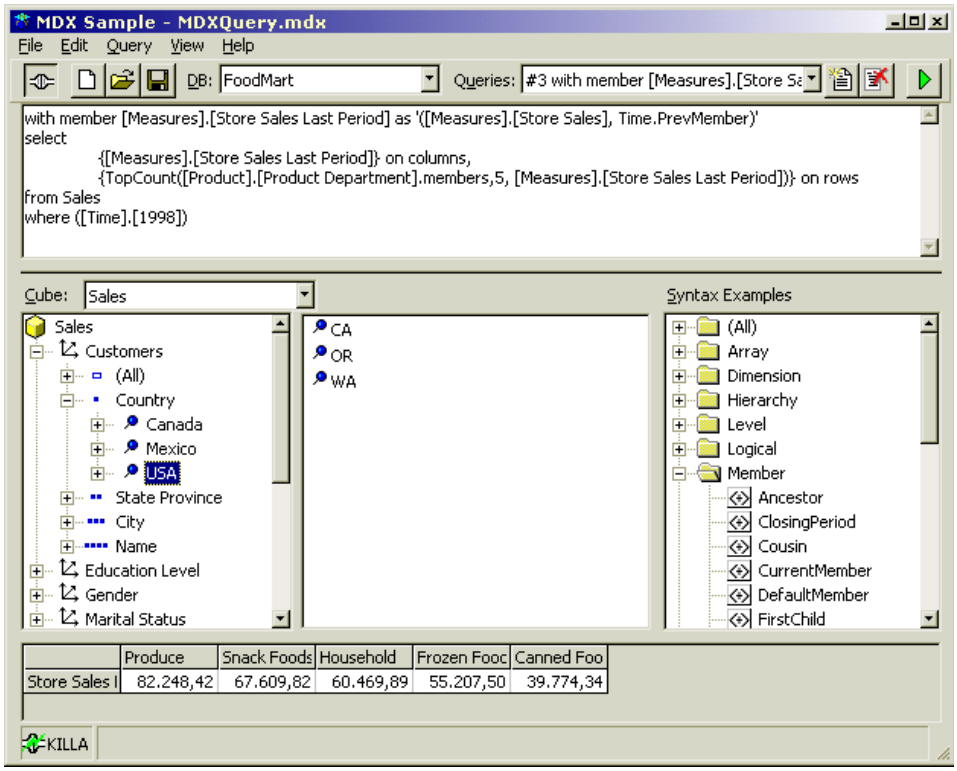


Abbildung 1-2 Datenanalyse im DW FoodMart

Beispiel 1-1: Datenanalyse in der Anwendung DW FoodMart
Abbildung 1-2 zeigt eine typische DW-Anwendung. Im oberen Teilfenster ist das MDX[1]-Anfragefenster zu sehen. Diese Anfrage berechnet die Umsätze aus dem Verkaufswürfel für die Top-Fünf-Produkte der letzten Periode. Das Ergebnis wird in Form eines Tabellenblatts einer Tabellenkalkulation im unteren Fenster dargestellt.

DWs verarbeiten historische, aggregierte und konsolidierte Daten im Gegensatz zu detaillierten, einzelnen Datensätzen. Da DWs im Allgemeinen Daten aus verschiedenen

[1] MDX ist eine von Microsoft entwickelte Anfragesprache für OLAP

Datenbanksystemen über sehr lange Zeitperioden als Datenquellen heranziehen, ist deren Platzbedarf gegenüber operationalen Datenbanksystemen um mehrere Größenordungen gestiegen.

Die Fakten (Kennzahlen, numerische Daten) vom Typ Verkäufe, Kosten, etc., auf denen das Augenmerk der Analyse liegt, werden bezüglich verschiedener Dimensionen organisiert. Eine Dimension besteht aus kategorisierenden Daten (z.B. Containergröße eines Produktes, Bayern für die geographische Kategorisierung von Kunden), die den Kontext der Kennzahlen bestimmen. Sie beschreiben die mögliche Sicht des Anwenders zu den assoziierten betriebswirtschaftlichen Kennzahlen.

Die Dimensionen spannen einen n-dimensionalen Raum auf. Die einzelnen Punkte des Raumes bezeichnet man als Zelle, deren Koordinaten als Zelladresse. Die Dimensionen zusammen mit den Fakten bilden den Datenwürfel (data cube).

Abbildung 1-3 zeigt einen 3-dimensionalen Datenwürfel mit den Dimensionen *Time*, *Produkt* und *Segment*. Hierbei handelt es sich um ein DW der GfK(Gesellschaft für Konsumforschung)-Gruppe.

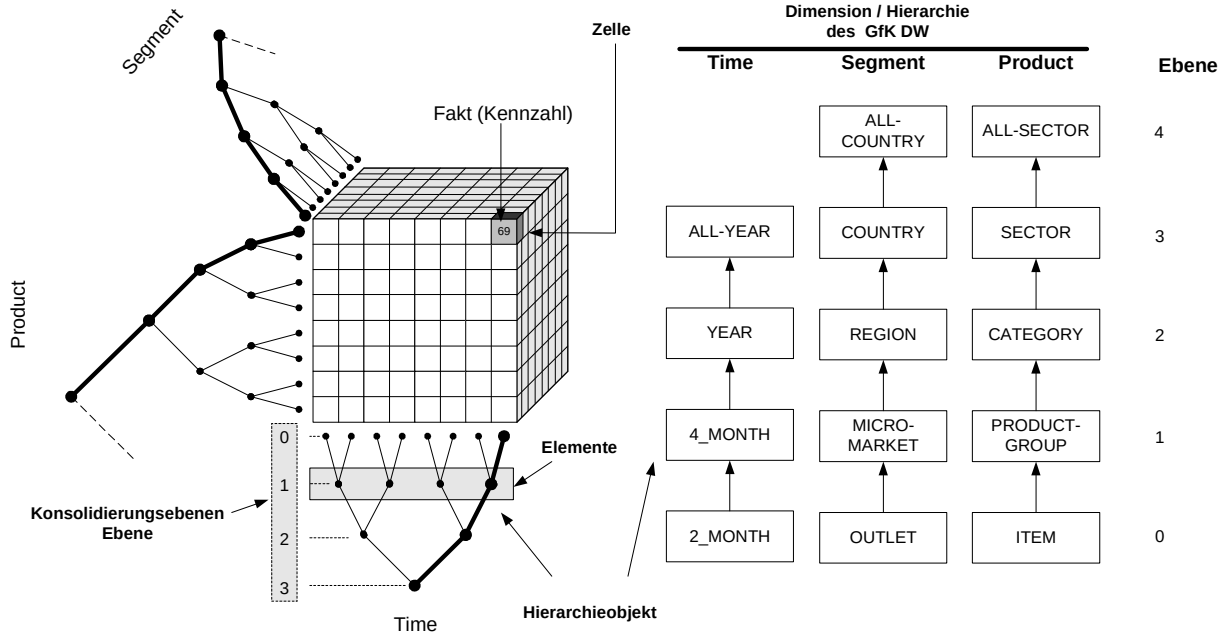


Abbildung 1-3: GfK Data Warehouse

Ein wichtiges Konzept im Bereich OLAP-Datenmodelle ist das Konstrukt der *Dimensionshierarchie*. Hierarchien werden benutzt, um Dimensionen zu strukturieren. Falls eine Dimension keine Hierarchiestruktur besitzt, bezeichnet man sie allgemein als *flach*. Üblicherweise können die Daten innerhalb einer Dimension in *Hierarchieobjekte* zusammengefasst werden, die untereinander in einer semantischen Beziehung stehen. Sie repräsentieren unterschiedliche Konsolidierungsebenen der assoziierten betriebswirtschaftlichen Kennzahlen. So sind z.B. *Monat*, *Tag*, *Jahr* und *Quartal* übliche Hierarchieobjekte der Dimension *Zeit*. Ein *Hierarchieobjekt* besteht aus einer variablen

Menge von Elementen bzw. Ausprägungen. So besteht z.B. das Hierarchieobjekt *Country* der Dimension Segment aus den Elementen *Deutschland, England, Italien* usw.. Diese Elemente sind eindeutig dem Hierarchieobjekt *Country* zugeordnet und können somit nicht weiteren Hierarchieobjekten zugeordnet sein. Allgemein bilden die Hierarchieobjekte einer Dimension einen (gerichteten) *nicht-zyklischen* Graphen. Auf der höchsten Konsolidierungsebene liegt immer das ALL-Hierarchieobjekt, das aus genau einer Ausprägung besteht z.B. ALL-YEAR. Einzelne zusammenhängende Pfade zwischen dem ALL-Knoten und der Konsolidierungsebene 0 durch den azyklischen Graphen werden als Aggregationspfad bezeichnet. Bestehen zwischen den Hierarchieobjekten einer Dimension nur 1:1 Beziehungen sprechen wir von einer *einfachen Hierarchie*, ansonsten von einer *komplexen* Hierarchie. Die in Abbildung 1-3 dargestellten Dimensionen *Time, Segment, Product* besitzen *einfache Hierarchien*. Da Anwender von OLAP-Systemen an Sichten auf unterschiedliche Konsolidierungsebenen interessiert sind, stellen *Hierarchien* eine geeignete Methode dar, um verschiedene Sichten auf unterschiedlichen Ebenen auf die Daten zu spezifizieren [Kur99].

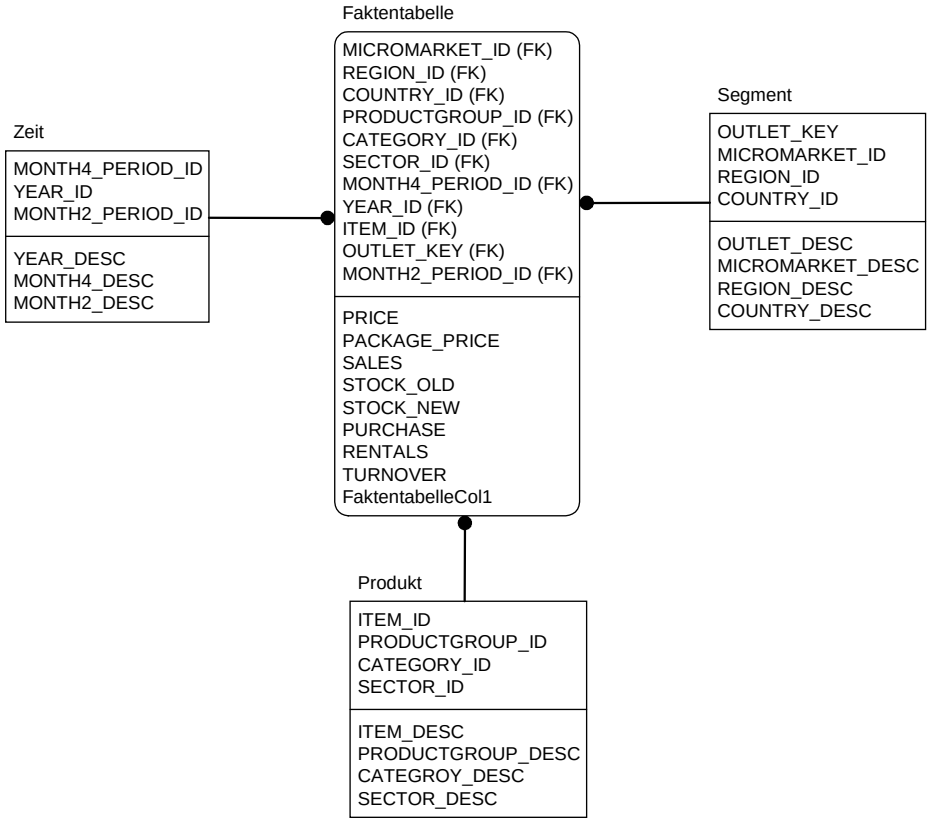


Abbildung 1-4: GfK Star-Schema[2]

Das mehrdimensionale Modell für DW-Systeme wird üblicherweise auf einem relationalen Datenbanksystem implementiert. Dazu wird in der relationalen Welt das mehrdimensionale Modell auf ein Star-Schema abgebildet. Die Datenbank besteht dann aus einer einzigen Faktentabelle und einer Tabelle für jede Dimension. Jedes Tupel der Faktentabelle enthält einen Fremdschlüssel zu jeder Dimensionstabelle. Falls eine Dimensionstabelle in eine Menge von Tabellen aufgeteilt wird, um die Normalisierung zu erfüllen, spricht man vom *Schneeflocken-Schema* (snowflake schema).

[2] IDEF1X Notation

Über die Hierarchien der einzelnen Dimensionen kann nun ein Benutzer verschiedene Sichten auf den Datenwürfel erstellen. Dazu stehen in OLAP-Systemen die Operationen *Slice*, *Dice*, *Rollup* und *Drill down* zur Verfügung.

Slicing (Schneiden)

Die *Slice*-Operation reduziert die Dimensionalität eines Datenwürfels (siehe Abbildung 1-5). Sie kann in SQL durch eine Projektion in Kombination mit einer Gruppierung auf den Dimensionen D_1, D_2, D_3, ... , D_n abgebildet werden.

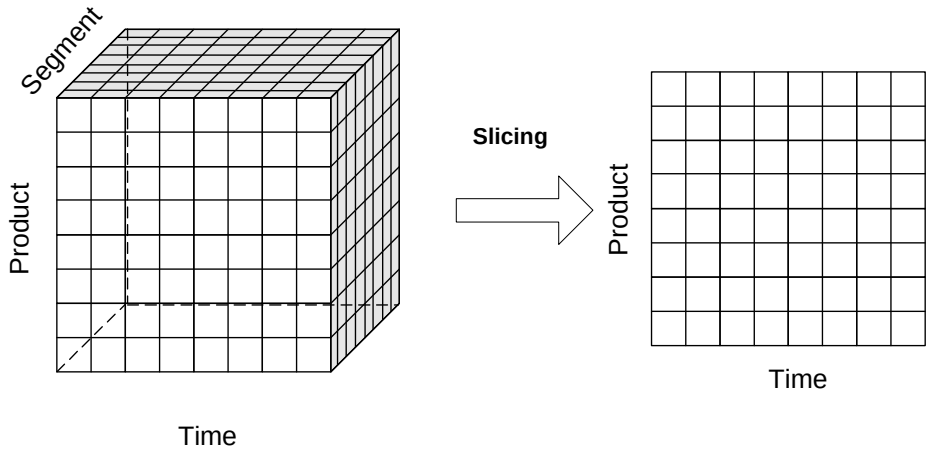


Abbildung 1-5: Slicing

Wendet man z.B. die *Slice*-Operation auf den GfK-Würfel an, um ihn um die Dimension *Segment* zu reduzieren, erhält man ein 2-dimensionales Ergebnis mit den Dimensionen *Produkt* und *Zeit* (siehe Abbildung 1-5). Die äquivalente SQL-Anweisung[3] hat dann folgende Form:

```
SELECT      YEAR_ID, 4_MONTH_ID,2_MONTH_ID,SECTOR_ID,
            CATEGORY_ID, PRODUCT-GORUP_ID, ITEM_ID,
            sum(ITEM_SOLD), sum(TURNOVER),…
FROM        FACT_TABLE
WHERE  OUTLET_ID = "Kaufhof Marienplatz"
GROUP BY    YEAR_ID, 4_MONTH_ID,2_MONTH_ID, SECTOR_ID,
            CATEGORY_ID,PRODUCT-GORUP_ID,ITEM_ID
```

Abbildung 1-6: *Slice-Operation*

Zu beachten ist hier jedoch, dass diese SQL-Anweisung nur die Aggregate auf der untersten Ebene berechnet. Will man alle Konsolidierungsebenen auf einmal berechnen, müssen die Konsolidierungsstufen der einzelnen Hierarchien sowie ihre Kombination zu den anderen Dimensionen berechnet werden, d.h. für jede Hierarchieobjektkombination muss das entsprechende Aggregat berechnet werden. Sind nicht alle Hierarchieobjekte eines Cubes in der GROUP-BY-Klausel eines Aggregats enthalten, bezeichnet man das Aggregat als Superaggregat.

[3] Die Faktentabelle *Fakt_TABLE* ist eine nicht normalisierte Tabelle, die sowohl aus den Elementen der Dimension Time, Segment und Product als auch aus den Fakten besteht. Es handelt sich hier nicht um ein Star-Schema.

Da man im mehrdimensionalen Datenmodell davon ausgeht, dass die Dimensionen orthogonal, d.h. voneinander unabhängig, sind, ergibt sich die Anzahl der Gruppierungen aus dem Produkt der Kardinalitäten der Hierarchieobjekte der beteiligten Hierarchien. Bei diesem Beispiel hat die Zeithierarchie die Kardinalität $|H_{time}| = 4$ und Produkt die Kardinalität $|H_{product}| = 5$. Somit müssen bereits 20 Gruppierungen mit den entsprechenden Aggregationsfunktionen berechnet werden. Lässt man alle Kombinationen zu, muss sogar die Potenzmenge berechnet werden, also 2^9=512. Die einzelnen Teilergebnisse werden dann mit Hilfe der UNION-Klausel verbunden.

Zur relationalen Darstellung der Superaggregate wird für die Gruppierungsattribute, über die in der jeweiligen Kombination aggregiert wird, das reservierte Schlüsselwort „ALL“ eingeführt. Dadurch wird im Sinne einer „Normalisierung“ eine relationale Rekonstruktion einer statischen Kreuztabelle („cross table“) erreicht. Dabei werden die Teil- und Gesamtsummen durch das Schlüsselwort „ALL“ als Ausprägung des Hierarchieobjektes adressiert [Leh98].

```
      SELECT  YEAR,4_MONTH,2_MONTH, SECTOR, CATEGORY,
            PRODUCT-GROUP,ITEM, sum(ITEM_SOLD),
            sum(TURNOVER),…
      FROM  FACT_TABLE
      GROUP BY YEAR,4_MONTH,2_MONTH, SECTOR, CATEGORY,
            PRODUCT-GROUP,ITEM
UNION
      SELECT   YEAR,4_MONTH,'ALL', SECTOR, CATEGORY,
            PRODUCT-GROUP, sum(ITEM_SOLD), sum(TURNOVER),…
      FROM     FACT_TABLE
      GROUP BY YEAR,4_MONTH, SECTOR, CATEGORY,
            PRODUCT-GROUP
UNION
      SELECT   YEAR,'ALL','ALL', SECTOR, CATEGORY,
            PRODUCT-GROUP, sum(ITEM_SOLD), sum(TURNOVER),…
      FROM     FACT_TABLE
      GROUP BY YEAR, SECTOR, CATEGORY,
            PRODUCT-GROUP
UNION
      …
```

Abbildung 1-7: Datenwürfel als Menge von UNIONS

Abbildung 1-7 zeigt einen Teil der SQL-Anweisung, um einen Slice aus einem Datenwürfel *FACT_TABLE* mit all seien Konsolidierungsebenen zu berechnen.

Würfeln (Dicing)

Die *Dice-Operation* schneidet einen Unterwürfel aus der Faktentabelle, wobei die Dimensionalität der Basiswürfel erhalten bleibt (siehe Abbildung 1-8).

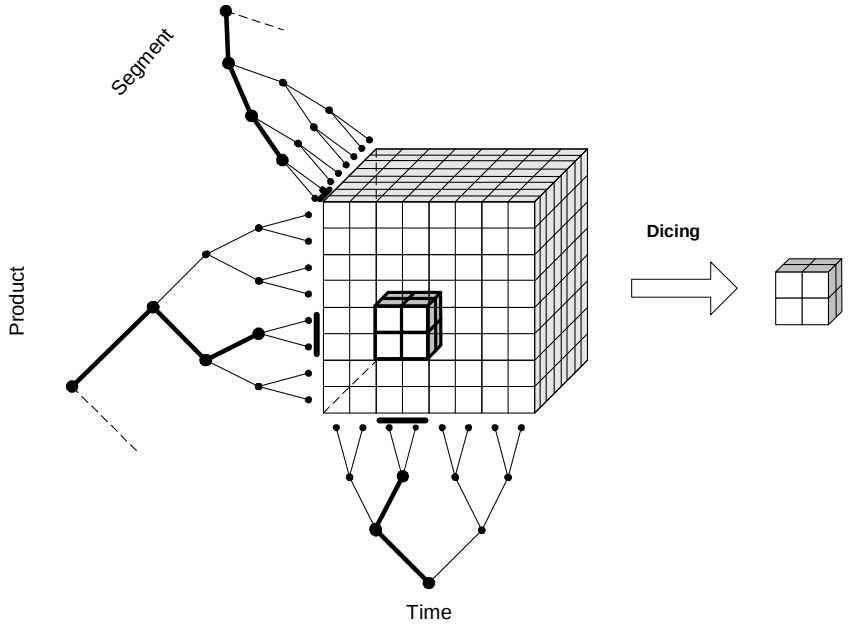


Abbildung 1-8: Dicing

Ein Unterwürfel kann anhand einer mehrdimensionalen Bereichsanfrage spezifiziert werden. Diese hat in SQL die folgende Form:

```
        SELECT      YEAR_ID,4_MONTH_ID,2_MONTH_ID,
                    SECTOR_ID, CATEGORY_ID,
                    PRODUCT-GROUP_ID,ITEM_ID, COUNTRY_ID,
                    REGION_ID, MICROMARKET_ID
                    OUTLET_ID, sum(ITEM_SOLD),
                    sum(TURNOVER),…
        FROM        FACT_TABLE
        WHERE       YEAR = 1999 AND
                    PRODUCT_GROUP = 'TV' AND
                    MICROMARKET = 'SÜD_BAYERN'
        GROUP BY    YEAR_ID,4_MONTH_ID,2_MONTH_ID,
                    SECTOR_ID, CATEGORY_ID,
                    PRODUCT-GROUP_ID,ITEM_ID, COUNTRY_ID,
                    REGION_ID, MICROMARKET_ID,OUTLET_ID
UNION
        SELECT      YEAR_ID,4_MONTH_ID,ALL, SECTOR_ID,
                    CATEGORY_ID,
                    PRODUCT-GROUP_ID,ITEM_ID, COUNTRY_ID,
                    REGION_ID, MICROMARKET_ID
                    OUTLET_ID,sum(ITEM_SOLD),sum(TURNOVER),.
        FROM        FACT_TABLE
        WHERE       YEAR = 1999 AND
                    PRODUCT_GROUP = 'TV' AND
                    MICROMARKET = 'SÜD_BAYERN'

        GROUP BY    YEAR_ID,4_MONTH_ID, SECTOR_ID,
                    CATEGORY_ID,
                    PRODUCT-GROUP_ID,ITEM_ID, COUNTRY_ID,
                    REGION_ID, MICROMARKET_ID
                    OUTLET_ID
UNION …
```

Abbildung 1-9: Dice Operation in SQL

Rollup und Drilldown

Eine weitere übliche Operation im DW ist *Rollup*, die inverse Operation bezeichnet man als *Drilldown*. Rollup bezeichnet die Operation, den Konsolidierungsgrad bezüglich einer Dimension zu vergrößern, d.h., von einer niedrigen Konsolidierungsstufe wie z. B. *2_Month* zu einer höheren Konsolidierungsstufe, wie etwa Jahr, überzugehen. Um die Rollup-Operation in DW besser zu unterstützen, wurde z.B. in das relationale Datenbanksystem von IBM und Microsoft die Rollup-Klausel integriert. Um die Kennzahl *Umsätze* für alle Produkte und Segmente des GfK DW bezüglich der Zeithierarchie zu berechnen, werden vier Gruppierungen benötigt. Benutzt man die Rollup-Klausel, kann man die SQL-Anweisung wie folgt schreiben:

```
SELECT ALL_YEAR, YEAR, 4_MONTH, 2_MONTH, sum(TURNOVER)
FROM   FACT_TABLE
WHERE OUTLET      = 'MARIENPLATZ' AND
      ITEM        = 'VIDEO'
GROUP  BY ROLLUP (ALL_YEAR, YEAR, 4_MONTH, 2_MONTH)
```

Abbildung 1-10: Rollup-Anweisung

Abbildung 1-10 erzeugt also folgende Gruppierungen:

```
1)   ALL_YEARS
2)   ALL_YEARS, YEAR
3)   ALL_YEARS, YEAR, 4_MONTH
4)   ALL_YEARS, YEAR, 4_MONTH, 2_MONTH
```

CUBE-Operator

Neben dem bereits vorgestellten Rollup-Operator wurden von den Datenbankherstellern, wie IBM und Microsoft, weitere Prädikate in SQL integriert, mit denen die Formulierung derartiger Anfragen vereinfacht werden. Sie werden als *Super-Gruppierungen* (*super groups*) bezeichnet, mit denen es möglich ist, mehr als eine Gruppierung in einer einzigen SQL-Anweisung berechnen zu lassen. Hier sind die Konstrukte der Gruppierungsmengen (*GROUPING SET*) und die CUBE Anweisung zu nennen. Die CUBE-Anweisung berechnet aus den Gruppierungsattributen alle möglichen Gruppierungen, d.h. die Potenzmenge $2^{|\text{Attribute}|}$ (siehe Abbildung 1-11).

```
SELECT ALL_YEAR,YEAR,4_MONTH,2_MONTH,
ALL_SECTOR,SECTOR, CATEGORY,PRODUCT_GROUP, ITEM
sum(ITEM_SOLD), sum(TURNOVER)
FROM   FACT_TABELE
GROUP BY CUBE (ALL_YEAR,YEAR,4_MONTH,2_MONTH,
PRODUCT_GROUP, ITEM,
ALL_SECTOR,SECTOR, CATEGORY);
```

Abbildung 1-11: CUBE-Klausel

Grundoperationen im DW

Die oben beschriebenen DW-Operationen basieren im Wesentlichen auf den folgenden zwei Grundoperationen: *mehrdimensionale Einschränkungen, Gruppierungen (super groups).*

In [MarZB99] wird gezeigt, wie man mit Hilfe der UB-Baum-Technologie mehrdimensionale Bereichsanfragen effizient abarbeiten kann. Hierbei werden die semantischen Hierarchien der Hierarchieobjekte für die physikalische Clusterung der Faktentabelle ausgenutzt. Neben der mehrdimensionalen Einschränkung sind jedoch Gruppierungen für komplexe DWs von großer Bedeutung, um für die jeweilige Konsolidierungsstufe die jeweiligen Kennzahlen zu berechnen. Da die einzelnen Dimensionen in relationalen Datenbanksystemen in separaten Tabellen verwaltet werden (Star Schema), werden sie bei einem Zugriff mit der Faktentabelle verbunden. Eine Möglichkeit, den Verbundsoperator effizient abzuarbeiten, ist der Sort-Merge-Join [Här77], [Gra93]. Somit ist die dritte Grundfunktion im DW die Sortierung. Diese drei Grundfunktionen muss also ein Datenbankverwaltungssystem (DBVS) effizient unterstützen, um den Anforderungen eines DW gerecht zu werden.

1.3 Fragestellungen und Zielsetzung

Aus den im vorherigen Abschnitt beschriebenen Anforderungen ergeben sich für diese Arbeit im Wesentlichen drei Fragestellungen. Ausgangspunkt ist die UB-Technologie, d.h. die Faktentabelle des DW ist als UB-Baum [Bay96], [Bay97a] organisiert.

Um die Verbundoperation mit einer Dimension effizient zu unterstützen, sollte die Faktentabelle sortiert nach dem Dimensionsschlüssel gelesen werden. Die daraus entstehende Fragestellung lautet:

- Wie erzeugt man aus einem UB-Baum einen sortierten Strom von Tupeln unter Berücksichtigung von Restriktionen?

Die Faktentabelle wird durch den UB-Baum organisiert. Somit können die Restriktionen in allen Dimensionen ausgenutzt werden, da es sich hier um eine Art symmetrischen clusternden Index handelt. Für die einzelnen Konsolidierungsstufen müssen die Daten bezüglich der Hierarchieobjekte gruppiert werden, um die geforderten Aggregate zu berechnen. Die zweite Fragestellung dieser Arbeit behandelt die Gruppierung.

- Wie erzeugt man aus einem UB-Baum eine Gruppierung unter Berücksichtigung von Restriktionen?

Die dritte Fragestellung beschäftigt sich mit dem Problem der Indexerzeugung. Da die Daten nicht kontinuierlich in DW geschrieben werden, müssen sie durch ein Ladeverfahren in die Faktentabelle integriert werden. Üblicherweise sind die Daten bezüglich der Zeitdimension vorsortiert. Das Ladeverfahren sollte die Eigenschaft der Vorsortierung berücksichtigen. Dies führt zur folgenden Fragestellung:

- Wie lädt man eine vorsortierte Folge von Tupeln in einen UB-Baum?

1.3.1 Sortieren eines Tupelstroms unter Berücksichtigung von Restriktionen

Betrachten wir zunächst das Sortieren. Der UB-Baum, der im Wesentlichen ein B^*-Baum ist, verwaltet die Daten bezüglich der Z-Ordnung auf dem Sekundärspeicher. Hierbei werden üblicherweise die Werte der Schlüsselattribute der Tabelle verwendet. Ziel ist es nun, die Daten, die das Selektionsprädikat erfüllen, vom Sekundärspeicher zu holen und nach einem Attribut sortiert auszugeben. Hierbei soll das Attribut Element des Schlüssels sein.

```
SELECT K1, K2, K3, …
FROM   Fakten_Tabelle_UB-Baum
WHERE  K1 BETWEEN r1 AND r2,
K2 BETWEEN r3 AND r4,
K3 BETWEEN r5 AND r6,
       …
ORDER BY Ki,…
```

Abbildung 1-12: Bereichsanfrage mit Sortierung

Dieser Sachverhalt ist in Abbildung 1-12 als SQL dargestellt. Hierbei ist die Faktentabelle als UB-Baum organisiert. Die einzelnen Schlüsselattribute werden in der WHERE-Klausel auf Intervalle eingeschränkt und bilden somit eine zusammenhängende mehrdimensionale Bereichsanfrage *Q*. Die Ergebnismenge *E*, eine Projektion auf K_1, K_2, K_3, … K_n der Faktentabelle, wird bezüglich *Ki*, einem Attribut des Schlüssels, sortiert ausgegeben.

Da ein wesentlicher Teil des Problems in der Erstellung einer bestimmten Ordnung besteht, definieren wir zunächst die Begriff *Ordnung* und *totale Ordnung*:

Definition 1-1: Ordnung

Eine Teilmenge $\leq$ von $R \times R$ heißt *Ordnungsrelation* ,

1. identitiv: $(a \leq b) \wedge (b \leq a) \Rightarrow (a = b)$
2. reflexiv: $\forall a \in R: a \leq a$
3. transitiv: $(a \leq b) \wedge (b \leq c) \Rightarrow a \leq c$

Basierend auf Definition 1-1 definieren wir die totale Ordnung wie folgt:

Definition 1-2: Totale Ordnung

Eine Teilmenge $\leq$ von $R \times R$ heißt *total geordnete Menge* wenn

1. „$\leq$“ ist eine Ordnungsrelation
2. Für alle $a, b \in R$ gilt: $(a \leq b) \vee (b \leq a)$

Formal können wir nun das zu behandelnde Sortierproblem, wie folgt beschreiben: Gegeben ist eine Folge von Tupeln:

$$\langle x_{(0)}, x_{(1)}, x_{(2)}, \ldots, x_{(n-1)} \rangle \text{ mit } x_{(0)} \prec x_{(1)} \prec x_{(2)} \prec \ldots \prec x_{(n-1)} \qquad \text{Gl: 1-1}$$

Hierbei bezeichnet $\prec$ die Z-Ordnung, $x_{(m)}$ das *m*-te Element der Folge. Ohne Beschränkung der Allgemeinheit besteht ein Tupel nur aus Schlüsselattributen $\boldsymbol{K} = \{ K_0, K_2, K_3, \ldots, K_{d-1} \}$, wobei x_i den Wert des Attributes K_i des Tupels *x* bezeichnet. Mit $x_{(m),i}$ kann somit auf den Wert des Attributes K_i des *m*-ten Elements der Folge zugegriffen werden.

Der Sortieralgorithmus soll nun eine Permutation π der Folge erzeugen, für die gilt:

$$\langle x_{\pi(0)}, x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n-1)} \rangle \text{ mit } x_{\pi(0),i} \leq x_{\pi(1),i} \leq x_{\pi(2),i} \leq \ldots \leq x_{\pi(n-1),i} \qquad \text{Gl: 1-2}$$

Das Symbol "≤ " bezeichnet die Ordnungsrelation auf der Domäne von Attribut K_i.

In dieser Problembeschreibung sind noch einige Details offengelassen. Diese werden in den nachstehenden Randbedingungen spezifiziert.

Randbedingungen

Da wir uns hier im Bereich von Data-Warehouse und Relationalen Datenbanken befinden und der UB-Baum eine externe Speicherstruktur ist, besteht eine wesentliche Randbedingung darin, dass die betrachtete Datenmenge *R* viel größer als der zur Verfügung stehende Arbeitsspeicher *M* ist und als UB-Baum organisiert ist.

Zielsetzung

Für externe Speicherstrukturen und im Speziellen im Datenbankbereich sind Eingabe- und Ausgabe-Operationen (E/A-Operationen) ein entscheidender Kostenfaktor. Die Zugriffszeit für den Arbeitsspeicher liegt bei ca. 10^{-8} s, für den Sekundärspeicher bei ca. 10^{-2} s. Somit ergibt sich ein Verhältnis von ca. 10^6. Übliche externe Sortierverfahren, wie z.B. Sortieren durch Verschmelzen, müssen die Daten mehrfach lesen. Somit ist ein Ziel die Minimierung der E/A-Operationen.

Sortieren ist üblicherweise eine blockierende Operation, d.h., erst wenn das Ergebnis vollständig sortiert ist, kann das Ergebnis an den Aufrufer weitergereicht werden. Ein weiteres Ziel ist somit, die Antwortzeit dadurch zu verbessern, indem man die Z-Ordnung ausnutzt, um das unsortierte Zwischenergebnis in Partitionen aufzuteilen, die unabhängig von einander sortiert werden können. Die Konkatenation der Teilergebnisse ergibt das sortierte Ergebnis. Eine Partitionierung eröffnet somit auch die Möglichkeit der Parallelisierung.

Das wesentliche Ziel ist somit die Entwicklung eines speziellen Sortierverfahrens auf der externen Datenstruktur UB-Baum unter Berücksichtigung der zu Grunde liegenden Z-Ordnung und Arbeitspeichergröße, um so zusätzliche E/A-Operationen für das Sortieren, wie sie z.B. beim externen Sortieren durch Verschmelzen entstehen, zu vermeiden sowie die Antwortzeit durch Partitionierung zu reduzieren.

1.3.2 Gruppieren eines Tupelstroms unter Berücksichtigung von Restriktionen

Betrachten wir als nächstes die Gruppierung. Auch hier liegen die Daten in einem UB-Baum. Ziel ist es nun, die Daten, die das Selektionsprädikat erfüllen, vom Sekundärspeicher zu holen und nach einem Attribut oder einer Menge von Attributen zu gruppieren. Hierbei soll das Attribut Element des Schlüssels sein. Im Allgemeinen wird dann auf den einzelnen Gruppen eine Mengenfunktion (Aggregatfunktionen, Spaltenfunktionen) angewendet, wie z.B. SUM. Die Berechnung des Ergebnisses der Mengenfunktionen ist jedoch nicht Bestandteil dieser Arbeit.

```
SELECT Ki,..., sum (Km),
FROM   Fakten_Tabelle_UB-Baum
WHERE  K1 BETWEEN r1 AND r2,
K2 BETWEEN r3 AND r4,
K3 BETWEEN r5 AND r6,
       …
GROUP BY Ki,…
```

Abbildung 1-13: Bereichsanfrage mit Gruppierung

Diesen Sachverhalt stellen wir wieder in SQL (siehe. Abbildung 1-13) dar. Die Ergebnismenge wird bezüglich *Ki*, ... , Attributen des Schlüssels, gruppiert.

Formal können wir das zu behandelnde Gruppierungsproblem wie folgt beschreiben: Gegeben ist eine Folge von Tupeln:

$$\langle x_{(0)}, x_{(1)}, x_{(2)}, \ldots, x_{(n-1)} \rangle \text{ mit } x_{(0)} \prec x_{(1)} \prec x_{(2)} \prec \ldots \prec x_{(n-1)} \qquad \text{Gl: 1-3}$$

Ohne Beschränkung der Allgemeinheit bestehen die Tupel nur aus Schlüsselattributen und die Menge der Gruppierungsattribute aus einem Attribut. Der Gruppierungsalgorithmus soll nun die Folge in eine Menge von disjunkten Partitionen $\{B_0, B_1, \ldots, B_{k-1}\}$ bezüglich des Gruppierungsattributs zerlegen

$$\{B_0, B_1, \ldots, B_{k-1}\} \text{ mit } \forall B_i: x, y \in B_i \Rightarrow x_j = y_j \qquad \text{Gl: 1-4}$$
$$\text{und } \bigcup_{i=0}^{k-1} B_i = \{x_{(0)}, x_{(1)}, x_{(2)}, \ldots, x_{(n-1)}\}$$

Das Symbol x_j bezeichnet das Gruppierungsattribut.

Randbedingungen

Es bestehen dieselben Randbedingungen wie in Abschnitt 1.3.1.

Zielsetzung

Die Zielsetzung entspricht im Wesentlichen der von Abschnitt 1.3.1, d.h., die Reduzierung der E/A-Operationen sowie die Beschleunigung der Antwortzeiten, wobei die Möglichkeit der Zerlegung in Partitionen durch die Z-Ordnung ausgenutzt werden soll.

1.3.3 Massenladen

Das effiziente Laden großer Mengen von Daten in einen UB-Baum stellt die letzte Problemstellung dieser Arbeit dar. Übliche Ansätze, die vom B-Baum adaptiert werden, wie z.B. die Daten durch externes Sortieren durch Verschmelzen in die Z-Ordnung zu bringen und in den B-Baum zu laden, berücksichtigen nicht die Vorsortierung der zu ladenden Daten. Da die Daten meistens nach einem Attribut sortiert vorliegen, wie z.B. Zeit, soll diese Sortierung zum Erstellen des UB-Baums ausgenutzt werden. Somit kann das Ladeproblem auf das oben beschriebene Sortierproblem abgebildet werden, mit dem Unterschied, dass jetzt die Z-Ordnung die Zielordnung und nicht die Quellordnung ist.

Formal können wir das Problem also wie folgt beschreiben. Gegeben ist eine Folge von Tupeln:

$$\langle x_{(0)}, x_{(1)}, x_{(2)}, \ldots, x_{(n-1)} \rangle \text{ mit } x_{(0),i} \leq x_{(1),i} \leq x_{(2),i} \leq \ldots \leq x_{(n-1),i} \qquad \text{Gl: 1-5}$$

Ohne Beschränkung der Allgemeinheit bestehen die Tupel nur aus Schlüsselattributen. Der Sortieralgorithmus soll nun eine Permutation der Folge erzeugen, für die gilt:

$$\langle x_{\pi(0)}, x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n-1)} \rangle \text{ mit } x_{\pi(0)} \prec x_{\pi(1)} \prec x_{\pi(2)} \prec \ldots \prec x_{\pi(n-1)} \qquad \text{Gl: 1-6}$$

Randbedingungen

Es bestehen dieselben Randbedingungen wie in Abschnitt 1.3.1.

Zielsetzung

Die Zielsetzung entspricht im Wesentlichen der von Abschnitt 1.3.1, d.h., die Reduzierung der E/A-Operationen, durch Ausnutzung der Quellordnung.

1.4 Beiträge der vorliegenden Arbeit

Eine sehr wichtige Fragestellung im DB-Bereich ist die Auswahl des besten Zugriffspfades für eine Anfrage an das System. Besonders in der OLAP-Umgebung [CodCS93], wo riesige Mengen von Daten verwaltet werden, sind spezielle Datenstrukturen vorgeschlagen worden [Inf97, NeiQ97] [RouKR97]. Im Allgemeinen benutzen DWs redundante Datenstrukturen, wie Indexe und materialisierte Sichten [ChaD97]. Ein Schwerpunkt der Forschung stellten somit Präaggregationstechniken dar. Eine übliche Methode ist die Erstellung von Sekundärindizes auf Tabellen, die die Anfrageverarbeitung beschleunigen [GupHR+97]. Präaggregationstechniken ergeben die beste Antwortzeit jedoch auf Kosten von Ladezeiten und Speicherplatzbedarf. Nach [CodCS93] sind Ad-hoc-Anfragen eine Anforderung an DW-Systeme, so dass eine Präagregation aller Anfrageergebnisse unmöglich ist. Somit muss ein Zwischenweg gefunden werden, zu dem die UB-Technologie [Bay96] einen entscheidenden Beitrag liefert.

Um Aggregate oder Verbundoperatoren effizient zu berechnen, ist die Sortierung ein entscheidender Faktor [ZirMB00a], [Gra93]. In [Mar99] wird gezeigt, dass mehrdimensionale Bereichsanfragen im DW-Bereich durch mehrdimensionale Datenstrukturen (UB-Baum, R-Baum) erheblich besser unterstützt werden als durch 1-dimensionale Datenstrukturen. Dennoch besteht noch immer das Sortierproblem. In dieser Arbeit wird der Tetris-Algorithmus vorgestellt [MarZB99], eine Verarbeitungstechnik, in der das Sortieren und mehrdimensionale Einschränkungen kombiniert werden. Diese Technik kann sehr gewinnbringend für die Implementierung des Verbundsoperators sowie der Gruppierung eingesetzt werden.

Der TempTris-Algorithmus [ZirMB01] löst das Problem der effizienten Erzeugung von UB-Bäumen beim Massenladen. Diese Verarbeitungstechnik nutzt die Vorsortierung der Daten aus, so dass man ihn zur Klasse der adaptiven Sortier-Algorithmen zählt. Er reduziert die E/A-Komplexität gegenüber dem externen Sortieren durch Verschmelzen auf die Hälfte.

Neben künstlich erzeugten Daten wird das DW der *Gesellschaft für Konsumforschung* (*GfK*) zu Analyse-Zwecken herangezogen, um die Tauglichkeit der in dieser Arbeit vorgestellten Algorithmen unter realen Bedingungen zu prüfen. Anhand dieses DWs

werden die typischen Fragestellungen in der Anfrageverarbeitung sowie die neuen Techniken, die in dieser Arbeit vorgestellt werden, erläutert. Deshalb basieren die meisten Beispiele auf dem DB-Schema der GfK. Das GfK-Schema (siehe Abbildung 1-3) ist ein Schneeflocken-Schema mit drei Dimensionen: Zeit (time), Segment (*segment*) und Produkt (*product*). Das GfK-DW besteht aus einer Faktentabelle, wobei die Fakten nur als Datensätze der feinsten Granularität bezüglich jeder Dimension abgelegt sind. Das Schema wird mit Hilfe von Hierarchiegraphen erklärt. Die Richtung der Pfeile beschreibt die der Granularität der Daten und somit die Genauigkeit der Daten. Da es sich hier um einen gerichteten Graphen ohne Zyklen handelt, kann die Verbindung der einzelnen Hierarchieebenen über 1-zu-N Relationen abgebildet werden. Das GfK Schema enthält keine N-zu-M Beziehungen. Eine genaue Beschreibung des GfK-Schemas ist in [Mer99] zu finden.

1.5 Mistral-Projekt

Diese Arbeit ist innerhalb des Mistral-Projektes entstanden. Mistral steht für „**M**ultidimensional **I**ndexes for **St**orage and for the **R**elational **Al**gebra". Ziel dieses Projekts ist die Optimierung von relationalen Operatoren unter Zuhilfenahme von mehrdimensionalen Zugriffsstrukturen. Hierbei wird ein besonders Augenmerk auf den UB-Baum [Bay97a] gelegt, der von Herrn Prof. R. Bayer erfunden wurde. Hierbei werden vor allem die folgenden Punkte betrachtet:

- Grundlagenforschung im Bereich mehrdimensionaler Indexstrukturen
- Implementierung, Benchmarks sowie Verbesserung des UB-Baums
- Erstellung einer Softwarebibliothek mit UB-Baum-Funktionalität
- Integration des UB-Baums in kommerzielle Datenbanksysteme
- Implementierung relationaler Operatoren unter Verwendung des UB-Baums

Allgemein können mit der UB-Baum-Technologie Datenbankoperationen stark beschleunigt werden. In Kooperation mit Projektpartnern wie SAP, Teijin System Technology, NEC, Hitachi, GfK, Transaction Software und Microsoft Research konzentriert sich das Projekt auf die Gebiete DW, Data-Mining sowie auf die Integration des UB-Baums in den Kern des relationalen Datenbanksystems Transbase. Hypercube von der Firma Transaction ist die erste kommerzielle Implementierung des UB-Baums, die verfügbar ist. Die Integration wurde von den Mitgliedern des Mistral-Projekts in Kooperation mit Transaction vorgenommen. [RamMF*00] beschreibt ausführlich den Bereichsanfragealgorithmus, die Integration ins kommerzielle DB-System Transbase sowie Leistungsmessungen auf dem DW der GfK. Der UB-Baum ist von Herrn Prof. Bayer patentiert. Eine interessante Abhandelung zum Thema Urheberecht ist in [Zir02] zu finden.

1.6 Aufbau der Arbeit

Die Arbeit gliedert sich in zwei Teile, Algorithmen und Anwendungen. In Kapitel 2 werden verwandte Arbeiten zu diesem Bereich behandelt. Kapitel 3 stellt die Grundlagen für diese Arbeit vor. Es wird eine Speicherhierarchie eingeführt, die für die theoretische Analyse herangezogen wird. Neben der Klassifizierung von Zugriffsmethoden wird als wesentliches Konzept die Clusterung eingeführt. Der Hauptteil dieses Kapitels widmet sich der Zugriffsstruktur UB-Baum, die wesentlich für die vorgestellten Algorithmen ist. Kapitel 4 führt das sortierte Lesen ein. Dort wird der SRQ-Algorithmus analysiert. Nach der Einführung eines formalen Modells wird ein abstrakter Algorithmus für das sortierte

Lesen vorgestellt. Basierend auf diesem Prinzip wird sodann die Implementierung des SRQ-Algorithmus, der auf dem bekannten Bereichsanfragealgorithmus basiert, vorgestellt. Für diesen Algorithmen wird ein Kostenmodell vorgestellt und durch Messungen auf künstlichen Daten bestätigt. Kapitel 5 führt den Tetris-Algorithmus ein, der auf einer neuen raumfüllenden Kurve basiert, der Tetris-Kurve. Es ist die zweite Implenentierung des sortierten Lesens. Nach Einführung der Tetris-Ordnung wird ein linearisierter Algorithmus für die Berechnung der minimalen Tetris-Adresse einer Region vorgestellt. Es ist der Kernalgorithmus für den Tetris-Algorithmus. Kapitel 6 führt einen Gruppierungsalgorithmus ein, der auf dem Tetris-Algorithmus basiert. Der Algorithmus wird mit dem theoretischen Modell analysiert. Kapitel 7 beschreibt den TempTris-Algorithmus, die inverse Operation des Tetris-Algorithmus. Basierend auf einem formalen Modell wird der Algorithmus beschrieben. Nach einem Korrektheitsbeweis werden die grundlegenden Leistungsmerkmale und die Optimierungsmöglichkeiten betrachtet. Nach der theoretischen Analyse werden die Ergebnisse durch Messungen auf künstlichen Daten bestätigt. Kapitel 8 beschreibt die Prototypimplementierung, die für die Messungen verwendet wurde. Der zweite Teil der Arbeit beschäftigt sich mit den Anwendungen. Kapitel 9 betrachtet das Anwendungsgebiet Data-Warehouse. Nach der theoretischen Beschreibung des DW wird speziell der CUBE-Operator betrachtet. Für den CUBE-Operator werden verschiedene Verarbeitungstechniken präsentiert, die auf dem UB-Baum basieren. Die Arbeit endet mit einer Zusammenfassung und einem Ausblick in Kapitel 10.

A little more than kin, and less than kind.

***William Shakespeare** (1564–1616)*

2 Verwandte Arbeiten

Im Folgenden fassen wir die relevanten Publikationen aus den Bereichen mehrdimensionale Datenstrukturen, Anfragenverarbeitung und Massenladen zusammen, die grundlegend für diese Arbeit sind.

2.1 Mehrdimensionale Datenstrukturen

Suchoperationen in Datenbanksystemen benötigen eine spezielle Unterstützung auf der physischen Ebene. Dies trifft sowohl auf konventionelle Datenbanksysteme sowie GIS-Datenbanksysteme zu, deren Anfragenprofil Punkt- und Bereichsanfragen enthalten. Die intensive Forschung auf dem Gebiet der GIS-Datenbanksystem führte in den letzten zehn Jahren zu einer Vielzahl von mehrdimensionalen Zugriffsmethoden, die solche Anfragen unterstützen. Typische Anwendungsgebiete sind die Auswertung der Daten von ERP-Systemen [SAP01a] oder der medizinische Bereich [Zir98]. In [GaeG98], [Sam90] ist ein umfassender Überblick zu diesem Gebiet zu finden.

Die d-dimensionalen Datenstrukturen werden allgemein nach der Art der Daten klassifiziert [GaeG98]. Die Daten werden als räumliche Daten bezeichnet. Räumliche Daten bestehen aus komplexen Strukturen, wobei ein Objekt aus einem einzigen Punkt oder aus tausenden von Polygonen bestehen kann, die willkürlich über das Universum verteilt sind [GaeG98]. Ein Vertreter der Zugriffspfade für räumlich ausgedehnte Objekte ist der R-Baum [Gut84]. Der R-Baum basiert auf dem Konzept der *überlappenden Regionen* (object bounding) [GaeG98], so dass die Regionen des R-Baumes das Universum nicht in disjunkte Teilräume zerlegen. Hierdurch soll die Tiefe des Baumes reduziert werden. Das überlappende Regionskonzept verursacht jedoch die größten Kosten dieser Zugriffsmethode, so dass neue Varianten entwickelt wurden, wie z.B. der R^*-Baum [BecKS+90], der eine Variation des Split-Algorithmus ist. Diese Varianten reduzieren die Überlappungen und damit die Anzahl der Pfade, die bei der Suchoperation betrachtet werden müssen. Das grundsätzliche Problem bleibt jedoch erhalten. Der R^+-Baum [SelRF87] zerlegt die räumlich ausgedehnten Objekte in disjunkte Teilräume, um die Überdeckungsproblematik zu lösen. Dieser Typ von Verfahren wird in der Literatur als Clipping (Ausschneiden) bezeichnet. Hierdurch können jedoch mehre Datenobjekte entstehen, die alle auf dasselbe ausgedehnte räumliche Objekt verweisen. Die Repräsentation eines räumlich ausgedehnten Objekts durch eine Menge von Datenobjekten kann die physische Clusterung auf dem Sekundärspeicher zerstören, so dass hierdurch ein erheblicher Leistungsverlust entsteht. Daneben sind die Algorithmen für bestimmte Anfragetypen, wie z.B. die Enthaltenseinsanfrage, erheblich komplexer. Trotz flexiblerer Indexierungsmöglichkeiten sind bisher im allgemeinen Fall keine Leistungsvorteile zugunsten des R^+-Baums publiziert worden [Gre89], [HärR99]. Da die Wartung zur Wiederherstellung der Balancierung des R^+-Baums erhebliche komplexe Algorithmen

benötigt [SelRF87], kann er nicht als verbesserte Variante des R-Baums bezeichnet werden.

Handelt es sich bei den Daten um Punkte im d-dimensionalen Raum, können spezielle d-dimensionale Punktzugriffsmethoden herangezogen werden. In diese Klasse fallen z.B. k-d-B-Bäume [Rob81], [ChaF81], Grid-Files [NieHS84] sowie die B-Bäume [BayM72],[Knu98v3],[Com79], deren Schlüssel aus einer Konkatenation von Attributen entsteht. Die Sortierung auf den einzelnen Attributen kann hier separat festgelegt werden [BlaCE77]. Dieser Zugriffspfad ist jedoch nicht allgemein nutzbar, da es sich hier um eine unsymmetrische Zugriffsmethode handelt, die das erste Attribut, das zum Schlüssel beiträgt, favorisiert.

Dieses Problem wird vom UB-Baum [Bay96],[Mar99], eine Variante des zkd-B-Tree [OreM84], überwunden, indem die Schlüsselattribute bit-interleaved (verzahnt) werden. Der Schlüssel wird dadurch erzeugt, dass zunächst die höchstwertigen Bits der Schlüsselattribute nach einer festen Reihenfolge konkateniert werden, dann die zweithöchstwertigen, und so weiter, bis alle Bits aufgebraucht sind. Das grundlegende Konzept besteht darin, dass auf einem d-dimensionalen Raum eine totale Ordnung erzeugt wird, wobei die räumliche Nähe bis zu einem gewissen Grad erhalten bleibt, d.h., dass zwei Objekte, die im d-dimensionalen Raum benachbart sind, auch in der totalen Ordnung nicht weit voneinander entfernt sind. Hierzu verwendet der UB-Baum die Z-Kurve [Sag94], die einen d-dimensionalen Raum auf einen 1-dimensionalen Raum abbildet. Die Verwaltung des 1-dimensionalen Raums wird dann vom 1-dimensionalen Zugriffspfad, dem B^*-Baum, übernommen. Im Gegensatz zum zkd-B-Baum werden diese Schlüssel jedoch nur für die Regionen berechnet, die die Datenseiten des B^*-Baums [BayU77, Com79] darstellen und demzufolge im Index-Teil gespeichert sind. Einem ausgiebigen theoretischen und praktischen Vergleich des UB-Baums mit dem R*-Bäumen ist in [Ram02] zu finden. Dort schneidet der UB-Baum im Bezug auf die Anfrageperformanz sowie bei den wichtigen Aspekten der Wartungsperformanz, Indexgröße, u.a. besser als der R*-Baum ab. In [Fenk, 2002 #211]wird eine Methode vorgestellt, mit der Intervalle effizient durch den UB-Baum verwaltet werden. Hierzu wird der Dualraum eingesetzt.

2.2 Anfrageverarbeitung

Die Anfrageoptimierung ist seit einigen Jahrzehnten von großem Interesse in der Datenbankforschung. Man unterteilt sie im Wesentlichen in folgende Phasen: *lexikalische und syntaktische Analyse, Semantische Analyse, Zugriffs- und Integritätskontrolle, Anfrageoptimierung, Code-Generierung/direkte Ausführung oder interpretative Ausführung*. Ein detaillierte Beschreibung der Anfragenverarbeitung ist in [Mit95] und [FreMV94] zu finden. Nach der lexikalischen, syntaktischen und semantischen Analyse sowie der Integritätskontrolle liegt die deklarative Anfrage als äquivalenter Ausdruck der relationalen Algebra vor, der als Eingabe für die Anfrageoptimierung dient. In der Anfrageoptimierung wird für einen gegebenen algebraischen Ausdruck ein möglichst effizienter *Auswertungsplan* (*query evaluation plan,* QEP) erzeugt. Der Auswertungsplan kann dann entweder kompiliert oder direkt interaktiv interpretiert werden. Ein Algorithmus zur Implementierung eines Operators wird als Operator einer *physischen Algebra* betrachtet. Genau wie relationale Operatoren *„verbraucht“* eine Implementierung eine oder mehrere Eingabequellen, um eine oder mehrere Ausgaben zu erzeugen. Erste Beiträge zu diesem Thema finden sich bereits in der Beschreibung des INGRES Optimierers 1976 [WonY76], in der die Zerlegung von Anfragegraphen beschrieben wird. [SelAC+79] veröffentlichen eine wegweisende Arbeit über den System R Optimierer, in dem physische

Eigenschaften von Relationen zur Auswahl einer günstigen Verbundoperation ausgenutzt werden. Die Optimierung basiert auf dynamischer Programmierung, wobei jedoch alle Alternativen bewertet werden.

Allgemein unterscheidet man die *logische Optimierung* und die *physische Optimierung*. Bei der logischen Optimierung wird durch Anwendung von *Transformationsregeln* unter Berücksichtigung von Heuristiken ein äquivalenter algebraischer Ausdruck erzeugt. Die Grundidee besteht darin, die Regeln so anzuwenden, dass die Ergebnisse der einzelnen relationalen Operatoren möglichst klein sind. Das ist besonders wichtig, wenn das Ergebnis aufgrund eines Hauptspeichermangels temporär auf dem Sekundärspeicher ausgelagert werden muss [Jar84]. Die physische Optimierung betrachtet die physischen Algebraoperatoren. Für die logischen Operatoren existieren meistens mehrere physische Operatoren, um eine effiziente Implementierung bereitzustellen [Gra93]. Hierbei wird der physische Aufbau der Datenbank, also die Existenz von Zugriffsstrukturen, Clusterung und Sortierung von Relationen einbezogen und durch ein Kostenmodell evaluiert [HarR96]. Man bestimmt den besten QEP für die Anfrage indem man die Kosten für die einzelnen Varianten durch ein mathematisches Modell bestimmt und dann den günstigsten QEP wählt. Als Metrik wird üblicherweise die Anzahl der Tupel, die ein Operator verarbeitet, verwendet. Deren Bestimmung (Kardinalität) untersucht die Forschung bereits seit über 20 Jahren [SelAC+79], [Gel93], [SwaS94]. Im Rahmen der Anfrageverarbeitung werden also höhere Operatoren wie Verbund und Selektion auf Zugriffsplänen abgebildet, die aus Planoperatoren bestehen. Da die Planoperatoren quasi als Bausteine zur Bestimmung und Generierung von optimalen Zugriffsplänen dienen, ist eine effiziente Implementierung besonders wichtig [Gra93]. Der Verbundoperator wird üblicherweise als Nested Loop, Sort-Merge oder Hash-Algorithmus implementiert. Ein ausführlicher Überblick über die verschiedenen Verarbeitungstechniken ist in [MisE92], [Gra93] zu finden. Mehrdimensionale Indexstrukturen werden eingesetzt, um räumliche Restriktionen, z.B. Bereichsanfragen, Durchschnitt, etc. sowie räumliche Joins effizient zu berechnen [Rot91],[Gün93]. Die meisten Arbeiten im Bereich RDBVS betrachten die Einsatzmöglichkeiten mehrdimensionaler Zugriffstrukturen zur Beschleunigung von Bereichsanfragen [NieHS84], [LomS90], [TroH81], [Bay96], [BayM98].

In [HarNK+90] und [Bay97b] werden Verarbeitungstechniken für den Verbundoperator vorgestellt, die die physikalische Organisation von Punktdaten durch mehrdimensionale Zugriffsstrukturen ausnutzen, um die Daten sortiert nach dem Verbundprädikat zu verarbeiten. Eine Verallgemeinerung dieses Verfahrens auf Bereichsanfragen wurde in [MarZB99] vorgestellt, das Bestandteil dieser Arbeit ist.

Im Bereich von OLAP-Systemen [CodCS93], [ChaD97], in denen primär historische Daten verwaltet werden, wurden, basierend auf dem Star-Schema [Kim96], in der physikalischen Optimierung z.B. Bitmap-Index, Verbund-Indexe (Join Index, Foreign Colum Join Index, Mutiple Join Index) [NeiG95], [Inf97] vorgeschlagen. Hierbei handelt es sich jedoch um 1-dimensionale Zugriffsstrukturen, die nur die Selektivität in einer Dimension ausnutzen. Um die Restriktionen aller Dimensionen auszunutzen, werden auf allen Dimensionen Indexe angelegt und durch einen Indexschnitt das Ergebnis der Faktentabelle bestimmt. In [Sar97] und [Mar99] wird gezeigt, dass mehrdimensionale Zugriffsmethoden wie der R-Baum [Gut84] und der UB-Baum [Bay96] für Bereichsanfragen besser geeignet sind als 1-dimensionale Zugriffsmethoden.

Der wesentliche relationale Operator im OLTP- System ist die Gruppierung in Verbindung mit einer Aggregationsfunktion. Klassische Arbeiten wie [SelAC+79] im Bereich der

Anfrageverarbeitung unterstützen keine Aggregationsoperatoren. Diese Erweiterungen werden in [ChaS94], [YanL94],[YanL95] vorgestellt. In [GraCB+97] wird der CUBE-Operator vorgestellt, der die Aggregationsfunktionen und den Gruppierungs-Operator auf den n-dimensionalen Fall verallgemeinert. Der wesentliche Kern dieser Arbeit ist die Einführung des Schlüsselworts „All“, das die Ausprägungen eines Attributs auf einen Wert reduziert und somit die größte Verdichtung darstellt. Bei der Implementierung werden Präaggregationstechniken von Teilwürfeln sowie Aggregationsgitter verwendet, um die Antwortzeit zu verbessern [SAP01a]. In [HarRU96] wird das Problem, welcher Teilwürfel vorberechnet werden soll, betrachtet. Dies führt zu einem einfachen Greedy-Algorithmus. Die Kombination Index und Präagregation wird in [GupHR+97] behandelt. Einen ausgezeichneten Überblick über Aggregationstechniken ist in [Leh98] zu finden. In [ZirMB00a] und [ZirMB01] werden Verarbeitungstechniken vorgestellt, basierend auf dem UB-Baum, die ein gegebenes Aggregationsgitter durch Vermeidung von externem Sortieren effizient berechnen. Diese Arbeiten stellen einen weiteren Kern dieser Arbeit da. In [Pieringer, 2003 #212]wird die Implementierung vom MHC (multidimensionales hierarchisches Clustering) in das relationale Datenbanksystem Transbase Hypercube beschrieben. MHC wird notwendig, wenn Data Warehouse-Daten in einem clusternden multidimensionalen Index gespeichert werden sollen. Multidimensionales hierarchisches Clustering ist eine Technologie, um hierarchische Prädikate auf Dimensionstabellen auf Intervalle abzubilden, die über den multidimensionalen Index (UB-Tree in Transbase) optimal verarbeitet werden können. Hierarchie-Pfade werden durch Surrogate (numerische Codierung) abgebildet und in der Faktentabelle gespeichert. Interessante Aspekte zeigen sich beim Query-Processing von sog. Star Join Queries,[Pieringer, 2003 #209],[Karayannidis, 2002 #210] wo durch die Hierarchie-Codierung in der Faktentabelle bereits sehr früh hierarchische Eigenschaften ausgenutzt werden können. Dies beschleunigt die Gruppierung und den Residual-Join erheblich.

2.3 Massenladen

Um riesige Datenbestände in ein Datenbanksystem zu laden, werden Massenladealgorithmen eingesetzt. Hierbei wird die Mehrbenutzersynchronisation für die Dateiverwaltung ausgeschaltet, die Daten in die Datenbasis geschoben und die entsprechenden Indexe erzeugt.

Die übliche Methode, einen optimalen B-Baum [BayM72] bezüglich der Seitenauslastung durch Massenladen aufzubauen, beruht auf dem Sortieren der Quelldaten nach dem Schlüsselattribut. Die sortierten Quelldaten werden dann auf Seiten mit dem gewünschten Füllungsgrad geschrieben. Um große Datenmengen zu sortieren, wird das externe Sortieren durch Verschmelzen eingesetzt. Eine detaillierte Analyse ist in [Knu98v3] zu finden. Ein entscheidender Faktor für die E/A-Komplexität beim externen Sortieren ist die Größe des Arbeitspeichers und damit die Anzahl der Initialläufe. Verwendet man „replacement selection“ [Gra93], der auf einem Heapsort [Knu98v3] beruht, können im Durchschnitt Läufe erzeugt werden, die zweimal so groß wie der Arbeitsspeicher sind. Vergleicht man dies mit dem Quick-Sort Algorithmus [Knu98v3], erhält man nur die Hälfte der Initialläufe. Der Vorteil der Reduzierung der Initialläufe wird jedoch durch ein schlechteres E/A-Verhalten und kompliziertes Speichermanagement zunichte gemacht. In [LarG98] wird jedoch gezeigt, dass es möglich ist Läufe, die 1,8 Mal größer als der Arbeitsspeicher sind, zu erzeugen.

Auf dem Gebiet mehrdimensionaler Datenstrukturen wurden bereits einige Arbeiten, wie z.B. für den R-Baum [KamF93], Grid-Files [LeuN97], Quad Trees [HjaSS97] und UB-

Bäume [FenKM00], vorgestellt. Bei [FenKM00] handelt es sich um eine Adaption des Masselade-Algorithmus für B^+-Bäume. Diese Algorithmen haben aufgrund des externen Sortierens eine E/A-Komplexität von $O(P \log P)$ für P Seiten.

Eine sehr ausführliche Untersuchung der grundlegenden Sortier-Algorithmen ist in [Knu98v3] zu finden. Ein interessanter Teilbereich ist die Klasse der adaptiven Sortier-Algorithmen [EstW92]. Hierunter versteht man Sortier-Algorithmen, die weniger Arbeit brauchen, wenn die Daten bereits einen gewissen Grad an Sortiertheit besitzen.

Dieser Aspekt wird im TempTris-Algorithmus [ZirMB01] aufgenommen, um die E/A-Komplexität zu linearisieren.

Reason, Observation, and Experience—the Holy Trinity of Science.

***Robert G. Ingersoll** (1833–1899), U.S. lawyer.*

3 Grundlagen

Dieses Kapitel führt den Leser in die grundlegenden Definitionen und Notationen dieser Arbeit ein und bildet somit die Grundlage für die weiteren Kapitel.

Im ersten Teil wird das relationale Modell kurz eingeführt. Danach wird ein kleiner Überblick über mehrdimensionale Zugriffsmethoden gegeben, wobei der Schwerpunkt auf den UB-Baum [Bay96] gelegt wird. Zunächst werden 1-dimensionale Zugriffsmethoden betrachtet, die in heutigen Datenbanksystemen wie ORACLE, DB2 oder TransBase State of the Art sind. Hierbei wird speziell auf den B-Baum [BayM72] und seine Varianten [BayU77], [WedZ86], [Knu98v3], die die Grundlage für den UB-Baum bilden, eingegangen. Für einen umfassenden Überblick wird auf [Com79] verwiesen. Daneben wird noch auf die entstehenden Kosten bei einem externen Speicherzugriff eingegangen.

3.1 Relationales Modell

Das relationale Modell wurde von [Cod70] in den siebziger Jahren vorgestellt. Die wesentliche Innovation dieses Modells lag in der *mengenorientierten Verarbeitung* der Daten. Dies war eine bedeutende Abstraktion vom damals vorherrschenden *satzorientierten* Datenmodell, dem *Netzwerkmodell* und dem *hierarchischen Modell*. Im satzorientierten Datenmodell werden die Informationen auf Datensätze, die über Referenzen verknüpft sind, abgebildet. Bei der Verarbeitung erfolgt die Navigation durch Folgen der Referenzen. Die einzelnen Datensätze werden eindeutig durch ihren *Tupelidentifikator* (TID) identifiziert.

Das relationale Datenmodell besteht im Wesentlichen aus einer Struktur, der Relation, auch Tabelle genannt. Die *Wertebereiche* (*Domänen*) D_1,D_2, ... ,D_n bestehen ausschließlich aus atomaren Werten. Die Relation oder Tabelle ist somit eine Teilmenge des kartesischen Produkts (Kreuzprodukt) der *n* Domänen:

$$R \subseteq D_1 \times D_2 \times \ldots \times D_n \qquad \text{Gl: 3-1}$$

Das Schema einer Relation ist durch die *n* Domänen definiert. Die *Instanz* (*Ausprägung*) der Relation ist die aktuelle Teilmenge des Kreuzprodukts. Die Elemente der Relation, die Daten, werden als *Tupel* bezeichnet, dessen *Stelligkeit* (engl. *arity*) sich aus dem Relationenschema ergibt.

Die Ausprägung einer Relation kann als Tabelle dargestellt werden, wobei zur Spezifizierung der einzelnen Komponenten des Tupels Namen benutzt werden. Diese Namen, die die Spalten einer Tabelle bezeichnen, werden *Attribute* genannt und müssen

innerhalb einer Tabelle eindeutig sein. In dieser Arbeit wird das Relationenschema wie folgt spezifiziert:

$$R =: \{[A_1 :D_1 ,A_2 :D_2,...,A_n:D_n]\} \qquad \text{Gl: 3-2}$$

Die eckigen Klammern [,] spezifizieren die Struktur eines Tupels, also den Attributnamen und dessen Domäne. Die geschweiften Klammern drücken den Mengenaspekt aus.

Beispiel 3-1: Tabelle
Abbildung 3-1 zeigt die 4-stellige Relation „Kunde“ mit 3 Tupeln und dem Relationenschema

Kunde:{[Custkey : Integer, Name : Char, Address: Char , Phone: Char]}

Custkey	Name	Address	Phone
1	Zirkel	München	08950029680
2	Müller	Fürstenfeldbruck	0814134232
3	Radtke	München	08950029681

Abbildung 3-1: Tabelle Kunde

Mit **sch**(*R*) bezeichnen wir die Menge der Attribute $\{A_1,A_2,... ,A_n\}$, mit **sch**(a_i) das Attribut A_i mit $a \in R$. Die aktuelle Ausprägung der Relation bezeichnen wir mit *R*. Mit **dom**(*A*) bezeichnen wir die Domäne des Attributs *A*. Ein Schlüssel oder logischer Schlüssel ***K*** ist die minimale Menge von Attributen, deren Wert ein Tupel innerhalb der Relation eindeutig identifizieren. Gib es mehrere Schlüssel (-Kandidaten), wird meist ein sogenannter *Primärschlüssel* bestimmt.

Die Tupel, die in den Tabellen gespeichert sind, werden durch entsprechende Operatoren ausschließlich mengenorientiert verknüpft und verarbeitet. Das Ergebnis ist wieder eine Tabelle. Die einzelnen Operationen sind im Rahmen der relationalen Algebra definiert und arbeiten entsprechend der Grundeigenschaft einer Relation mengenorientiert. Die relationale Algebra besteht aus dem *kartesischen Produkt* „$\times$“, *Projektion* „π“, *Selektion* „σ“, *Verbund* „$\triangleright\triangleleft$“, *Vereinigung* „$\cap$“ und *Mengendifferenz* „$-$“. Wir bezeichnen mit φ_X ein beliebiges Prädikat über die Attributmenge *X* mit $X \in$ **sch**(*R*). Für eine genaue Definition der relationalen Operatoren verweisen wir auf [EmE96], [Ull88], [ElmR94] und [Date95] .

3.2 Zugriffspfade

Datenbankverwaltungssysteme (DBVS) wurden entwickelt, um riesige Mengen von Daten effizient zu verarbeiten. So sind Systeme mit einem Datenvolumen von über 1 Terabyte keine Seltenheit mehr, wie z.B. der Microsoft TerraServer [BarGN*98]. Sie stellen eine abstrakte Sicht auf die vorhandenen Daten dar und geben dem Endbenutzer die Möglichkeit, über eine einfache deklarative Anfragesprache, wie z.B. die „Structured Query Language“ (SQL), auf beliebige Daten zuzugreifen. Grundsätzlich sollten beliebig komplizierte Anfragen effizient abgearbeitet werden.

Die heutige Datenbank-Technologie basiert im Wesentlichen auf dem Größenunterschied zwischen Datenmenge und dem zur Verfügung stehenden Hauptspeicher, so dass die Daten meistens im Sekundärspeicher (Festplatte) oder sogar im Tertiärspeicher (CD, DVD, Bandlaufwerk) liegen. Obwohl die Preise pro KB für die jeweiligen Speichermedien stark

gefallen sind, ist das Verhältnis zwischen diesen fast konstant geblieben [GraG97],[YuM98]. Allgemein kann man sagen, dass bei Datenbanken, die nicht vollständig im Arbeitsspeicher gehalten werden können, die E/A-Kosten einen wesentlichen Kostenfaktor bilden. Für die Leistungsfähigkeit eines DBVS ist es mitunter entscheidend, Datensätze über inhaltliche Kriterien (Schlüssel) möglichst effizient aufzufinden. Hierzu werden so genannte Zugriffspfade oder Zugriffsmethoden bereitgestellt, die einen direkten Zugriff auf ein Datum oder eine Menge von Daten gewährleisten. Wenn kein geeigneter Zugriffspfad existiert, wird die ganze Tabelle gelesen. Dieser Vorgang wird als vollständiges sequenzielles Lesen (Scan) bezeichnet.

Definition 3-1: Vollständiges sequenzielles Lesen

Vollständiges sequenzielles Lesen ist eine Zugriffsmethode, die eine Tabelle vollständig liest, ohne eine Hilfsstruktur zu benutzen.

Zu beachten ist, dass dies nicht nur für 1-dimensionale Zugriffsmethoden gilt. In [BayM98] wird jedoch gezeigt, dass vollständiges sequenzielles Lesen im Einmannbetrieb (single user) für Bereichsanfragen mit einer Selektivität größer als 10 Prozent herkömmlichen Indexstrukturen überlegen ist. Will man z.B. für die einzelnen Quartale des laufenden Jahres bezogen auf die Geschäftsstellen für ein bestimmtes Produkt die Umsätze im GfK-Schema berechnen, hat man ein 3-dimensionales Problem. Wenn die Selektivität mehr als 10 Prozent der gesamten Faktentabelle der GfK ist, stellt das sequenzielle Lesen der gesamten Relation die beste Variante dar, d.h., bei einer Größe der Faktentabelle von 4 GB sollte bei einer Größe der Ergebnisse von mindestens 400 MB das lineare Lesen herangezogen werden. Um die E/A-Kosten, die einen wesentlichen Teil für externe Zugriffsmethoden darstellen, genau zu analysieren, beschreiben wir zunächst die Speicherhierarchie für heutige Datenbank-Verwaltungs-Systeme.

3.2.1 Speicherhierarchie

Eine typische Speicherhierarchie (Abbildung 3-2: Speicherhierarchie) für ein kommerzielles Datenbanksystem besteht aus fünf Schichten. An der Spitze der Hierarchie steht der Cache. Hierbei handelt es sich um einen relativ kleinen, sehr schnellen und teuren, wahlfreien Zugriffs-Speicher, der die Geschwindigkeitsanpassung zwischen den Registern und dem Hauptspeicher herstellt. Leistungsfähige Prozessoren wie der Pentium 4 [Int01] sind heutzutage mit einer Cachehierarchie ausgestattet, wobei bis zu drei Hierarchieebenen L1-L3 mit ansteigender Cachegröße und fallender Zugriffsgeschwindigkeit eingesetzt werden. Dabei ist der L1-Cache mit seinem L1-Controler typischerweise im Chip integriert und erreicht somit Zugriffszeiten, die in der Nähe der Prozessorzyklen liegen.

Die durchschnittliche Zugriffszeit im Cache liegt bei ca. 1 ns [Int01]. Auf der nächsten Stufe befindet sich der allgemeine Haupt- oder Arbeitsspeicher. Die Zugriffszeiten liegen heutzutage bei ca. 5 bis 10 ns und die Größen bei ca. 64 MB bis einigen GB. Bei diesem Speicher handelt es sich um ein flüchtiges Medium, d.h., bei Stromausfall gehen die vorhandenen Daten verloren. Die nun folgenden Speichermedien sind dagegen nicht flüchtig und speichern die Daten auch ohne Stromzufuhr persistent.

Der Sekundärspeicher stellt eine besondere Rolle im Datenbankbereich dar. Es ist die dritte Stufe der Speicherhierarchie mit einer Zugriffszeit von ca. 5 bis 7 ms. Da die Algorithmen, die in dieser Arbeit vorgestellt werden, für Sekundärspeicher entwickelt worden sind,

werden die einzelnen Komponenten einer Festplatte im folgenden Abschnitt genauer behandelt.

In den letzten Jahren ist die Menge an zu verwaltenden Daten so rapide gestiegen, dass die Daten im Allgemeinen nicht mehr im Sekundärspeicher gehalten werden können. Ein Bespiel stellt der Microsoft Terra Server [BarGN*98] dar. Der Microsoft Terra Server speichert Land- und Satelliten-Bilder der Erde, die über das Internet zugänglich sind. Die Nutzer dieser Anwendung kommen zum größten Teil aus dem Gebiet der geographischen Informations-Systeme (Geographical Information System (GIS)) von regionalen, nationalen und internationalen Behörden und Forschungszentren. Derzeit werden ca. 5 TB an Daten gehalten. Riesige Datenmengen entstehen auch im Bereich der Telekommunikation, des Data-Warehousing und Data-Minings. In diesen Fällen wird ein weiteres Speichermedium benötigt, das die vierte Stufe der Speicherhierarchie darstellt: Der Online-Archivspeicher. Hierunter fallen CD-ROMs (**C**ompact **D**isc-**R**ead **O**nly **M**emory) und Band-Roboter. Die Zugriffszeit beträgt ca. 100 ms bis 1 s. Auf der untersten Stufe steht der Archivspeicher, der üblicherweise mit Bändern realisiert wird, deren Zugriffszeit zwischen 1s bis 1 min beträgt. Allgemein kann man sagen, dass ein sehr schneller Speicher eine geringe Kapazität, ein langsamer eine große Kapazität besitzt.

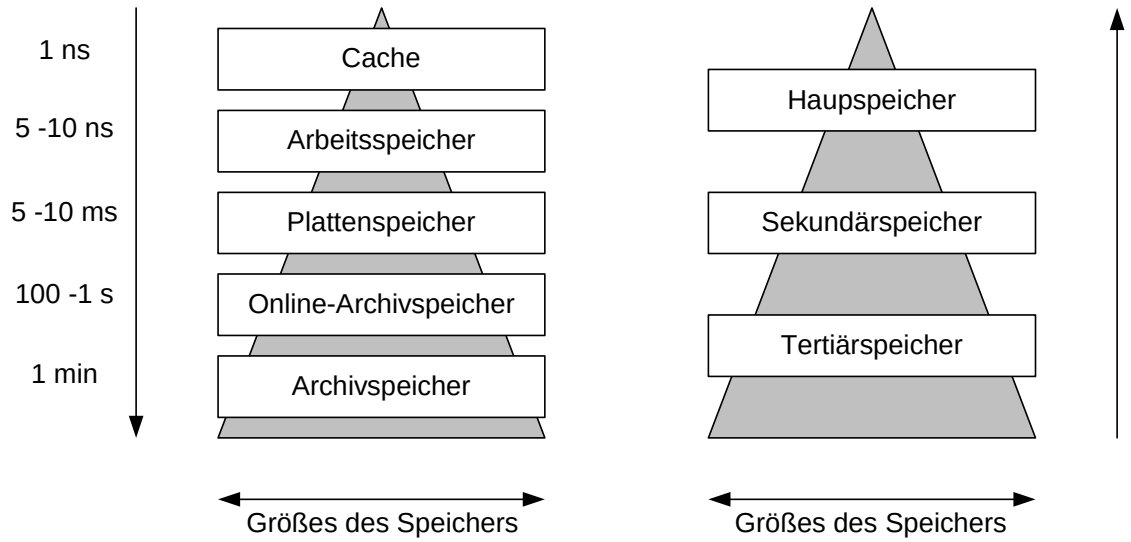


Abbildung 3-2: Speicherhierarchie

In dieser Arbeit wird eine vereinfachte Speicherhierarchie, die aus drei Stufen besteht, benutzt. Dazu kombiniert man den Cache und den Arbeitsspeicher zum Hauptspeicher, dessen Größe mit *M* bezeichnet wird. Die Einheit wird in Seiten angegeben. Den Online-Archivspeicher und den Archivspeicher fasst man als Tertiärspeicher mit der Größenbezeichnung *TS* zusammen. Die Größe für den Sekundärspeicher wird mit *P* bezeichnet.

Ein perfekter Speicher/Cache müsste die vom Rechnerkern benötigten Daten gerade im Speicher/Cache haben. Dies ist nur dann möglich, wenn der Speicher alle Daten vorhalten kann oder es genaue Vorhersagen gibt, welche Daten benötigt werden. In beiden Fällen wäre die Zugriffszeit mit der des Caches identisch. Da man jedoch das Zugriffsverhalten eines Algorithmus nicht zu 100 Prozent vorhersagen kann, werden Heuristiken eingesetzt. Eine sehr beliebte und in vielen Anwendungen auch sinnvolle Heuristik ist das Prinzip der

Lokalität. Dieses Prinzip geht von folgender Annahme aus: „*Daten werden mit ihren benachbarten Daten oft zusammen verarbeitet*“.

Der Grad der *Lokalität* wird allgemein mit der Trefferrate (hit ratio) [GraR97] angegeben:

$$Trefferrate = \frac{\begin{array}{c}\text{Referenz auf den Cache, die über den}\\ \text{Cachezugriff aufgelöst werden kann}\end{array}}{\text{Alle Referenzen auf den Cache}} \qquad \text{Gl: 3-3}$$

Man bezeichnet eine Referenz auflösbar genau dann, wenn sie auf einen Speicherbereich des Caches zeigt. Falls eine Referenz auf den Cache nicht aufgelöst werden kann, muss auf die nächst tiefere Stufe der Speicherhierarchie zugegriffen werden.

Da der Kern dieser Arbeit die sortierte Verarbeitung einer Bereichsanfrage auf externen Daten behandelt, definieren wir die *Trefferrate* wie folgt:

$$Trefferrate = \frac{\text{Anzahl der Tupel, die zum Ergebnis gehören}}{\text{Anzahl der Tupel, die geladen worden sind}} \qquad \text{Gl: 3-4}$$

3.2.1.1 Festplattenarchitektur

Abbildung 3-3 zeigt die Komponenten einer herkömmlichen Festplatte. Eine Festplatte besteht üblicherweise aus mehreren Platten, die durch eine Spindel miteinander verbunden sind. Jede Platte hat zwei Oberflächen, die wiederum in Spuren unterteilt sind. Da eine Festplatte aus mehreren Platten besteht, werden die Spuren, die den gleichen Durchmesser haben, als Zylinder zusammengefasst. Eine einzelne Spur wird in Sektoren unterteilt, die auch als Block bezeichnet werden. In dieser Arbeit werden Sektoren und Blöcke synonym benutzt. Die Größe eines Blocks liegt bei aktuellen Festplatten zwischen 512 Byte und 16 KB. Ein Block ist die kleinste Adressierungseinheit, die von der Festplatte in den Arbeitsspeicher geladen werden kann, und wird über das Tupel (Zylinder, Platte, Block) spezifiziert.

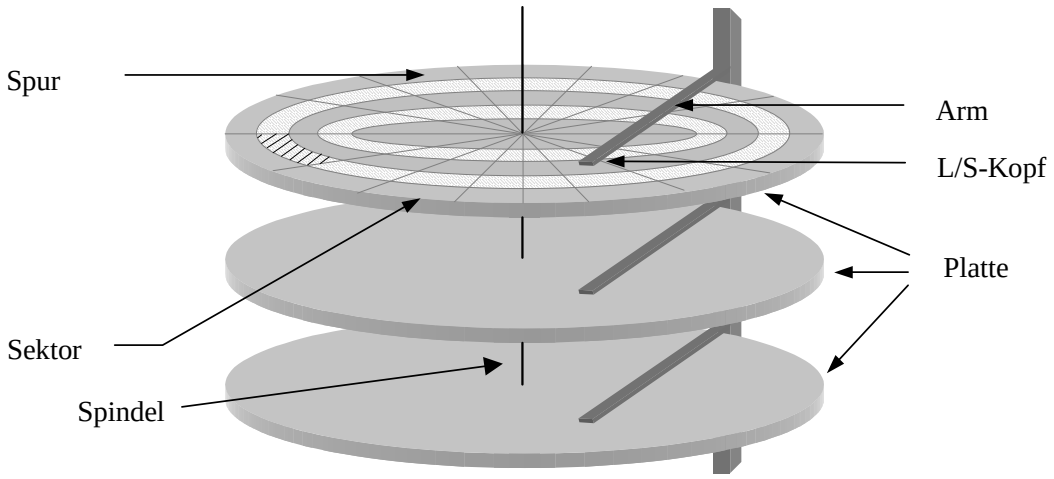


Abbildung 3-3: Komponenten einer Festplatte

Datenbanksysteme adressieren in der Regel eine feste Anzahl, üblicherweise 1,2,4 oder 8, zusammenhängende Blöcke als Seite. In dieser Arbeit wird zwischen Block und Seite nicht unterschieden.

Definition 3-2: Seite *p*

Eine Seite *p* ist ein Byte-Behälter mit fester Größe, z.B. Speicherseite oder Sekundärspeicherseite.

Definition 3-3: Inhalt einer Seite p

Der Inhalt einer Seite *p*, *content*(*p*), ist die Menge der Elemente, die in der Seite *p* enthalten sind:

$$\text{content(p)} = \{x_1, x_2, x_3, \ldots, x_n\}$$

Ist aus dem Zusammenhang ersichtlich, dass nicht die Seite als Behälter, sondern die Menge der Elemente gemeint ist, schreiben wir *p* statt *content*(*p*).

Die Anzahl κ der Tupel, die auf einer Seite *p* gespeichert werden können, bezeichnen wir als Kapazität der Seite. Die Größe einer Seite in Byte wird mit $p_{größe}$, eines Tupels mit $x_{größe}$ bezeichnet.

Definition 3-4: Kapazität

$$\kappa = \left\lfloor \frac{\text{Seitengröße}}{\text{Tupelgröße}} \right\rfloor = \left\lfloor \frac{p_{größe}}{x_{größe}} \right\rfloor$$

Die Anzahl der Tupel, die auf einer Seite *p* tatsächlich gespeichert sind, bezeichnen wir mit $|p|$. Somit gilt:

$$\kappa \geq |p| \qquad \text{Gl: 3-5}$$

Beispiel 3-2: Seitenkapazität bei der GfK

Das GfK DW, das für Messungen in dieser Arbeit herangezogen wird, ist auf einer Festplatte mit einer Blockgröße von 512 Byte gespeichert. Jede Sekundärspeicherseite hat eine Größe von 2 KB, so dass sie aus vier zusammenhängenden Blöcken besteht. Ein Datensatz der Faktentabelle besteht aus 14 Integern zu je 4 Bytes. Ein Datensatz benötigt also 56 Byte. Mit 2 KB Seiten können somit bis zu $\lfloor 2048/56 \rfloor$ = 36 Datensätze abgelegt werden. Normalerweise benötigt ein DBMS zusätzlichen Platz, um die Daten auf einer Seite zu verwalten. Für TransBase führt dies zu einer Kapazität von 31 Datensätzen pro Seite.

3.2.1.2 Kostenmodell für Sekundärspeicher

Dieser Abschnitt präsentiert ein Kostenmodell für den Sekundärspeicher. Das Kostenmodell wurde bereits vom Autor in [MarZB99] veröffentlicht.

Das Laden von Daten aus dem Sekundärspeicher in den Arbeitsspeicher wird als Übertragungskosten bezeichnet. Die Übertragungskosten kann man in drei Teilkosten -

Zugriffsbewegungszeit (seek time), Umdrehungswartezeit (rotational delay, latency) und Übertragungszeit (transfer time)- aufteilen. Es werden immer nur die durchschnittlichen Zeiten betrachtet, da z.B. die durchschnittliche Zugriffsbewegungszeit stark vom relativen Abstand der L/S-Kopf und der zu lesenden Seite abhängt. Die durchschnittliche Zugriffsbewegungszeit ist wie folgt definiert:

Definition 3-5: Durchschnittliche Zugriffsbewegungszeit

Die *durchschnittliche Zugriffsbewegungszeit* t_δ ist die durchschnittliche Zeit, die benötigt wird, um den Lese/Schreib–Kopf (L/S-Kopf) auf die richtige Spur zu setzen.

Die durchschnittliche Zugriffsbewegungszeit von heutigen Festplatten liegt bei 5 bis 7 ms. Nachdem der L/S-Kopf auf die richtige Spur positioniert wurde, entstehen nun die Kosten der Umdrehungswartezeit. Die *Umdrehungswartezeit* ist die Zeit, die benötigt wird, um die Spur um 360° zu drehen. Die durchschnittliche Umdrehungswartezeit ist wie folgt definiert:

Definition 3-6: Durchschnittliche Umdrehungswartezeit

Die *durchschnittliche Umdrehungswartezeit* t_λ ist die Zeit, die durchschnittlich benötigt wird, die Spur soweit unter dem L/S-Kopf zu bewegen, bis der Anfang der Seite *p* erreicht ist.

Die *durchschnittliche Umdrehungswarezeit* t_λ entspricht offensichtlich der halben *Umdrehungswartezeit*. Die durchschnittliche Umdrehungswartezeit beträgt derzeit ca. 4 ms. Der dritte Bestandteil ist die Block-Übertragungszeit. Die Übertragungszeit lässt sich als linearer Zusammenhang von Blockgröße und Transferrate ansetzen. Hierbei bezeichnet die Transferrate die Datenmenge, die pro Zeiteinheit übertragen wird:

Definition 3-7: Durchschnittliche Block-Übertragungszeit

Die durchschnittliche Block-Übertragungszeit t_τ ist :

$$t_\tau = \frac{Blockgröße}{Transferrate}$$

Die durchschnittliche Übertragungszeit ist die Zeit, die benötigt wird, um den Block von der Festplatte abzutasten und in eine Seite des Arbeitsspeichers zu laden. Sie liegt derzeit bei ca. 0,6 ms.

Das Kostenmodell wird vereinfacht, indem man die Zugriffsbewegungs- und Umdrehungszeit als Positionierungszeit zusammenfasst.

Definition 3-8: Durchschnittliche Positionierungszeit

Die *durchschnittliche Positionierungszeit* t_π ist die Summe aus der durchschnittlichen Zugriffsbewegungszeit t_σ und der durchschnittlichen Umdrehungszeit t_λ.

$$t_\pi = t_\sigma + t_\lambda$$

Beispiel 3-3: Datenblatt einer Festplatte
Die Messungen wurden in dieser Arbeit zum Teil auf einer Seagate 18,4 GB Festplatte mit der Bezeichnung ST 318416W durchgeführt. Technische Daten, die dem offiziellen Datenblatt entnommen wurden, sind in der folgenden Tabelle zusammengefasst:

Durchschnittliche Zugriffsbewegungszeit (average seek time)	5,9 (msec)
Durchschnittliche Umdrehungswartezeit (average latency)	4,17(msec)
Durchschnittliche Übertragungsrate (transfer rate)	40,0(Mbytes/sec)
Blockgröße	2 Kbyte
Sektorgröße	512 byte
Unformatierte Kapazität	18,4 GB

Wie man aus dem offiziellen Datenblatt entnehmen kann, beträgt die durchschnittliche Positionierungszeit t_π = 10,07 ms. Die durchschnittliche Übertragungszeit für eine Seite p mit einer Seitengröße von 2KB beträgt $t_\tau = \frac{2\ \text{kbyte}}{40\ \text{Mbyte/s}} = 0{,}1\ \text{ms}$.

Die Positionierungszeit beläuft sich im
Beispiel 3-3 auf 99 Prozent der Gesamtkosten. Das wahlfreie Lesen von Seiten (random page access) ist somit eine relativ teure Operation in Datenbanksystemen. Wir definieren das wahlfreie Lesen wie folgt:

Definition 3-9: Durchschnittliche wahlfreie Zugriffszeit (Random access)

Die *wahlfreie Zugriffszeit* $t_{E/A\text{-}ran}$ ist die Zeit, die benötigt wird, um eine beliebige Seite vom Sekundärspeicher in den Arbeitsspeicher zu laden:

$$t_{E/A\text{-}ran} = t_\pi + t_\tau$$

Beispiel 3-4: Durchschnittlicher wahlfreier Zugriff
Ein durchschnittlicher wahlfreier Zugriff auf eine Seagate 18.4 GB Festplatte beträgt ca. 5,9 ms + 4,17 + 0,1 ≈ 10,2 ms.

Im Folgenden wird nur mit durchschnittlichen Zeiten gearbeitet. Aktuelle Betriebsysteme, wie z.B. UNIX, LINUX und Windows XP, lesen bei einer E/A-Operation mehrere aufeinanderfolgende Seiten in den Arbeitsspeicher, um die durchschnittliche Zugriffszeit zu reduzieren. Hier steht wieder die Heuristik des Lokalitätsprinzips Pate. Die Strategie wird in der Literatur als Prefetching bezeichnet.

Definition 3-10: Prefetching, E/A-Einheit

Die Menge aufeinanderfolgender Seiten, die durch eine E/A-Operation in den Arbeitsspeicher geladen wird, wird als *Prefetching* bzw. *E/A-Einheit* bezeichnet. Die Kardinalität wird mit *C* bezeichnet.

Die Zugriffszeit für *C* Seiten ergibt sich aus der einmaligen Positionierungszeit t_π und der Übertragungszeit t_τ. Somit sind die Kosten für eine E/A-Operation

$$c_{E/A} = t_\pi + C\ t_\tau \qquad \text{Gl: 3-6}$$

In dieser Arbeit wird für die theoretische Analyse eine Prefetchfaktor von 8 verwendet. Dieser Faktor wurde durch Messungen belegt.

3.2.2 Clusterung, Indizierung

Um die E/A-Kosten zu reduzieren und das Auffinden von Daten in großen Datenbanksystemen zu beschleunigen, werden besondere Indexstrukturen (Datenstrukturen) verwendet. In diesen Strukturen werden *Tupel-Identifikatoren* (*TID*) verwaltet. Ein TID besteht aus zwei Teilen: einer *Seitennummer* und einer *Positionsnummer* in der internen Datensatztabelle, die auf das entsprechende Tupel verweist. Die zusätzliche Indirektion ist nützlich, wenn die Seite intern reorganisiert werden muss. Allgemein wird ein Index und ein Indexschlüssel wie folgt definiert:

Definition 3-11: Index; Indexschlüssel

> Ein Index ist eine Hilfsstruktur (Datenstruktur) für den direkten Zugriff auf dem Datensatz einer Tabelle. Die Datensätze werden dabei durch einen Indexschlüssel identifiziert.

Der Indexschlüssel stellt somit das Suchkriterium da. Der Begriff hat jedoch nichts mit dem bisher eingeführten logischen Schlüssel einer Tabelle zu tun (siehe 3.1), der ein Tupel eindeutig identifiziert. Er ist ein Attribut oder eine Attributmenge einer Tabelle *R*, über die ein Index aufgebaut wird. Ein Index ist somit nicht unbedingt eindeutig und kann somit auch Duplikate enthalten. Mit einem Index kann man über einen Schlüssel direkt auf einen Datensatz zugreifen. Ziel ist es nun, Einschränkungen, die durch eine Anfrage definiert wurden, auszunutzen, um die Anzahl der Plattenzugriffe zu reduzieren. Indexstrukturen spielen eine entscheidende Rolle im Bereich der Anfrageoptimierung. Um die Vorteile dieser Strukturen optimal auszunutzen, ist eine genaue Kenntnis der E/A-Kosten unerlässlich. Deshalb muss für jede Zugriffsmethode ein möglichst genaues Kostenmodell existieren, mit dem der Aufwand abgeschätzt werden kann.

Neben Indexstrukturen allgemein ist die physische Lokalität (Clustering) eine weitere Technik der Leistungssteigerung.

Definition 3-12: Clustering

> Das *Clustering* speichert Daten, die logisch zusammen gehören und deshalb oft zusammen verarbeitet werden, physisch zusammen.[Date95].

Grundsätzlich können Daten nur nach einem Kriterium geclustert werden. In dieser Arbeit werden drei Stufen der physikalischen Clusterung unterschieden, die untereinander eine Hierarchie bilden. Die kleinste adressierbare Einheit, die wir im Datenbanksystem betrachten, ist ein Datensatz (Tupel) und dementsprechend das Tupel-Clustering.

Definition 3-13: Tupel-Clustering

> Das *Tupel-Clustering* speichert logisch zusammengehörige Datensätze einer oder mehrerer Relationen auf einer Seite physisch zusammen.

Durch das Tupel-Clustering wird nur das Clustering innerhalb einer Seite garantiert. Falls es durch das Einfügen eines Datensatzes zu einem Überlauf der betroffenen Seite kommt,

wird diese in zwei Seiten aufgeteilt. Die Lage der neuen Seite auf dem Sekundärspeicher ist abhängig von der aktuellen Struktur der Festplatte beim Einfügezeitpunkt, so dass eine physische Clusterung nicht gewährleistet ist. Ist zusätzlich eine Clusterung bezüglich der Seiten gefordert, sprechen wir von Seiten-Clustering.

Definition 3-14: Seiten-Clustering

> Eine Menge von Seiten ist *seitengeclustert* genau dann, wenn die Tupel auf den Seiten *tupelgeclustert* sind und die Seiten der logischen Struktur entsprechend physisch zusammenhängend auf der Festplatte abgelegt sind.

Da das Seiten-Clustering auf dem Tupel-Clustering basiert, bildet es eine Cluster-Hierarchie, *kein Clustering*, *Tupel-Clustering* und *Seiten-Clustering*. Abbildung 3-4 zeigt die drei verschiedenen Clusterungstypen. Gegeben ist eine Menge von Zeichenketten, die nach der lexikographischen Ordnung geclustert werden sollen.

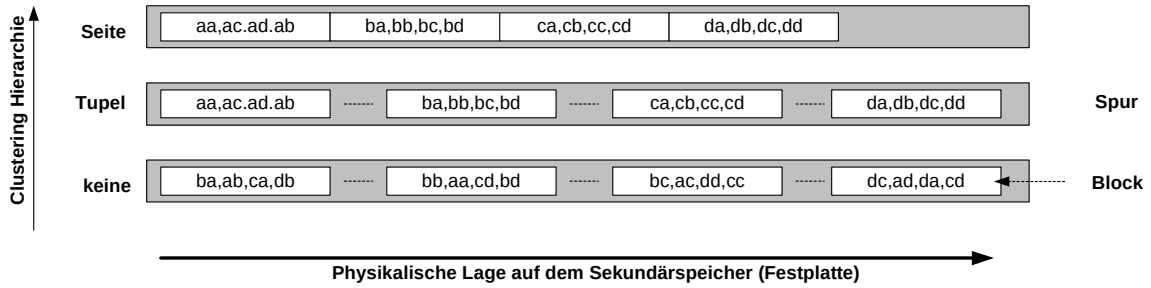


Abbildung 3-4: Cluster-Hierarchie

Beispiel 3-5: Clusterung
Abbildung 3-4 zeigt die drei verschiedenen Clusterungstypen. Der Prefetchfaktor C sei 2. Um nun die Tupel, die das Prädikat $\varphi = \{e \mid e \in [aa, bd]\}$ erfüllen, in den Arbeitsspeicher zu laden, benötig man beim Seitenclustern 1 E/A-Operation, 2 E/A-Operationen beim Tupelclustern, und 4, wenn keine Clusterung vorhanden ist.

Um n Tupel (mit $n \geq 2$) vom Sekundärspeicher mit einer Seiten-Kapazität κ und Prefetchfaktor C zu lesen, ergeben sich mit dem Kostenmodell aus Abschnitt 3.2.1.2 folgende Formeln:

- Wahlfreier Zugriff:

$$c_{wf} = n \cdot c_{E/A} \qquad \text{Gl: 3-7}$$

- Tupel-Clustering:

$$c_{TC} = \left(\left\lceil \frac{n}{\kappa} \right\rceil + 1 \right) \cdot c_{E/A} \qquad \text{Gl: 3-8}$$

- Seiten-Clustering

$$c_{SC} = \left(\left\lceil \frac{n}{C \cdot \kappa} \right\rceil + 1 \right) \cdot c_{E/A} \qquad \text{Gl: 3-9}$$

Hierbei handelt es sich also um lineare Kostenfunktionen, den schlechtesten Fall (worst case). Der proportionale Leistungsgewinn von einer Hierarchiestufe zur anderen ist nur abhängig von der Kapazität κ und dem Prefetchfaktor *C*. Vom wahlfreien Zugriff auf Tupel-Clustering ist somit ein Leistungsgewinn von Faktor κ möglich sowie zum Seiten-Clustering ein Faktor von $C \cdot \kappa$

Beispiel 3-6:
Die Faktentabelle der GfK enthält 42.867.092 Tupel. Bei einer Seitengröße von 2 KB ergibt sich eine Kapazität κ von 31 (siehe Beispiel 3-2) und somit 1382810 Seiten. Eine E/A-Operation benötigt $c_{E/A}$ = 10 ms. Der Prefetchfaktor beträgt C = 8. Die Zeiten für die unterschiedlichen Zugriffsmethoden sind in der folgenden Tabelle aufgeführt:

Wahlfreier Zugriff	c_{wf} = 4,96 Tage
Tupel-Clustering	c_{TC} = 3,84 Stunden
Seiten-Clustering	c_{SC} = 28 min

Wir unterscheiden zwischen *Primär-* und *Sekundärindexen*. Primindexe legen die physische Anordnung der indizierten Daten auf den Sekundärspeicher fest und erzeugen somit mindestens Tupel-Clustering. Daher kann es für jede Tabelle nur einen Primärindex geben, aber mehrere Sekundärindexe. Will man dennoch den Vorteil des Clusterings für verschiedene Indexe auf eine Tabelle haben, muss die Tabelle durch Replikation redundant für jeden weiteren Primärindex abgelegt werden, was jedoch zu einem erheblichen Speicherbedarf führt.

Definition 3-15: Primärindex, Sekundärindex

Ein Index ist ein *Primärindex*, falls die Tupel auf dem Sekundärspeicher nach seinen Indexattributen geclustert sind, sonst ist er ein Sekundärindex.

Im Sekundärindex werden für einzelne Attribute oder Attributmengen Indexe (*Inverted File*, IF) angelegt. Sie stellen somit eine Replikation der TIDs da. Es werden also nur Verweise auf Datensätze in die Struktur eingetragen. Wird eine Bereichsanfrage mit einen Sekundärindex ausgewertet, kommt es zu einem wahlfreien Zugriff für jedes einzelne Tupel.

3.3 Klassifikation von Zugriffspfaden

Allgemein kann man Datenstrukturen nach ihrer Dimensionalität klassifizieren. Hierunter versteht man die Anzahl der Dimensionen (Attribute), die zu dem Schlüssel, der von der Datenstruktur intern verwendet wird, beitragen. Zu beachten ist, dass es sich hier nicht notwendigerweise um den logischen Schlüssel des Datensatzes handelt. Man unterteilt sie grundsätzlich in 1 und *n*-dimensionale Zugriffsstrukturen.

Neben der Dimensionalität ist die Unterstützung von *physischer Clusterung* ein weiteres grundsätzliches Klassifikationsmerkmal.

Die eindimensionalen Strukturen können wiederum in baumartige, wie z.B. der B-Baum [BayM72] und seinen Varianten [Com79], der einen wahlfreien und sortierten Zugriff

unterstützt, und gestreute (hash) Zugriffsmethoden [Lar83] für einen wahlfreien Zugriff und einen sequentiellen Zugriff unterteilt werden.

Die d-dimensionalen Datenstrukturen werden allgemein nach der Art der Daten klassifiziert [GaeG98]. Diese Daten werden als räumliche Daten bezeichnet. Sie bestehen üblicherweise aus komplexen Strukturen. Ein Objekt kann hierbei entweder aus einem einzigen Punkt oder aus 1000 Polygonen bestehen, die willkürlich über das Universum verteilt sind.

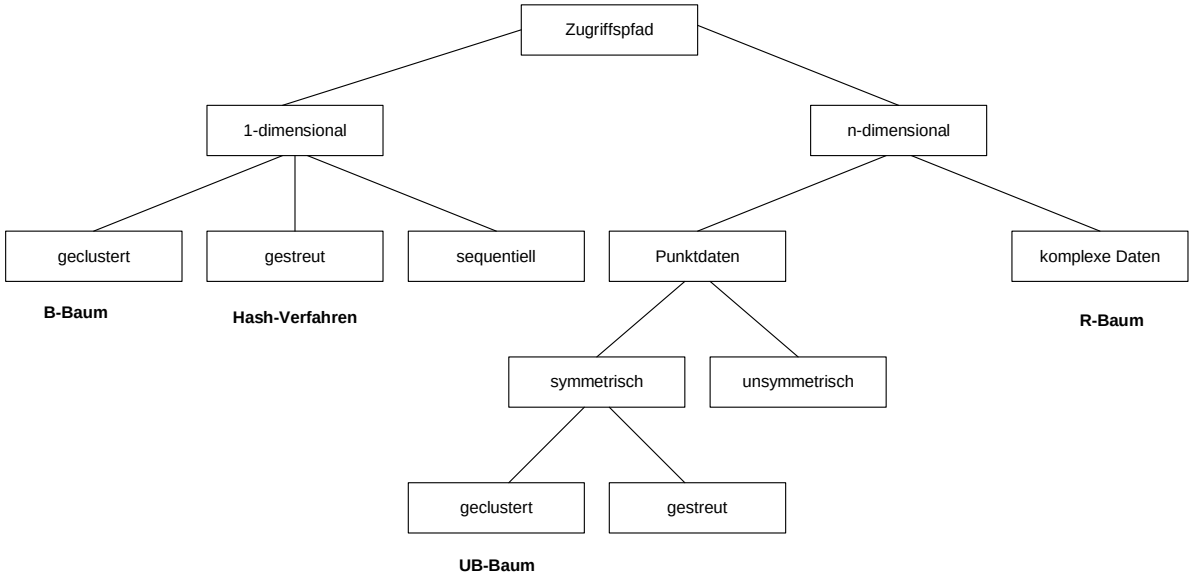


Abbildung 3-5: Klassifizierung der Zugriffspfade

Handelt es sich bei den Daten um Punkte im d-dimensionalen Raum, können spezielle d-dimensionale Punktzugriffsmethoden herangezogen werden. Als weitere Unterteilung kann die Symmetrie der Schlüsselattribute benutzt werden. Allgemein können *n-dimensionale* Zugriffspfade nach ihrer Symmetrie klassifiziert werden. Je nachdem, ob eine Dimension anderen Dimension gegenüber bevorzugt wird oder nicht, spricht man von *unsymmetrischen* oder *symmetrischen* Zugriffspfaden.

3.4 B-Baum

Ein B-Baum [BayM72] ist ein vollständig balancierter Mehrwegbaum, der sowohl Punktanfragen als auch Bereichsanfragen im 1-dimensionalen Raum sehr effizient abarbeitet. Hierbei werden nur solche Seiten in den Arbeitspeicher geladen, die zum Ergebnisraum gehören.

Definition 3-16: B-Baum[BayM72]

Sei $h \geq 0$ und $k \in \mathbb{N}$. Ein Baum T ist ein B-Baum der Klasse $\tau(k,h)$ genau dann, wenn T ist leer ($h = 0$) oder die folgenden Bedingungen erfüllt sind:

- Jeder Pfad von der Wurzel zu einem Blatt hat die gleiche Länge h (Höhe), so dass der Baum vollständig balanciert ist. h ist die Anzahl der Knoten im Pfad.
- Jeder Knoten, mit Ausnahme von der Wurzel und den Blättern hat mindestens k+1 Kinder. Die Wurzel ist entweder ein Blatt oder hat mindestens zwei Kinder.
- Jeder Knoten hat höchstens 2k+1 Kinder.

Der B*-Baum [Wed74], [Knu98v3] und der Präfix B-Baum [BayU77] sind Varianten des B-Baums. Die Innovation dieser beiden Varianten ist die Trennung zwischen Daten und Index, wobei beim Präfix B-Baum der Index noch komprimiert wird. Dadurch wird ein erheblich größerer Verzweigungsgrad (fan-out) erreicht, was gleichbedeutend mit einer Reduzierung der Baumhöhe bei gleicher Anzahl von Daten ist. Als externe Speicherstruktur reduziert der B-Baum somit die E/A-Komplexität. Heutige Datenbanksysteme verwenden dementsprechend hauptsächlich B*-Baum-Implementierungen. Die E/A-Kosten um ein Knoten in einem B-Baum zu lesen entspricht im schlechtesten Fall der Höhe h des Baumes, d.h., der Knoten ist ein Blatt. Für einen B-Baum mit I Schlüsseln ergeben sich somit im schlechtesten Fall folgende E/A-Kosten $c_{E/A\text{-}lesen}$:

$$c_{E/A-lesen} = h = 1 + \log_{k+1} \frac{I+1}{2} \qquad \text{Gl: 3-10}$$

Das Einfügen besteht aus einem Lese- und Schreibvorgang. Im schlechtesten Fall sind alle Knoten des Pfades mit 2k+1 Schlüssel besetzt, so dass es auf jeder Ebene zu einer Teilung (Split) kommt. Dies führt dazu, das für h Knoten $2h$ und eine neue Wurzel entstehen. Somit ergeben sich im schlechtesten Fall für das Einfügen in einen B-Baum folgende E/A-Kosten $c_{E/A\text{-}schreiben}$:

$$c_{E/A-schreiben} = \underbrace{h}_{lesen} + \underbrace{2h+1}_{schreiben} = 3h + 1 = 1 + 3c_{E/A-lesen} \qquad \text{Gl: 3-11}$$

Liegt der Schlüssel, der gelöscht werden soll, nicht in einem Blatt, wird der Schlüssel durch seinen Nachfolger ersetzt und der Nachfolger im Blatt gelöscht. Liegt der Schlüssel bereits im Blatt, kann er sofort aus dem Blatt entfernt werden. Falls durch das Entfernen des Schlüssels ein Unterlauf entsteht, d.h., der Knoten enthält weniger als k Schlüssel, kommt es zu einem Ausgleich der Schlüssel durch den Nachbarn. Hierbei werden die Schlüssel auf die beiden Knoten gleichmäßig verteilt. Die Operation kann jedoch scheitern, falls der Nachbar auch bereits nur k Schlüssel enthält. Dann werden die beiden Knoten und der Schlüssel des Vaterknoten, der als Separator für die beiden Knoten fungierte, zu einem Knoten zusammengefasst. Da jedoch ein Schlüssel aus dem Vaterknoten in den Knoten aufgenommen wird, kann es auch im Vaterknoten zu einem Unterlauf kommen. Der Unterlauf kann sich im schlechtesten Fall bis zur Wurzel fortsetzen, so dass sich die Höhe des B-Baums um 1 reduziert. Für die E/A-Kosten ergibt sich im schlechtesten Fall somit $c_{E/A\text{-}löschen}$:

$$c_{E/A-löschen} = \underbrace{2h-1}_{lesen} + \underbrace{h+1}_{schreiben} = 3h = 3c_{E/A-lesen} \qquad \text{Gl: 3-12}$$

3.5 Hash-Verfahren

Neben den baumstrukturierten stellen die *gestreuten Zugriffspfade* die zweite Klasse dar, die in der Literatur als Hash-Verfahren bezeichnet wird. Die Hash-Verfahren basieren auf Schlüsseltransformationen. Diese Verfahren werden allgemein in statische und dynamische Verfahren unterteilt.

Der Grundgedanke statischer Verfahren ist die direkte Berechnung der Speicheradresse eines Satzes aus seinem Schlüssel, ohne auf weitere Hilfsstrukturen zurückzugreifen. Dies stellt jedoch den idealisierten Fall dar, der in vielen Situationen nicht garantiert werden kann.

Eine Hash-Funktion *H* bildet die Menge der möglichen Schlüsselwerte $K = \{D_0 \times D_1 \times ... \times D_{d-1}\}$ auf die Speicheradresse $A = \{0, ..., n-1\}$ ab.

$$H: K \rightarrow A \qquad \text{Gl: 3-13}$$

Aus der Vielzahl der vorgeschlagenen Verfahren [OttW96],[Knu98v3] sind Hash-Verfahren, die für Externspeicher auf Seitenbasis angewendet werden können, von besonderem Interesse. Dabei wird die Bildmenge *A* als relative Seitennummer interpretiert, so dass die berechnete Nummer einer Seite einem zusammenhängenden Segmentbereich zugeordnet werden kann. Da üblicherweise $A << K$ gilt, kann es zu Kollisionen kommen, d.h., zwei Schlüssel werden auf dieselbe Speicheradresse $a \in A$ abgebildet, die durch eine Kollisionsbehandlung aufgelöst werden.

Ein interessantes Verfahren stellt das externe Hashing mit Separatoren dar [LarK84], [Lar88]. Trotz auftretender Kollisionen bei der Adressberechnung garantiert dieses Verfahren, dass jeder Satz genau mit einem Externspeicherzugriff aufgefunden werden kann. Dazu wird eine relativ kleine Menge an zusätzlichem internen Speicherplatz (etwa ein Byte pro Bucket) für die Separatortabelle benötigt. Um den Zugriffsfaktor 2 zu erzielen, muss jedoch auf den Einsatz von verketteten Überlaufbereichen verzichtet werden.

Allgemein kann jedoch für statische Hash-Verfahren festgestellt werden, dass wegen der großen Abhängigkeit der Performanz, wie Zugriffs- und Belegungsfaktor, von der Schlüsselverteilung und Parametereinstellung kein allgemeingültiges Verfahren angegeben werden kann. Ist die optimale Parametereinstellung verfehlt, kann es allgemein zu deutlichen Leistungseinbrüchen kommen.

Für stark wachsende Datenbestände bringt die statische Zuweisung des Hash-Bereichs gravierende Nachteile mit sich. Um zu schnelles Überlaufen zu vermeiden, muss er bei der Initialisierung genügend groß dimensioniert werden, was zu einer erheblichen Speicherverschwendung führt. Wird das geplante Fassungsvermögen des Hash-Bereichs überschritten, ist die Speicherstruktur entweder nicht mehr aufnahmefähig oder neue Daten werden im Überlaufbereich aufgefangen, was zu einer schlechten Performanz führt. Die schlechte Performanz kann nur durch periodische Reorganisierung der Daten behoben werden.

Dynamische Hash-Verfahren erlauben das Wachsen und Schrumpfen der Datentabelle ohne Überlauftechniken, die eine statische periodische Reorganisation mit vollständigem Rehashing benötigen. Dabei garantieren sie eine hohe Speicherplatzauslastung und eine Zugriffszeit von maximal zwei Seiten. Hierzu werden entweder Indizes herangezogen, die den Überlauf lokal beheben, wie z.B. das „erweiterbare Hashing“ [FagN+79], oder Verfahren, die das Überlaufproblem von Verwendung von Indizes lösen. Hierbei wird versucht hinsichtlich der Hash-Datei eine globale Lösung, wie z.B. das „*lineare Hashing“* [Let80], zu finden.

Durch die Transformation des Schlüssels auf eine Speicheradresse geht die Ordnung verloren, so dass eine physische Clusterung nach den Schlüsselattributen *K* im Allgemeinen nicht möglich ist. Somit können Hash-Verfahren nur für Punktanfragen effizient eingesetzt werden. Bereichsanfragen können nicht unterstützt werden, so dass Hash-Verfahren zum Vergleich in dieser Arbeit nicht herangezogen werden.

3.6 UB-Baum

Der UB-Baum [Bay96], eine Variante des zkd-B-Baums [OreM84], benutzt eine raumfüllende Kurve (Z-Kurve), um das mehrdimensionale Universum *U* zu partitionieren. Der wesentliche Vorteil liegt darin, dass der UB-Baum die räumliche Nähe im d-dimensionalen Raum relativ gut erhalten kann (siehe [Mar99]). Der UB-Baum organisiert (clustert) die Daten bezüglich der Z-Kurve. Das Kernkonzept hierbei besteht darin, die Seiten des UB-Baums auf Z-Intervalle abzubilden. Diese Z-Intervalle werden als Regionen bezeichnet.

Diese Arbeit betrachtet ein Tupel als einen Punkt im mehrdimensionalen Raum. Das Universum *U* ist ein mehrdimensionaler Raum, der durch die Domänen einer Menge von Attributen aufgespannt wird. Jede Domäne definiert eine *Dimension*. Der Wert eines Attributs bestimmt somit die Koordinate eines Punkts im mehrdimensionalen Raum. Diese bijektive Abbildung von einem Tupel auf einen Punkt führt dazu, dass folgende Bezeichnungen als Synonym in dieser Arbeit verwendet werden:

- mehrdimensionale Domäne, mehrdimensionaler Raum Universum
- Relation, Tabelle, Teilmenge des mehrdimensionalen Raums
- Tupel, Datensatz, Punkt
- Attribut, Spalte, Koordinate, Dimension
- Stelligkeit, Dimensionalität

3.6.1 Z-Kurve

Da die Z-Kurve ein Kernkonzept des UB-Baums ist und die entwickelten Algorithmen darauf basieren, wird das Konzept der raumfüllenden Kurven (space filling curve [Sag94]) vorgestellt.

Im d-dimensionalen Raum gibt es keine natürliche totale Ordnung, so dass man z.B. zwei Punkte im Raum in der Regel nur auf ihre Gleichheit prüfen kann. Das Konzept der raumfüllenden Kurven jedoch erzeugt auf einen d-dimensionalen Raum eine totale Ordnung (siehe Definition 1-2), wodurch die Vergleichsoperationen „>“ und „<“ anwendbar sind. Allgemein kann man sagen, dass ein mehrdimensionaler Raum auf einen 1-dimensionalen abgebildet wird. Dies hat zur Folge, dass man zur Verwaltung des Raums eine 1-dimensionale Zugriffsmethode, wie z.B. den B*-Baum, verwenden kann.

Das Grundprinzip besteht darin, das Universum *U* zunächst durch ein gleichförmiges Raster zu partitionieren, wobei jede Zelle durch eine eindeutige Nummer, die ihre Position in der totalen Ordnung definiert, identifiziert wird. Somit kann ein mehrdimensionales Objekt, wie z.B. das Tupel $x = (x_1,x_2 , ...,x_d)$ durch die Nummer der Zelle, zu dem es gehört, indexiert werden.

In Abbildung 3-6 sind drei Beispiele für raumfüllende Kurven für den 2-dimensionalen Raum zu sehen, die Z-Kurve [OreM84], die Hilbert-Kurve [Jag90] und der Gray-Code[Fal88]. Die erste Zeile zeigt das Grundmuster der Nummerierung und damit die Ordnung für den 2-dimensionalen Fall, nachdem das Universum zum ersten Mal geteilt wurde. Jede Zelle wiederum kann bei Bedarf unter Anwendung desselben Teilungsmusters separat wieder geteilt werden. Dieser rekursive Vorgang kann so oft wiederholt werden, bis die benötigte Auflösung erreicht ist. Die zweite und dritte Zeile zeigen die vollständige Unterteilung in der zweiten und dritten Rekursionsstufe.

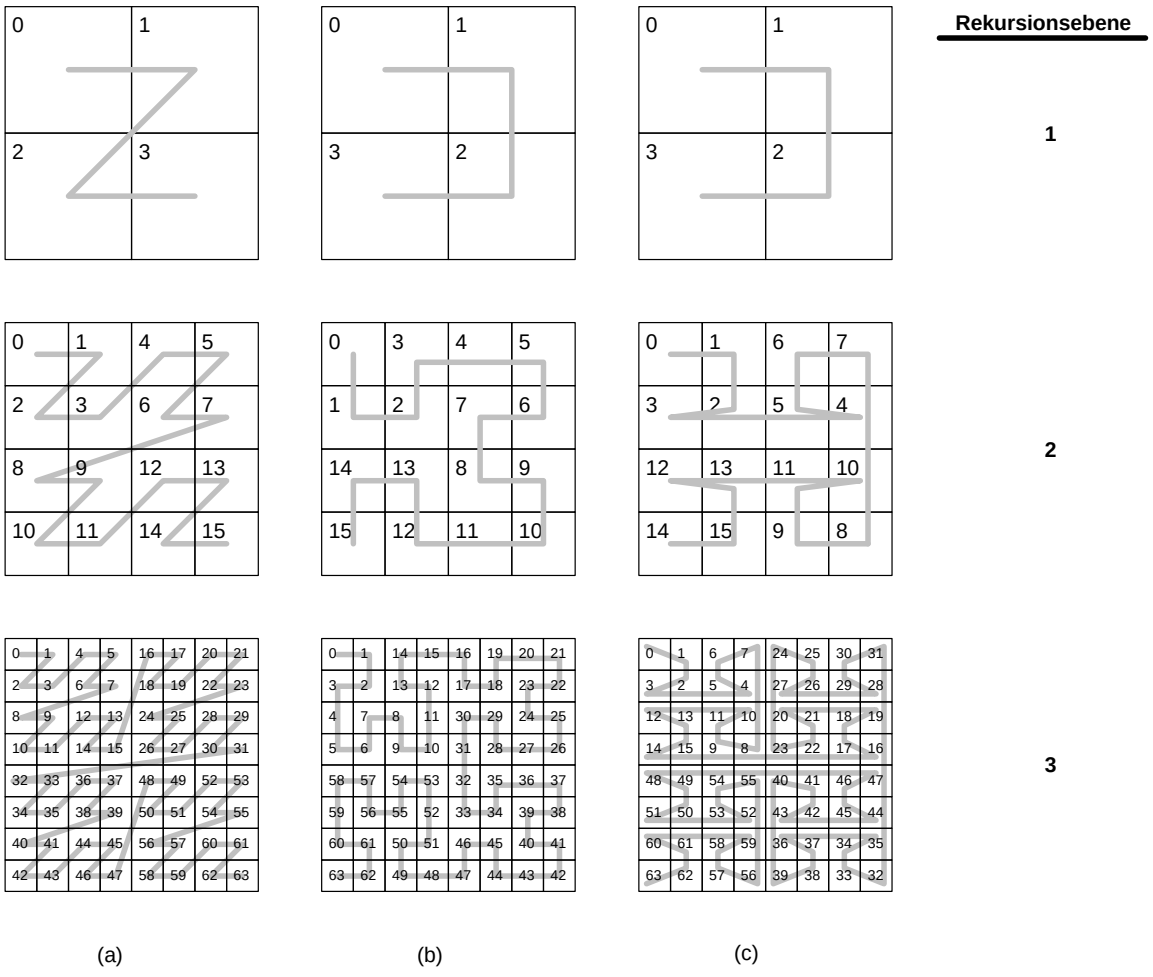


Abbildung 3-6: (a) Z-Kurve, (b) Hilbert-Kurve, (c) Gray-Code

Für den d-dimensionalen Raum ergibt sich für die Z-Kurve also folgendes Schema: Pro Stufe wird durch Halbieren aller Dimensionen der Raum in

$$P=2^d \qquad \texttt{Gl: 3-14}$$

Unterräume (Subwürfel) zerlegt, die nach der Z-Ordnung angeordnet werden (siehe Abbildung 3-6.a). Dieses Schema kann auf jeden Unterraum beliebig oft angewendet werden, so dass bei einer vollständigen Unterteilung der Stufe *s* genau

$$P = 2^{d \cdot s} \qquad \text{Gl: 3-15}$$

Unterräume existieren.

3.6.2 Basisraum

Da wir uns ausschließlich mit endlich diskreten Räumen beschäftigen, lassen wir die Forderung der Stetigkeit für raumfüllende Kurven fallen und betrachten nur noch diskrete raumfüllende Funktionen. Ohne Beschränkung der Allgemeinheit betrachten wir nur Teilmengen $\Omega_i = \{0,1,...,r_i\text{-}1\}$von $\mathbb{N}$, da man jede endliche, total geordnete Domäne $D_i = (\{a_1, ... , a_r\}, <)$ mit $r = |D_i| \leq r_i$ monoton auf Ω_i mit

$$f{:}D_i \rightarrow \Omega_i, \text{ und } f(a_j) < f(a_k) \Leftrightarrow a_j <_i a_k$$

abbilden kann. Das kartesische Produkt der einzelnen Domänen Ω_i der Dimension bezeichnen wir als Basisraum Ω.

Definition 3-17: Basisraum

Gegeben ist eine Menge von Dimensionen mit den Wertebereichen (Domäne) Ω_i, mit $\Omega_i = \{0,1,...,r_i\text{-}1\} \subset \mathbb{N}$. Zusätzlich fordern wir für $r_i = 2^s$ mit $s \in \mathbb{N}$. Der *Basisraum* Ω ist das kartesische Produkt der einzelnen Domänen:

$$\Omega = \Omega_1 \times ... \times \Omega_d = \{0,...,r_1 - 1\} \times ... \times \{0,...,r_d - 1\}.$$

Der Basisraums stellt somit den vollständig besetzten Raum mit der Kardinalität

$$|\Omega| = \prod_{i=1}^{d} r_i \qquad \text{Gl: 3-16}$$

dar. Eine Teilmenge des Basisraums wird als *Teilraum* oder *Unterraum* bezeichnet.

Unabhängig von der Dimensionalität bezeichnen wir die räumliche Ausdehnung als Volumen. So wird z.B. auch die Ausdehnung im 2-dimensionalen Raum nicht als Fläche bezeichnet, sondern als Volumen. Wir definieren das Volumen als normierten Wert bezüglichen Ω. Im diskreten Modell besitzt ein Punkt im Basisraum ein Volumen von $1/|\Omega|$.

Definition 3-18: Normalisiertes Volumen

Das *normalisierte Volumen V* eines Teilraums [[*x*,y]] im *d*-dimensionalen Basisraum Ω ist:

$$V([[x, y]] = \prod_{i=1}^{d} \frac{(y_i - x_i) + 1}{r_i}$$

Das *normalisierte Volumen V* für die leere Menge ∅ ist:

$$V(\varnothing) = 0$$

Die einzelnen Relationen, die auf dem *d*-dimensionalen Raum definiert sind, sind Teilmengen von Ω, d.h. $R \subseteq \Omega$.

Für den Basisraum Ω können wir nun die raumfüllende Funktion wie folgt definieren:

Definition 3-19: Raumfüllende Funktion, Ordnungszahl

Sei $S \subset \mathbb{N}$ und $a \in S$ sowie $x \in \Omega$. Eine Funktion:

$$f: S \rightarrow \Omega$$

ist eine *raumfüllende Funktion* genau dann, wenn $f: a \rightarrow x = f(a)$ bijektiv ist. $a \in S$ bezeichnet man als Ordnungszahl.

Die entscheidende Eigenschaft der raumfüllenden Funktion ist die Erzeugung einer totalen Ordnung auf einem *d*-dimensionalen Raum. Diese Eigenschaft wird im folgenden Satz näher beschrieben:

Satz 3-1:

Eine *raumfüllende Funktion* erzeugt eine 1-dimensionale Ordnung auf einen mehrdimensionalen Raum.

Beweis:

Die Ordnungsrelation ≤ auf ℕ definiert eine totale Ordnung auf Ω. Da nach Definition 3-19 jede raumfüllende Funktion eine bijektive Abbildung von $S \rightarrow \Omega$ ist ($S \subset \mathbb{N}$), erzeugt sie demzufolge eine totale Ordnung

$$f(0) < f(1) < f(2) < \ldots$$

auf Ω.
q.e.d.

Die Ordnungszahl für einen Punkt im d-dimensionalen Raum bezeichnen wir als Adresse. Die Funktion, mit der diese Adresse berechnet werden kann, ist die inverse Funktion der raumfüllenden Funktionen.

Definition 3-20: Adresse

Eine Adresse $\alpha \in (S \subset \mathbb{N})$ ist die Ordnungszahl für ein Tupel $x \in \Omega$ bezüglich der inversen Funktion f^{-1} der raumfüllenden Funktion f:

$$f^{-1} : x \mapsto \alpha = f^{-1}(x);\ x \in \Omega, y \in S$$

Da der UB-Baum die Z-Kurve zur Partitionierung des Raums benutzt, definieren wir nun die Z-Adresse Z(x) für die raumfüllende Z-Kurve. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass für die Kardinalzahlen der einzelnen Dimensionen gilt: $r_1 = r_2 = \ldots = r_d$ und $s = \log_2 r$. Des Weiteren benutzen wir die binäre Darstellung eines Tupels x. Um ein Bit in einem Tupel x zu identifizieren, benutzen wir folgende Notation:

$$x_{\underbrace{i}_{\substack{\text{Element } i \\ \text{von Tupel } x}},\ \underbrace{j}_{\substack{\text{Bit } j \text{ von} \\ \text{Element } i}}} \qquad \text{Gl: 3-17}$$

Die Z-Adresse wird wie folgt definiert:

Definition 3-21: Z-Adresse

Die Z-Adresse α des Tupels $x \in \Omega$ ist die Ordnungszahl

$$Z(x) = \sum_{j=0}^{s-1} \sum_{i=1}^{d} x_{i,j} \cdot 2^{j \cdot d + i - 1} = \alpha$$

Für die binäre Darstellung ergibt sich somit:

$$\underbrace{x_{d,s-1} x_{d-1,s-1} \ldots x_{1,s-1}}_{\text{Stufe 0}} \cdot \underbrace{x_{d,s-2} x_{d-1,s-2} \ldots x_{1,s-2}}_{\text{Stufe 1}} \cdots \underbrace{x_{d,0} x_{d-1,0} \ldots x_{1,0}}_{\text{Stufe s-1}} \qquad \text{Gl: 3-18}$$

Hierbei bezeichnet d die Anzahl der Dimensionen und

$$s = \log_2 r \qquad \text{Gl: 3-19}$$

die Anzahl der Bits, um den Wertebereich der jeweiligen Dimension A_i darzustellen. Eine Stufe bezeichnet die Bits der Z-Adresse, die in den jeweiligen Dimensionen dieselbe Bitposition j haben. Die Nummerierung der Stufen beginnt bei den höchstwertigen Bits mit Stufe 0.

Eine Z-Adresse kann also wie folgt interpretiert werden: Für jede Stufe wird der jeweilige Teilraum $[0, 2^d - 1]$ angegeben, in welchem der Punkt liegt.

Da jedem Bit $x_{i,j}$ eines Tupels x eindeutig ein Bit der Z-Adresse β zugeordnet wird, ist Z(x) eine bijektive Abbildung. Die Umkehrfunktion $Z^{-1}(\beta) = x$ ist also wie folgt definiert:

Satz 3-2:Z^{-1}

Gegeben ist eine Z-Adresse $\alpha \in [0,2^{sd}-1]$ mit der binären Darstellung

$$\underbrace{\alpha_{d(s-1)+(d-1)}\alpha_{d(s-1)+(d-2)}\ldots\alpha_{d(s-1)}}_{\text{step } 0}\,.\underbrace{\alpha_{d(s-2)+(d-1)}\alpha_{d(s-2)+(d-2)}\ldots\alpha_{d(s-2)}}_{\text{step } 1}\ldots\underbrace{\alpha_{d-1}\alpha_{d-2,}\ldots\alpha_{0}}_{\text{step } s-1}$$

Die Umkehrfunktion Z^{-1} ist dann:

$$Z^{-1}(\alpha) = (x_1, \ldots, x_d) \text{ mit } x_i = \sum_{j=0}^{s-1} \alpha_{jd+i-1} 2^j$$

Beweis

Nach Definition 3-21 ist die Z-Adresse $\alpha = \alpha_{sd-1}\,\alpha_{sd-2},\ldots.\,\alpha_0 = Z(x) = \sum_{j=0}^{s-1}\sum_{i=1}^{d} x_{i,j} \cdot 2^{j\cdot d+i-1}$

Setzen wir die Variable i auf einen Wert $\in [1,d]$, erhalten wir die Bits, die zu dem Attribut x_i gehören. Das sind also

$$\sum_{j=0}^{s-1} x_{i,j} \cdot 2^{j\cdot d+i-1} = \alpha_{(s-1)d+i-1}\alpha_{(s-2)d+i-1}\ldots\alpha_{i-1}$$

Das entspricht exakt der Formel von Satz 3-2.
q.e.d.

Die Z-Adresse ordnet also eindeutig jedem Punkt im Raum Ω einen Wert zu. Die daraus entstehende Ordnung bezeichnen wir als Z-Ordnung.

Definition 3-22: Z-Ordnung, $\preccurlyeq$

Die totale Z-Ordnung „$\preccurlyeq$" im *d*-dimensionalen Raum ist die Ordnung, die durch die Werte der Z-Adressen auf den d-dimensionalen Raum definiert werden.

Eine wesentliche Eigenschaft der Z-Ordnung ist, dass die Z-Adresse eines Tupels *x* durch Verschränkung der Bits (Bitinterleaving) der einzelnen Attribute in linearer Zeit bezüglich der Bitlänge der Adresse berechnet werden kann. Hierzu muss man lediglich die einzelnen Attributwerte vom höchstwertigen zum minderwertigsten Bit durchlaufen und die jeweils i-ten Bits zur i-ten Stufe der Adresse gruppieren. Innerhalb einer Stufe wird die Reihenfolge unter den i-ten Bits festgelegt, die auf allen Stufen beibehalten wird. Abbildung 3-7 zeigt den entsprechenden Algorithmus. Die Funktionen

$$steps(r_i) = \log_2 r_i \qquad \text{Gl: 3-20}$$

berechnen die Anzahl der Bits, die benötigt werden, um den Wertebereich der Dimension *i* des Basisraums Ω darzustellen. Um die Länge einer Stufe einer Z-Adresse zu berechnen, wird die Funktion *steplength*(Stufe) benutzt. Sie gibt somit die Anzahl der Dimensionen zurück, die zu dieser Stufe beitragen. Da wir ohne Beschränkung der Allgemeinheit davon ausgehen, dass der Wertebereich der einzelnen Dimensionen identisch ist, erhält man z.B. für den 3-dimensionalen Basisraum für alle Stufen den Wert 3.

BITINTERLEAVING

Eingabe: x : Tupel
d : Anzahl der Dimensionen
Ausgabe: α : Z-Adresse
// Die Dimensionen müssen nach der Auflösung
// absteigend sortiert sein
{
for(*step* = 0; step <= max({steps(r_j)| j ∈ [1,d]}))
{
for(i = 1; i <= steplength (step))
{
$\alpha_{step,i} = x_{i,step}$ //kopiere Bit *step* von x_i an Bitposition *i* von α_{step}
}//Ende der for Schleife
} // Ende der for Schleife
Ρετυρν α
}

Abbildung 3-7: Bitverschränkung

In Abbildung 3-8 wird das Bitinterleaving noch einmal verdeutlicht. In diesem Beispiel besteht das Tupel *x* aus drei Attributen zu je drei Bits. Das höchstwertige Bit der Attribute ist mit a.2, b.2 bzw. c.2 bezeichnet. Die Reihenfolge innerhalb einer Stufe ist a,b,c.

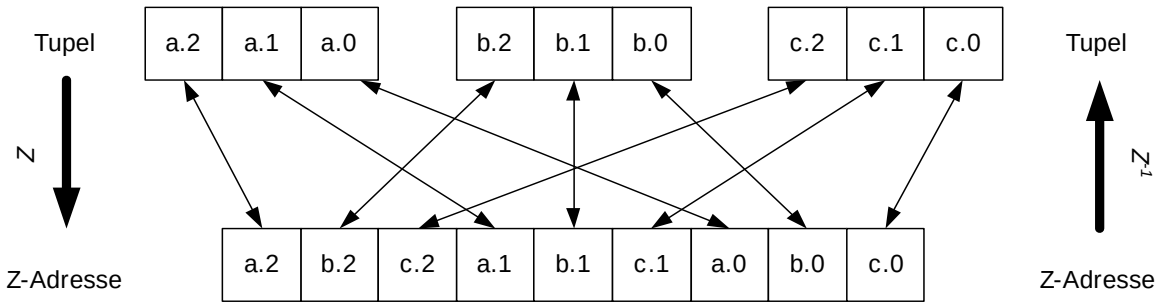


Abbildung 3-8: Bitverschränkung (Bitinterleaving)

3.6.3 Regionen

Das Kernkonzept des UB-Baums ist die Z-Region. Eine Z-Region ist ein mehrdimensionaler Teilraum, der durch ein Z-Intervall $[\alpha,\beta]_z$ abgedeckt wird und einer Seite *p* des Sekundärspeichers zugeordnet ist. Für die Elemente der Seite liegen die jeweiligen Z-Adressen im Z-Intervall $[\alpha,\beta]_z$.

Definition 3-23: Z-Region

Eine Z-Region $\rho = [\alpha:_z\beta]$ ist ein Teilraum des Basisraums Ω, der durch das Z-Intervall $[\alpha,\beta]_z$ vollständig abgedeckt ist. Die Z-Region ist genau einer Seite *p* des Sekundärspeichers zugeordnet, $\rho \leftrightarrow p$. Es gilt:

$$x \in \text{content}(p) \Rightarrow x \in \rho \quad \text{und}$$

$$x \in \text{content}(p) \Rightarrow Z(x) \in [\alpha,\beta]_z \text{ mit } \rho = [\alpha:_z\beta]$$

Da jeder Z-Region eine Sekundärseite *p* zugeordnet ist und einer Sekundärseite *g*enau eine Region ρ zugeordnet ist, handelt es sich hier um eine bijektive Abbildung von Region auf Seiten. Wenn aus dem Zusammenhang ersichtlich ist, dass wir die Z-Kurve benutzen, lassen wir das Z in $[\alpha:_z\beta]$ und $[\alpha,\beta]_z$ weg und schreiben $[\alpha:\beta]$ und $[\alpha,\beta]$. Zu beachten ist hier vor allem, dass $[\alpha:_z\beta]$ eine Menge von Punkten ist und $[\alpha,\beta]_z$ von Z-Adressen.

Das *normalisierte Z-Volumen* V_z eines Z-Intervalls ist die Differenz der Z-Adressen bezüglich des Basisraums Ω.

Definition 3-24: Normalisiertes Z-Volumen

Das *normalisierte Z-Volumen V* eines Z-Intervalls $[\alpha,\beta]$ im *d*-dimensionalen Basisraum Ω ist:

$$V_z([\alpha,\beta]] = \frac{(Z(\beta) - Z(\alpha)) + 1}{|\Omega|}$$

Das *normalisierte Volumen V* für die leere Menge $\varnothing$ ist:

$$V_z(\varnothing) = 0$$

Eine wichtige Eigenschaft der Z-Region besteht darin, dass ein zusammenhängendes Z-Intervall im kartesischen Raum in maximal zwei Bereiche zerfällt. Hierzu wird nun zunächst das Konzept der Nachbarschaft eingeführt.

Definition 3-25: ≤-Nachbar

Gegeben sei eine Menge *B*, auf der eine totale Ordnung „≤“ existiert (*B*,≤). Zwei Elemente $a, b \in B$ sind Nachbarn bezüglich der totalen Ordnung „≤“, genau dann, wenn gilt:

$$(a < b) \wedge (\neg\exists\, c \in B: a < c < b)$$

Satz 3-3: Anzahl der ≤-Nachbarn

Gegeben ist eine Menge *B*, auf der eine totale Ordnung „≤“ existiert (*B*,≤). Die Anzahl der ≤-Nachbarn(a) ist für ein Element *a* :

$$|\leq -Nachbarn(a)| = \begin{cases} 1 & , \quad a = \max(B) \\ 1 & , \quad a = \min(B) \\ 2 & , \quad sonst \end{cases}$$

Beweis:

Ist eine direkte Konsequenz aus Definition 3-25.
q.e.d.

Für den mehrdimensionalen Raum **Ω** wird der „≤-*Nachbar*“ auf den n-dimensionalen Raum wie folgt verallgemeinert:

Definition 3-26: ⊴-Nachbar

Zwei Punkte *x*, *y* sind ⊴-Nachbarn genau dann, wenn sie sich genau in einem Attribut *i* unterscheiden und es gilt x_i ≤-Nachbar y_i

Satz 3-4: Anzahl der ⊴-Nachbarn im *d*-dimensionalen Raum

Gegeben ist ein *d*-dimensionaler Basisraum **Ω**. Dann ist die Anzahl der ⊴-Nachbarn(a) für ein Tupel *x*:

$$|\trianglelefteq\text{-Nachbar(x)}| = \sum_{i=1}^{d} \leq\text{-}Nachbar(x_i)$$

Beweis:

Ist eine direkte Konsequenz aus Definition 3-26.
q.e.d.

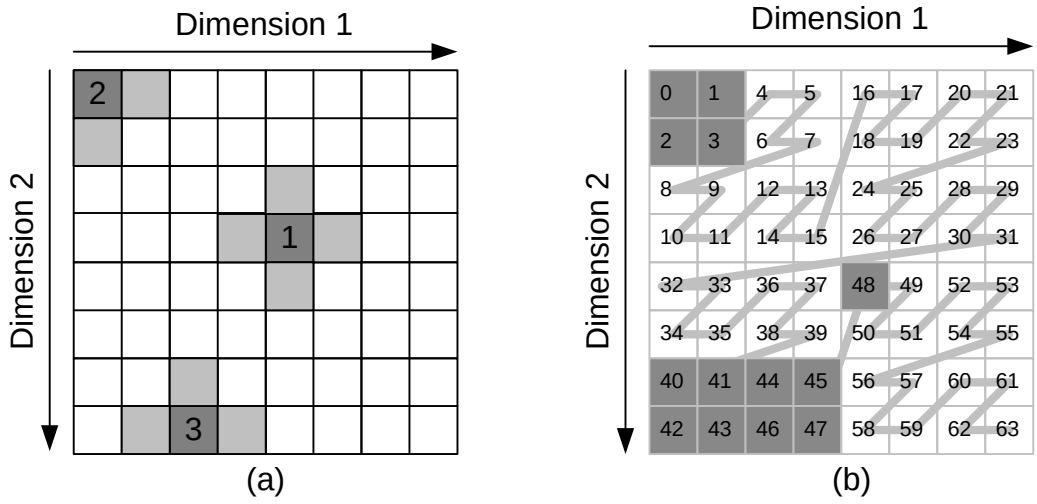


Abbildung 3-9: Nachbar im 2-dimensionalen Raum

Somit hat ein Tupel *x* im *d*-dimensionalen Basisraum **Ω** maximal *2d* und mindestens *d* Nachbarn. Abbildung 3-9 zeigt die drei möglichen Fälle im 2-dimensionlen Raum. Punkt 1 liegt mitten im Raum und hat somit 2d = 2*2 = 4 Nachbarn. Punkt 2 liegt genau in der oberen Ecke, so dass jede Dimension nur einen Nachbarn zum Ergebnis beiträgt. Punkt 3 hat genau 3 Nachbarn, da Dimension 1 nur einen Nachbar zum Ergebnis beiträgt.

Mit Hilfe der Nachbarschaftsbeziehung kann nun die Stetigkeit für diskrete raumfüllende Kurven wie folgt definiert werden:

Definition 3-27: Stetigkeit

Eine totale Ordnung „≤“ auf einem d-dimensionalen Basisraum **Ω** ist stetig, genau dann, wenn gilt:

$$\forall a, b \in \Omega : a \leq\text{-Nachbar } b \Rightarrow a \trianglelefteq\text{-Nachbar } b$$

Satz 3-5:

Die Z-Kurve ist nicht stetig.

Beweis:

Gegeben sind die Z-Adressen $\alpha = 1$ und $\beta = 2$. Nach Definition 3-25 sind α und β ≤-Nachbarn. $Z^{-1}(\alpha) = (1,0,0,...,0)$ und $Z^{-1}(\beta) = (0,1,0,0,...,0)$ sind jedoch keine ⊴-Nachbarn, so dass die Z-Kurve nicht nach Definition 3-27 stetig ist.
q.e.d.

Da die Z-Kurve nicht stetig ist, also aus der Nachbarschaft auf der Z-Kurve nicht die Nachbarschaft im Basisraum **Ω** folgt, kann eine Z-Region aus mindestens zwei nicht zusammenhängenden Teilbereichen bestehen. Hierzu wird nun das Konzept der *räumlichen Verbindung* eingeführt.

Definition 3-28: räumliche Verbindung

Gegeben sind zwei Teilräume T_1 und T_2 von **Ω**. T_1 und T_2 haben eine *räumliche Verbindung* genau dann, wenn gilt:

$$\exists\, x \in T_1, y \in T_2 : x \trianglelefteq\text{-Nachbar } y$$

Zwei Punkte *x*, *y* im diskreten Basisraum **Ω** besitzen also eine *räumliche Verbindung*, wenn sie sich in ihren Koordinaten genau in einer Dimension *i* unterscheiden und es gilt:

$$|x_i - y_i| = 1 \qquad \texttt{Gl: 3-21}$$

Grundsätzlich könnte es sein, dass eine Z-Region in *n* Teilräume zerfällt, die nicht miteinander räumlich verbunden sind. Würde also eine Z-Region in beliebig viele Teilräume zerfallen, würde das gegen das Prinzip der Lokalität verstoßen und somit zu einer nicht optimalen Leistung führen. Glücklicherweise ist die Obergrenze unabhängig von der Dimensionalität des Basisraums **Ω** und beträgt 2.

Satz 3-6: Verbindungs-Theorem [FriM97], [Mar99]

Eine Z-Region kann aus maximal zwei räumlich nicht verbundenen Teilräumen bestehen.

Beweis:

Ein ausführlicher Beweis, der auf Wiederspruch basiert, ist in [FriM97] und [Mar99] zu finden.

Z-Regionen, die aus zwei Teilbereichen bestehen, werden als Z-Sprungregionen bezeichnet (Abbildung 3-9.b).

3.6.4 Das UB-Baum-Konzept

Basierend auf der Z-Kurve und dem Regionskonzept können wir nun einen UB-Baum wie folgt beschreiben:

Der Basisraum Ω wird in eine Menge von Teilräumen T_i mit $i \in \{0,...,n-1\} = I$ partitioniert. Hierbei handelt es sich um eine Zerlegung. Eine Zerlegung basiert auf dem Konzept der *Disjunktion*. Zwei Mengen sind zueinander *disjunkt*, wenn sie kein gemeinsames Element besitzen, d.h. es gilt $A \cap B = \varnothing$. Nun können wir eine Zerlegung wie folgt definieren:

Definition 3-29: Zerlegung des Basisraums Ω

Eine Menge von Teilräumen $\mathbb{P} = \{T_0,T_1,... ,T_{n-1}\} \subseteq \wp(\Omega)$ bezeichnen wir als *Zerlegung* des Basisraums Ω, falls gilt:

- $\forall i, j \in I : i \neq j \Rightarrow T_i \cap T_j = \varnothing$
- $\bigcup_{i=0}^{n-1} T_i = \Omega$

Jeder Teilraum T_i entspricht nun genau einer Z-Region ρ_i. Die Menge von Regionen wiederum bezeichnen wir als Z-Regionszerlegung Θ_z.

Definition 3-30: Z-Regionszerlegung Θ_z

Eine Z-Regionszerlegung Θ_z des Basisraums Ω ist eine Menge von *Z*-Regionen $\{\rho_0, ..., \rho_{n-1}\}$ mit

$$\bigcup_{i=0}^{n-1} \rho_i = \Omega \text{ und } \forall_{j,i=1,...,k \text{ und } j \neq i} \rho_i \cap \rho_j = \varnothing$$

Bei einer gegebenen Z-Regionszerlegung Θ_Z wird jede Z-Region ρ_i eindeutig einer Seite p_i des Sekundärspeichers mit $\rho_i \leftrightarrow p_i$ zugeordnet.

Bei einer gegebenen Z-Regionszerlegung Θ_Z bestimmt die Z-Adresse $Z(x)$ eines Tupels x eindeutig eine Z-Region $\rho = [\alpha:\beta]$ und die entsprechende Seite p mit $\rho \leftrightarrow p$, da es nur eine

Z-Region und ein entsprechendes Z-Intervall [α,β] gibt, in dem die Z-Adresse Z(x) enthalten ist, d.h. $Z(x) \in [\alpha,\beta]$. Die Abbildung der Z-Region auf die jeweilige Sekundärspeicherseite *p* wird durch die Funktion *seite*() beschrieben.

$$p = seite(\rho) \qquad \text{Gl: 3-22}$$

Die Menge der Seiten $\{p_0, ..., p_{n-1}\}$ bezeichnen wir als Seitenmenge P_R der Relation R mit $R \subseteq \Omega$.

Definition 3-31: Seitenmenge P_R

Eine *Seitenmenge* P_R der Relation *R* ist die Menge von Seiten $\{p_0, ..., p_{n-1}\}$, für die gilt:

$$\bigcup_{i=0}^{n-1} \rho_i = \Omega \text{ und } \forall_{j,i=1,...,n-1 \text{ und } j \neq i} \rho_i \cap \rho_j = \varnothing \text{ und } \forall_{i=1,...,n-1} p_i = \text{seite}(\rho_i)$$

Somit kann eine Z-Region nur eine begrenzte Anzahl von Daten (Tupel) aufnehmen, die durch die Kapazität κ der Sekundärspeicherseite festgelegt wird. Ist $|p| < \kappa$, so ist die Seitenauslastung der Seite nicht 100 Prozent. Es können weitere Daten aufgenommen werden.

Alle Z-Regionen werden in den Blättern des B^*-Baums gespeichert und sind zusätzlich als verkettete Liste organisiert. Hierdurch ist eine sequenzielle Abarbeitung nach der Z-Ordnung auf sehr effiziente Weise möglich. Eine Z-Region [α:β] ist (Abschnitt 3.6.3) durch seine minimale und maximale Z-Adresse bestimmt. Da ein UB-Baum den vollständigen Basisraum Ω in disjunkte Regionen unterteilt, kann die Untergrenze einer Z-Region durch Addition um 1 auf die Obergrenze der Vorgängerregion berechnet werden. Dasselbe gilt für die Obergrenze und die Nachfolgerregion. Nehmen wir an, wir haben zwei aufeinander folgende Z-Regionen [α:β] und [γ:δ] bzw. zwei Seiten p_1 und p_2, dann gelten für die beiden Z-Regionen folgende Gleichungen:

$$\gamma = \beta + 1 \qquad \text{Gl: 3-23}$$

$$\beta = \gamma - 1 \qquad \text{Gl: 3-24}$$

Ohne Beschränkung der Allgemeinheit wird zur Identifikation einer Z-Region deren Obergrenze benutzt. Jeder Z-Region-Identifikator kann auch als Separator betrachtet werden, so dass er auch im Indexteil des B^*-Baums gespeichert wird.

Jedes Tupel *x* wird genau in derjenigen Region [α:β] abgelegt, die durch ihre Z-Adresse *Z*(*x*) bestimmt wird. Da die der Z-Region zugeordnete Seite nur eine bestimmte Anzahl κ von Daten (Tupel) aufnehmen kann, kann es beim Einfügen eines Tupels zu einem Überlauf kommen. Dies führt dann zu einer Spaltung der Seite (page split) in der Mitte, wobei zwei neue Seiten mit einer gleichen Anzahl von Elementen entstehen. Die alte Z-Region, die jetzt nur noch die Hälfte der Element enthält, muss nun im B^*-Baum aktualisiert (update), die neue Seite mit dem neuen Separator in den Baum eingefügt (insert) werden.

Betrachten wir an dieser Stelle das Konzept der Z-Region etwas näher. Die Z-Region ist ein räumliches Konzept, d.h. zwischen den real existierenden Daten und der daraus entstehenden Z-Region besteht ein gewisser Freiheitsgrad bezüglich der genauen Festsetzung des Separators zwischen zwei aufeinander folgenden Z-Regionen ρ_1 und ρ_2. Sei Z(x) das Tupel mit dem größten Z-Wert in ρ_1 und Z(y) das Tupel mit dem kleinsten Z-Wert in ρ_2. Nun hat man die Möglichkeit den Separator β im Bereich

$$\beta \in [Z(x), Z(y)[\qquad \text{Gl: 3-25}$$

zu wählen. Dieser Freiheitsgrad hat auch einen Einfluss auf die Leistung bei Bereichsanfragen auf UB-Bäume und ist unter dem Begriff Fransen (fringes) in [FriM97], [Mar99] genau analysiert. Ziel ist es, möglichst rechteckige Z-Regionen zu bilden, um so den rechteckigen Anfragebereich durch die Z-Region besser approximieren zu können.

Formal kann man den UB-Baum wie folgt definieren:

Definition 3-32: UB-Baum

Ein B*-Baum ist ein UB-Baum genau dann, wenn folgende Bedingungen gelten:

- Die Schlüssel der internen Knoten sind Z-Adressen von Z-Regionen, d.h. die obere oder untere Grenze von $[\alpha:_z\beta]$
- Die Schlüssel sind nach der Z-Ordnung sortiert.
- Die Blätter enthalten die Tupel der entsprechenden Z-Regionen.

Die Blätter stellen somit die Seitenmenge P_R, der Index des B*-Baums (internen Knoten) die Z-Regionszerlegung Θ_Z dar.

Das Konzept des UB-Baums ist in Abbildung 3-10 zusammengefasst. Hierbei handelt es sich um einen 2-dimensionalen UB-Baum mit einer Auflösung von 8x8 Punkten, der seinen Ursprung (0,0) in der linken oberen Ecke hat. Die einzelnen Punkte des Basisraums Ω sind bezüglich der Z-Kurve durchnummeriert mit 0 als minimalen Wert und

$$|\Omega_1 \cdot \Omega_2| - 1 = |8 \cdot 8| - 1 = 63 \qquad \text{Gl: 3-26}$$

als maximalen Wert. Im Basisraum sind 13 Tupel abgelegt, die duch die schwarzen Vierecke dargestellt sind (siehe Abbildung 4-1.b). Der Basisraum wird durch einen UB-Baum der Ordnung 1 verwaltet, d.h. eine Seite p hat die Kapazität $\kappa = 2$. Abbildung 3-10.c zeigt die Partitionierung von Ω in 8 Z-Regionen. Abbildung 3-10.d zeigt noch eimal die genauen Daten der einzelnen Regionen. Man kann einfach erkennen, dass jede Region auf ein Z-Intervall abgebildet wird und dieses Intervall mimimal ein Tupel und maximal zwei Tupel enthält.

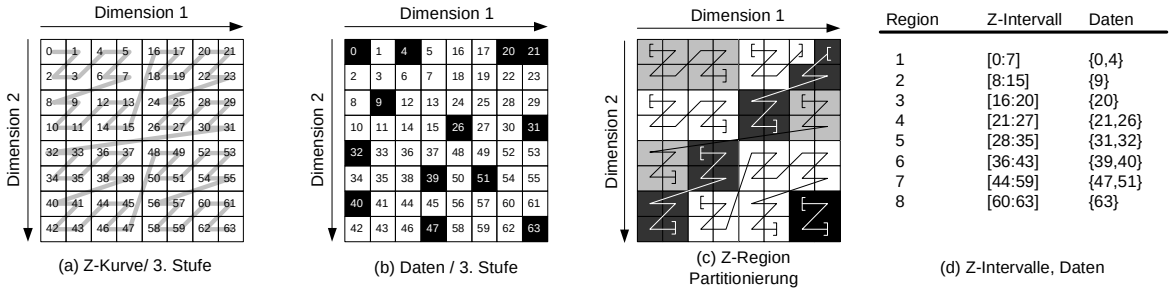


Abbildung 3-10: Partitionierung eines UB-Baums

Um z.B. das Tupel (4,1) in den UB-Baum einzufügen, muss zunächst der Z-Wert berechnet werden, also Z(4,1) = 18. Damit kann das Z-Intervall [16,20] eindeutig identifiziert werden, somit auch die Z-Region ρ_3. Da die Region nur das Tupel mit der Z-Adresse 20 enthält, entsteht kein Überlauf und der Algorithmus terminiert. Allgemein verlaufen die Einfüge- (Insert), Aktualisierungs- (Update) und Such- (Search) Operationen genauso wie beim B^*-Baum. Der Unterschied besteht in der vorgeschalteten Schlüsseltransformation, d.h. in der Abbildung auf den Z-Wert. Deshalb hat der UB-Baum auch die gleichen logarithmischen Leistungsmerkmale wie der B^*-Baum. Ein wesentlicher Vorteil gegenüber dem 1-dimensionalen B^*-Baum liegt in der effizienten Abarbeitung von Bereichsanfragen (Range-Query). Durch einen speziellen Bereichsanfragealgorithmus [Mar99],[RamMF*00] wird die Selektivität, die durch die jeweilige SQL-Anweisung in der WHERE Klausel definiert ist, vollständig ausgenutzt.

Definition 3-33: Selektivität *S*

Die *Selektivität* $S_{E(Q),R}$ einer Bereichsanfrage Q ist der Prozentsatz des Ergebnis E zur gesamten Relation R

$$S = S_{E(Q),R} = \frac{E(Q)}{R}$$

Da die Selektivitäten $S_i = S_{E([x_i,y_i]),R}$ bei unabhängigen Dimensionen multiplikativ sind, ergibt sich für die Bereichsanfrage Q folgende Formel:

$$S_{E(Q),R} = \prod_{i=0}^{d-1} S_i \qquad \text{Gl: 3-27}$$

Das heißt für den UB-Baum, dass für eine Relation R mit P Seiten und einer Bereichsanfrage Q nur circa

$$P_{\mathrm{UB}} \approx P \cdot S_{E(Q),R} = P \cdot \prod_{i=0}^{d-1} S_i \qquad \text{Gl: 3-28}$$

Seiten geladen werden müssen.

3.7 Indexstrukturen in kommerziellen Datenbanksystemen für Bereichsanfragen

Kommerzielle Datenbanksysteme wie Oracle 8i, DB2, SQL Server 200 oder TransBase unterstützen mehrdimensionale Bereichsanfragen in Form von konkatenierten Indexschlüsseln, die in B^*-Bäumen verwaltet werden. Diese können sowohl clusternd als

auch nicht clusternd erzeugt werden. Ein clusternder Index wird als *IOT* (index organized table) bezeichnet, nicht clusternde Indexe *IF* (inverted file). Im Datenteil des B^*-Baums sind nicht die Daten, sondern die TIDs gespeichert. Nach Kapitel 3.2 ist ein IOT ein Primärindex und eine IF ein Sekundärindex. Daneben verfügt Oracle über einen Bitmap-Index sowie über eine R-Baum-Implementierung.

Ein Bitmapindex ist ein Sekundärindex und deshalb nicht clusternd. Da ein Sekundärindex Duplikate bezüglich des Indexschlüssels enthält, können bei geringer Kardinalität des indizierten Attributs sehr lange Listen von TIDs entstehen, die denselben Indexschlüsselwert haben. Beispiel: Gegeben ist eine Tabelle „Angestellter“, in der das Attribut „Geschlecht“ enthalten ist; die Kardinalität der Domäne ist 2. Ein Sekundärindex hätte demzufolge zwei Blätter mit sehr langen TID-Listen des jeweiligen Geschlechts. Die Idee des Bitmapindex besteht darin, eine TID-Liste durch einen Bit-Vektor zu ersetzen. Hierbei repräsentiert jedes Bit ein Tupel in der Tabelle. Die Position des Bits bestimmt die Position des Tupels in der Tabelle. Erfüllt das Tupel das Suchkriterium, wird das entsprechende Bit auf 1 gesetzt. Wie man sieht, ist der Bitmapindex eine kompakte Darstellung von TID-Listen für Sekundärindex mit kleinen Kardialitäten. Da der Bitmapindex einem nicht clusternden B^*-Baum mit kompakter TID-Liste entspricht, wird er im Folgenden nicht mehr weiter betrachtet. TransBase unterstützt neben dem B-Baum auch den UB-Baum als clusternden Primärindex.

3.8 Klassifikation der betrachteten Zugriffsmethoden

Diese Arbeit vergleicht die entwickelten Algorithmen, die den UB-Baum als mehrdimensionale Zugriffsmethode verwenden, mit den in kommerziellen DBVS verfügbaren Zugriffsmethoden. Hierbei wird *IOT, IF* und das sequenzielle Lesen *FTS* (full table scan) betrachtet. Basierend auf der Klassifikation aus Kapitel 3.3 können die einzelnen Indextypen nach der Anzahl der Restriktionen, die von der Indexstruktur parallel unterstützt werden, sowie die unterstützte Clusterung wie folgt klassifiziert werden:

		Clusterung		
		Keine	*Tupel*	*Seiten*
Restriktion	*Punk*	IF	IOT,UB	IOT,UB
	1-dim	IF	IOT,UB	IOT,UB
	n-dim	IF ∩ IF ...	UB	UB
	keine			FTS

Tabelle 3-1 Klassifikation der Indextype

3.9 Zusammenfassung

In diesem Kapitel wurden die grundlegenden Definitionen und Notationen dieser Arbeit eingeführt. Sie bilden somit die Basis für die weiteren Kapitel. Tabelle 3-2 fasst die eingeführten Bezeichner zusammen.

Symbol	Bezeichnung
R	Relation
D	Wertebereiche (Domänen)
x	Tupel, Punkt
sch(R)	Menge der Attribute $\{A_1,A_2,\ldots,A_n\}$ der Relation R
sch(x_i)	das Attribut A_i mit $x \in R$
dom(A)	Domäne des Attributs A
$\boldsymbol{K}$	logischer Schlüssel
$\times$	kartesisches Produkt
π	Projektion
σ	Selektion
$\triangleright\triangleleft$	Verbund
$\cup$	Vereinigung
$-$	*Mengendifferenz*
φ_X	beliebiges Prädikat über die Attributmenge X mit $X \in$ **sch**(R)
M	Hauptspeicher in Seiten
TS	Tertiärspeicher in Seiten
P	Anzahl der Seiten
p	Seite, Identifikator
Content (p)	Inhalt einer Seite
$p_{größe}$	Größe einer Seite in Byte
$x_{größe}$	Größe eines Tupels in Byte
κ	Kapazität
t_λ	Durchschnittliche Umdrehungswartezeit
t_τ	Durchschnittliche Block-Übertragungszeit
t_π	Durchschnittliche Positionierungszeit
t_δ	Durchschnittliche Zugriffsbewegungszeit
$t_{E/A\text{-}ran}$	*wahlfreie Zugriffszeit*
C	Prefetching, E/A-Einheit
$c_{E/A}$	Kosten für eine E/A-Operation
TID	*Tupel-Identifikatoren*
c_{wf}	Kosten für den wahlfreien Zugriff
c_{TC}	Kosten für den Zugriff bei Tupel-Clustering
c_{SC}	Kosten für den Zugriff bei Seiten-Clustering
IF	*Inverted File, Sekundärindex*
T	Baumstrucktur, z.B. B-Baum
I	Anzahl der Schlüssel
$c_{E/A\text{-}lesen}$	E/A-Kosten für das Lesen von Daten
$c_{E/A\text{-}schreiben}$	E/A-Kosten für das Schreiben von Daten
$c_{E/A\text{-}löschen}$	E/A-Kosten für das Löschen von Daten
H	Hashfunktion
$\mathbb{N}$	Natürlichen Zahlen
$>$	Vergleichoperation „größer als“ auf $\mathbb{N}$
$<$	Vergleichoperation „ kleiner als“ auf $\mathbb{N}$
$\lvert D\rvert$	Kardinalität der Domäne

S	$S \subset \mathbb{N}$
U	Universum
$\preccurlyeq$	Z-Ordnung
d	Anzahl der Dimensionen
s	Anzahl der Stufen
Ω	Basisraum, kartesisches Produkt der einzelnen Domänen Ω_i
Ω_i	Teilmenge von $\mathbb{N}$
$\|\Omega\|$	Kardinalität des Basisraumes
V	Normalisiertes Volumen
r_i	Anzahl der Elemente in Ω_i
α	Adresse
ρ	Region, Teilraum von Ω

Tabelle 3-2: Symoble

You can play a tune of sorts on the white keys, and you can play a tune of sorts on the black keys, but for harmony you must use both the black and the white.

***James Emman Kwegyir Aggrey** (1875–1927)*

4 Bereichsanfragen in Kombination mit Sortierung

4.1 Das Problem

Wie bereits in Abschnitt 1.3 beschrieben, besteht ein Ziel dieser Arbeit darin, Verfahren zur effizienten Abarbeitung der folgenden SQL-Anweisung zu entwickeln und zu analysieren:

```
SELECT    A1, A2, A3, …
FROM      Fakten_Tabelle
WHERE     A1 BETWEEN r1 AND r2,
          A2 BETWEEN r3 AND r4,
          A3 BETWEEN r5 AND r6,
       …
ORDER BY Ai,…
```

Abbildung 4-1: Bereichsanfrage mit Sortierung

Die Faktentabelle ist als UB-Baum organisiert. Ohne Beschränkung der Allgemeinheit wird davon ausgegangen, dass auf einer Dimension jeweils nur ein zusammenhängendes Intervall als Einschränkung existiert. Der Anfragebereich Q kann also wie folgt definiert werden:

Definition 4-1: Anfragebereich

Der Anfragebereich Q ist ein orthogonales mehrdimensionales Intervall $[[ql, qh]]$ (Hyperwürfel) zwischen den zwei Punkten $l = (ql_1\ q,l_2\ ,..., ql_d)$ und $qh = (qh_1\ ,qh_2\ ,..., qh_d)$, welches die einzelnen Attribute A_i auf ein Intervall $[ql_i,qh_i]$ mit $ql_i \leq qh_i$ einschränkt:

$$Q = [[ql, qh]] = [ql_1, qh_1] \times ... \times [ql_j, qh_j] \times ... \times [ql_d, qh_d]$$

4.2 Herkömmliche Abarbeitung

In diesem Abschnitt werden zunächst zwei herkömmliche Methoden zur Abarbeitung der in Abbildung 1-12 dargestellten SQL-Anfrage betrachtet. Der Ausführungsplan kann in die folgenden beiden Teile gegliedert werden:

- Abarbeitung der Bereichsanfragen
- Sortierung der Tupel bezüglich des Sortierungsattributs A_i

Der erste Teil des Anfrageplans ist die Bereichsanfrage. Der Optimierer entscheidet, welcher Zugriffspfad herangezogen wird oder ob alle Seiten sequenziell (Scan) vom Sekundärspeicher gelesen werden, um die Suchbedingung zu testen.

Betrachten wir zunächst den konkatenierten Index. Der Schlüssel eines konkatenierten Index besteht aus der Konkatenation von Attributen, z. B. $A_1 \circ A_2$. Sie bestimmen die physikalische Ablage, also die physikalische Lokalität (siehe Abschnitt 0). Man erhält für einen 2-dimensionalen Basisraum Ω eine streifenförmige Raumaufteilung. Abbildung 4-2.a zeigt eine Aufteilung bezüglich eines konkatenierten Index $A_1 \circ A_2$, der aus 16 Seiten besteht. Deutlich erkennt man die Priorisierung von Attribut A_1. Um nun alle Tupel vom Sekundärspeicher zu laden, welche die Suchbedingung (hell umrandetes Rechteck in Abbildung 4-2.b) erfüllen, kann nur die Restriktion in Attribut A_1 ausgenutzt werden. Somit muss der gesamte schraffierte Bereich geladen werden (8 Seiten). Formal kann dieser Sachverhalt wie folgt ausgedrückt werden:

$$P_{kon} \approx \frac{|qh_i - ql_i|}{r_i} P = S_i \cdot P \qquad \text{Gl: 4-1}$$

Hierbei bezeichnet r_i die Kardinalität des Attributs A_i (siehe Gl: 3-16), S_i die Selektivität in Attribut A_i (siehe Definition 3-33) und P die Größe der Relation R in Seiten.

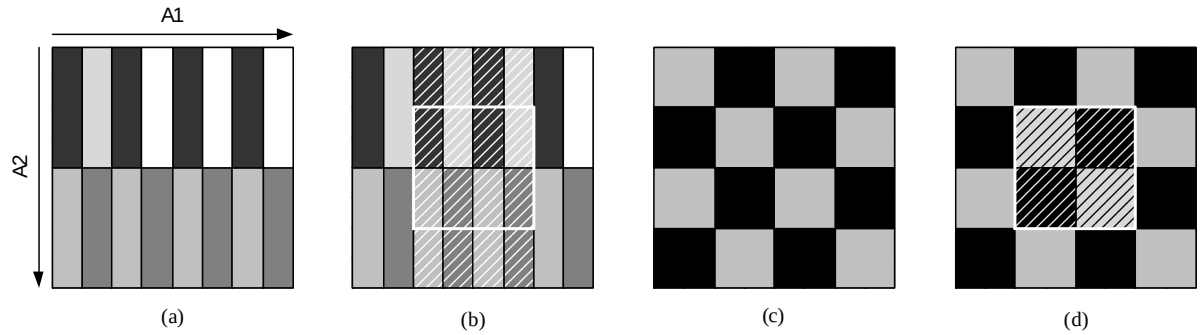


Abbildung 4-2: Zerlegung des Datenraums bezüglich der K-Kurve und Z-Kurve

Abbildung 4-2c zeigt die Zerlegung des Basisraumes bezüglich der Z-Kurve für gleichverteilte Daten, die durch den UB-Baum erzeugt wird. Die mehrdimensionale Datenstruktur UB-Baum kann durch die Verwendung des verzahnten Schlüssels (Z-Adresse, Definition 3-21) die Selektivität S (siehe Gl: 3-28) auf beiden bzw. allen Dimensionen ausnutzen. Um den Anfragebereich Q (siehe Abbildung 4-2d) zu laden, werden nur 4 Seiten benötigt. Allgemein gilt für ein großes P :

$$P_{ub} \approx S_{E(Q),R} \cdot P \qquad \text{Gl: 4-2}$$

$S_{E(Q),R}$ bezeichnet die Sekeltivität des Anfragebereichs Q auf die Relation R (siehe Definition 3-33 und Gl: 3-28). Zu beachten ist hier besonders, dass die Selektivitäten der einzelnen Dimensionen zueinander multiplikativ sind, da die Dimensionen nach Voraussetzung unabhängig voneinander sind.

Sortieren

Betrachtet man nun den zweiten Schritt in der Anfrageverarbeitung, das *Sortieren*. *Externes Sortieren* durch *Verschmelzen* (Mergesort) ist ein de facto Standard, um große Datenmengen zu verarbeiten [LarG98]. Es kommt dann zum Einsatz, wenn der verfügbare Arbeitsspeicher mit Größe *M* nicht alle Datensätze *R*, die zur Speicherung *P* Seiten benötigen, auf einmal aufnehmen kann, d.h. $M < P$ [Sal89]. In diesem Fall muss die Datenmenge in geeignete Partitionen zerlegt werden, die dann einzeln in den Arbeitsspeicher gelesen, mit einem internen Sortierverfahren, wie z.B. Quicksort oder Heapsort, sortiert und dann als so genannte Läufe (Runs, sortierte Teilfolgen) wieder auf den Sekundärspeicher zurückgeschrieben werden. Um die Initialläufe zu erzeugen, muss jede Seite einmal gelesen und geschrieben werden, so dass

$$P_{init} = P_{lesen} + P_{schreiben} = 2P \qquad \text{Gl: 4-3}$$

Seitenzugriffe entstehen. Danach folgt die Mischphase (Verschmelzungsphase), in der *m* Läufe parallel gelesen und als ein Lauf zurückgeschrieben werden. Hat die Initialisierungsphase *w* Initialläufe erzeugt, ist die Verschmelzungsphase (Verschmelzungsebenen) nach

$$v = \lceil \log_m w \rceil \qquad \text{Gl: 4-4}$$

beendet und die Satzmenge besteht aus einem sortierten Lauf [Gra93]. Die Anzahl der Mischphasen ist also abhängig von der Anzahl der Initialläufe und somit direkt abhängig von der Größe *M* des Arbeitsspeichers.

Initialläufe

Betrachten wir zunächst die Erzeugung der Initialläufe. Die Größe des Arbeitsspeichers bestimmt für viele interne Sortierverfahren (in-situ-Verfahren) die Größe eines Laufes. Da ein Datenbanksystem typischerweise mit variablen Satzlängen arbeitet, wird für das Sortieren der Datensätze eine Zwischenschicht eingeführt. Hierbei wird eine Liste von Zeigern auf die zu sortierende Liste verwaltet, die dann mit Hilfe herkömmlicher interner Sortierverfahren verarbeitet wird [Här77].

Wird dieses einfache Verfahren benutzt, erhält man bei einem Arbeitsspeicher der Größe *M* und einer Relation *R* der Größe *P*, wobei *P* und *M* die Anzahl der Seiten ist

$$w_{simple} = \left\lceil \frac{P}{M} \right\rceil \qquad \text{Gl: 4-5}$$

Läufe. Ein wesentlich besseres Verfahren ist „Ersetzen und Auswahl“ (Replacement Selection) [Knu98v3], das die Vorsortiertheit einer Satzmenge ausnutzt, um die Initialläufe im Mittel auf die Größe von 2 *M* zu steigern. Das Verfahren füllt zunächst den Arbeitsspeicher mit Daten, die durch einen „Prioritäts-Heap“ verwaltet werden, d.h. es

kann das kleinste Element mit O(1) Zeit entfernt werden. Jedoch muss danach die Heap-Eigenschaft wieder erzeugt werden. Dies schlägt mit O(log *n*) zu Buche. Das Einfügen eines neuen Datensatzes benötigt somit O(log *n*).

Ist der Arbeitsspeicher gefüllt, wird das kleinste Element aus dem Heap entfernt und in den aktuellen Lauf geschrieben. Der frei gewordene Platz wird durch einen neuen Datensatz aufgefüllt. Falls der Datensatz größer als das zuletzt geschriebene Element ist, gehört er noch zu dem aktuellen Lauf und wird deshalb in den Heap eingefügt. Der aktuelle Lauf ist dann beendet, wenn alle Elemente im Arbeitsspeicher kleiner als das zuletzt geschriebene Element sind. Da die Wahrscheinlichkeit relativ groß ist, dass das neue Element größer ist als der zuletzt in den aktuellen Lauf geschriebene Datensatz, können so Läufe entstehen, die wesentlich größer als der Arbeitsspeicher sind. Für gleichverteilte Daten ergeben sich nach [Gra93] und [Knu98v3] bei diesem Verfahren ca.

$$w_{replace} = \left\lceil \frac{P}{2 \cdot M} \right\rceil + 1 \qquad \text{Gl: 4-6}$$

Läufe. Außerdem wird eine Vorsortierung von Teilmengen der Datensätze vorteilhaft ausgenutzt, was wiederum zu einer Reduzierung der Anzahl der Läufe *w* führt. Für eine detaillierte Diskussion dieser Problematik, wird auf [Knu98v3] verwiesen.

Das Hauptproblem dieses Ansatzes ist jedoch die Speicherverwaltung. Falls die Datensätze im Puffer bleiben sollen, um zusätzliche Kopiervorgänge zu vermeiden, kann eine Seite erst durch eine neue Seite ersetzt werden, wenn der letzte Datensatz verbraucht worden ist. Da im Mittel die Hälfte der Seiten im Heap bleiben, wird die Leistungssteigerung durch die „Ersetzen-und-Auswahl-Technik“ aufgehoben. Das Problem wird noch durch Datensätze von variabler Länge gesteigert, die im DB-Bereich üblich sind. In [LarG98] wird jedoch eine Speicherverwaltung vorgestellt, mit der im Durchschnitt Lauflängen erzeugt werden, die 1,8 mal größer sind gegenüber dem einfachen Verfahren. Somit ergeben sich

$$w_{init} = \left\lceil \frac{P}{1{,}8 \cdot M} \right\rceil \qquad \text{Gl: 4-7}$$

Läufe bei der Initialisierungsphase.

Mischphase

Es wird nun die Mischphase des externen Sortierens durch Verschmelzen betrachtet. Die Initialläufe bezeichnet man als Ebene 0. Die Läufe der Ebene 0 werden zu Läufen der Ebene 1 verschmolzen, die wiederum in Läufe der Ebene 2 eingehen. Dieser rekursive Vorgang wird solange fortgesetzt, bis nur noch ein Lauf existiert und damit die Abbruchbedingung erfüllt ist. Während des Verschmelzvorganges wird jedem Eingabestrom und dem Ausgabestrom ein E/A-Puffer zugewiesen. Die Größe ist vom jeweiligen *Prefetchfaktor C* (siehe Definition 3-10) abhängig, d.h. der E/A-Puffer muss die

Seiten, die pro E/A-Operation vom Sekundärspeicher geholt werden, aufnehmen können. Die Größe ist somit *C* Seiten.

Der Mischgrad *m* ist die Anzahl der E/A-Puffer im Arbeitsspeicher, die zum Verschmelzen der Läufe benutzt werden. Der maximale Mischgrad *m* ergibt sich somit aus dem Quotient der Größe des Arbeitsspeichers *M* und dem Prefetchfaktor *C*, wobei jedoch noch ein E/A-Puffer für den Ausgabestrom berücksichtigt werden muss. Somit erhält man:

$$m = \left\lfloor \frac{M}{C} - 1 \right\rfloor = \left\lfloor \frac{M}{C} \right\rfloor - 1 \qquad \text{Gl: 4-8}$$

Da sich die Größe der Läufe pro Ebene um Faktor *m* vergrößert, ist die Anzahl der Verschmelzungsphasen, d.h. die Anzahl, die ein Datensatz in einen Lauf geschrieben wird, bis nur ein sortierter Lauf existiert, logarithmisch bezüglich der Eingabedaten und beträgt

$$v = \lceil \log_m w \rceil \qquad \text{Gl: 4-9}$$

In [Sal89] wird jedoch gezeigt, dass der E/A-Durchsatz erheblich beschleunigt werden kann, wenn beim Lesen der Daten ein „read-ahead“ und „write-behind“ eingesetzt wird. Dies hat jedoch zur Folge, dass pro Datenstrom zwei E/A-Puffer (Wechselpuffer) benötigt werden und der Mischgrad *m* ca. um die Hälfte reduziert wird.

$$m = \left\lfloor \frac{M}{2C} - 2 \right\rfloor \qquad \text{Gl: 4-10}$$

Dieser Nachteil kann jedoch durch Vorhersagetechniken, wie z.B. „forecasting“, teilweise aufgehoben werden [Gra93]. Die Grundidee besteht darin, aus den einzelnen E/A-Puffern das jeweils größte Element zu bestimmen, um dadurch den Lauf zu bestimmen, der als nächstes gelesen werden muss. Um eine gute E/A-Komplexität zu bekommen sollten pro Festplatte mindestens zwei E/A-Puffer bereitgestellt werden. Diese Methode führt zu einem Mischgrad von

$$m = \left\lfloor \frac{M}{C} \right\rfloor - 3 \qquad \text{Gl: 4-11}$$

, hierbei wird ein E/A-Puffer für den „*forecast*“ und zwei für die Ausgabe verwendet. Für den Rest der Arbeit setzen wir externes Sortieren und Sortieren durch Vermischen gleich.

4.3 Sortiertes Lesen

Die Verfahren, die im vorherigen Abschnitt beschriebenen wurden, verwenden Sortieren durch Verschmelzen, um die Zielordnung zu erzeugen. Kann das Zwischenergebnis der Bereichsanfrage nicht im Arbeitsspeicher gehalten werden, muss ein externes Sortierverfahren eingesetzt werden. Somit werden für eine Relation mit *P* Seiten bei einem *m*-Weg-Verschmelzen und einem Arbeitsspeicher der Größe *M* insgesamt

$$P_{ext\text{-}sort} = \begin{cases} \underbrace{2P}_{\text{Initiallauf}} + \underbrace{2P \log_m \frac{P}{M}}_{\text{Mischphase}} = 2P\left(1 + \left\lceil \log_m \frac{P}{M} \right\rceil\right) & , \log_m \frac{P}{M} > 1 \\ \underbrace{2P}_{\text{Initiallauf}} + \underbrace{2P}_{\text{Mischphase}} = 4P & , \text{sonst} \end{cases} \qquad \text{Gl: 4-12}$$

E/A Operationen benötigt. Da externes Sortieren keine Vorsortierung ausnutzt, stellt das Sortieren eine blockende Operation dar, d.h. es kann erst das erste Teilergebnis zurückgegeben werden, wenn die Ergebnismenge vollständig sortiert ist. Das „*sortierte Lesen*“ ist ein Algorithmus, der Bereichsanfragen und Sortieren kombiniert und so zusätzliche E/A-Operationen für das Sortieren vermeidet und durch eine geschickte Partitionierung der Ergebnismenge bereits kontinuierlich Teilergebnisse liefert.

Die Grundidee des *sortierten Lesens* besteht darin, die Zerlegung des Raumes, die der UB-Baum erzeugt hat, auszunutzen, um die Daten sortiert nach einem beliebigen Schlüsselattribut zu lesen. Hierzu wird mit einer Art *Sweepline Algorithmus* gearbeitet. Die Linie bewegt sich in Sortierrichtung über den Basisraum und liest jede Region, die geschnitten wird und innerhalb des Anfragebereichs liegt. Damit wird jede Region, die zum Ergebnis beiträgt, geladen. Alle Tupel, die kleiner als die aktuelle Sweepline sind, können bereits sortiert in die Ergebnisfolge eingefügt werden. Den Bereich, der kleiner als die aktuelle Sweepline ist, bezeichnet man als *statischen Bereich*. Entsprechend wird der andere Bereich als *dynamischer Bereich* bezeichnet. Durch die Bewegung der Sweepline sind alle Regionen im statischen Bereich mindestens einmal geschnitten worden, so dass keine Region in diesem Bereich existieren kann, die nicht bereits vom Sekundärspeicher geladen wurde.

Durch das Bewegen der Sweepline [4] in Sortierrichtung wird ein neuer Bereich der Bereichsanfrage Q statisch. Dieser Bereich wird als *Scheibe* bezeichnet und kann unabhängig vom alten statischen Bereich sortiert werden. Durch Anhängen der neuen sortierten Scheibe an den alten statischen Bereich entsteht iterativ das sortierte Ergebnis.

[4] im allgemeinen *n*-dimensionalen Fall handelt es sich um eine Sweep-Hyperebene. Hier wird zunächst die Terminologie „Sweepline“ verwendet.

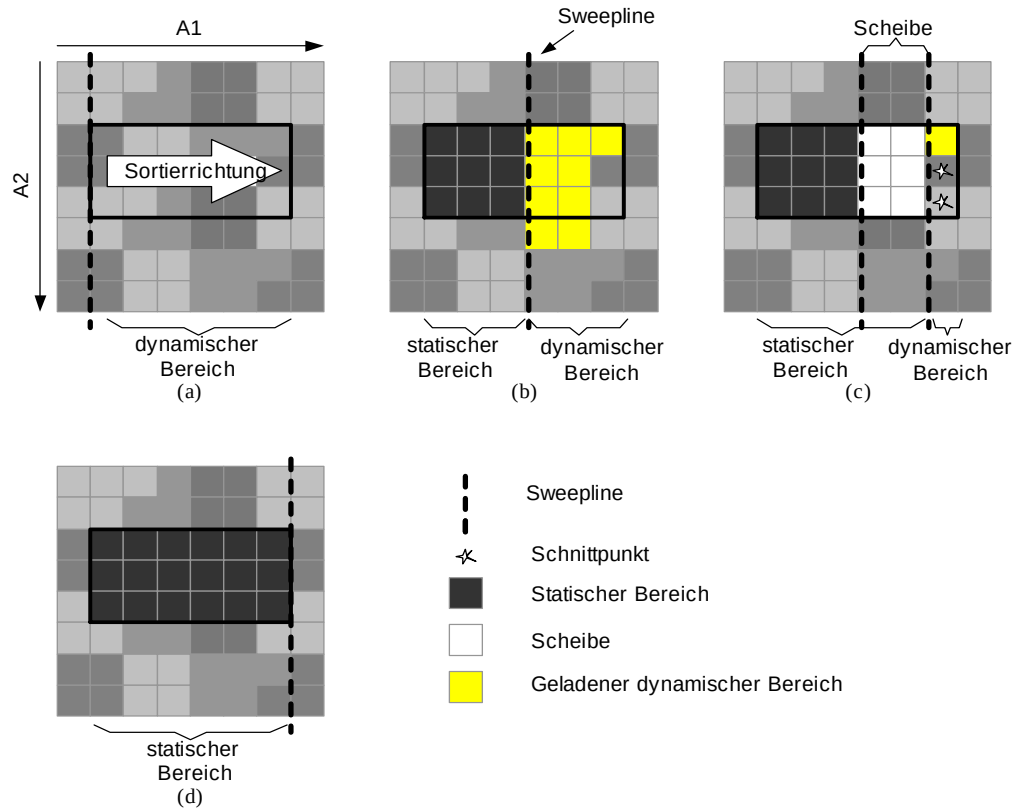


Abbildung 4-3:Grundidee des Sortierten Lesens

In Abbildung 4-3 ist die Grundidee des *sortierten Lesens* in Form eines 2-dimensionalen UB-Baums mit einer Auflösung des Universums von 8x8 Punkten und dem Ursprung (0,0) links oben in der Ecke dargestellt. Die Bereichsanfrage, das schwarzumrandete Rechteck, wird nach Attribut A_1 sortiert gelesen. Zunächst steht die Sweepline ganz links (siehe Abbildung 4-3.a), so dass der dynamische Bereich das ganze Rechteck abdeckt. Der Grund dafür, dass die Sweepline ganz links vom Anfragebereich steht, liegt in der Sortierung. Dieser Punkt repräsentiert das Minimum bezüglich der „*Sortier-Dimension*“. In Abbildung 4-3.b befindet sich die Sweepline bereits auf Position 4 bezüglich Attribut A_1. Der statische Bereich besteht aus dem linken Teil des Anfragebereichs bezüglich der *Sweepline* und der dynamische aus dem Rest. Hierbei sind bereits zwei Regionen des dynamischen Bereichs in den Arbeitsspeicher geladen worden.

In Abbildung 4-3.c wird die Sweepline nun soweit verschoben, bis der statische Bereich wenigstens eine neue Region schneidet (Regionen, die mit einem Stern markiert sind). Der Bereich zwischen der alten und der neuen Sweepline ist die aktuelle Scheibe, die, nachdem sie bezüglich Attribut A_1 sortiert worden ist, dem statischen Bereich hinzugefügt wird. Diese Schritte werden so lange wiederholt, bis die Sweepline das Ende des Anfragebereichs erreicht hat (siehe Abbildung 4-3.d).

4.3.1 Formales Model

Dieser Abschnitt beschreibt zunächst das formale Modell für die Partitionierung von Relationen, um die allgemeine Methode des *„sortierten Lesens“* unter Berücksichtigung von Restriktionen auf den Dimensionen zu beschreiben.

Gegeben ist eine *d*-dimensionale oder *d*-stellige Relation *R* mit dem Relationsschema **A** mit den Attributen **sch(*R*)** = { A_1,..., A_d} und den Domänen Ω_1,..., Ω_d, auf denen jeweils die totale Ordnungsrelation (siehe Definition 1-2) „≤“ definiert ist. Das kartesische Kreuzprodukt der Wertebereiche bildet den Basisraum (siehe Definition 3-17) Ω der Relation *R*. Die Datensätze der Relation haben dementsprechend die Form $x = (x_1,..., x_d)$. Die Relation *R* besteht aus der Seitenmenge

$$P_R = \{p_0,p_1, \ldots, p_{n-1}\} \qquad \text{Gl: 4-13}$$

(siehe Definition 3-31) und der daraus resultierenden Regionszerlegung Θ_z (siehe Definition 3-30) des Basisraums Ω.

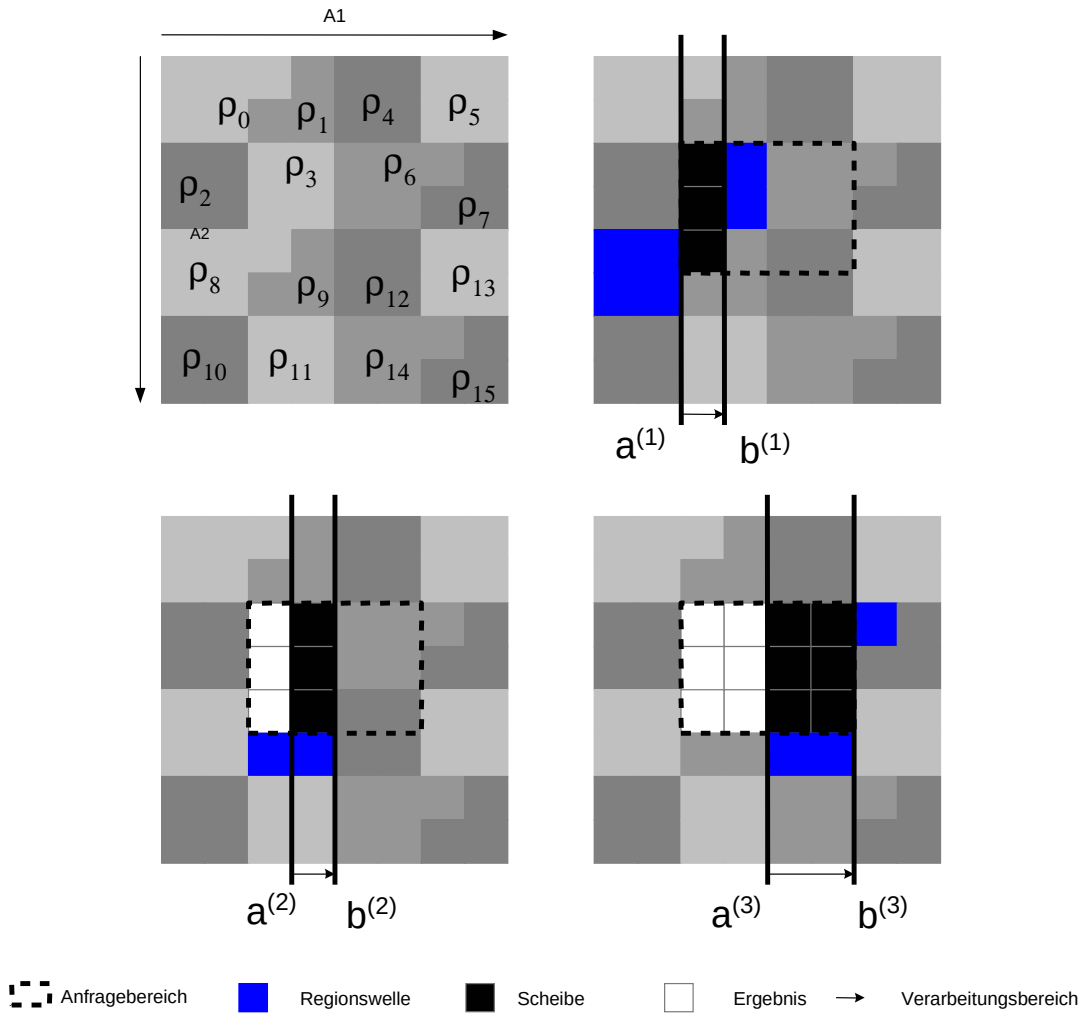


Abbildung 4-4: Regionszerlegung, Verarbeitungsbereich, Regionswelle, Scheibe und Anfragebereich

Da in dieser Arbeit nicht nur der 2-dimensionale Fall betrachtet wird, wird der Begriff der *Sweep-Hyperebene* anstelle von *Sweepline* verwendet. Die *Sweep-Hyperebene* $\Psi_{k,c}$ ist eine Hyperebene. Sie ist ein Teilraum von Ω, die in der Sortierdimension k den konstanten Wert c hat.

Definition 4-2: *Sweep-Hyperebene* $\Psi_{k,c}$

Eine *Sweep-Hyperebene* $\Psi_{k,c}$ ist eine Teilmenge von Ω mit der Eigenschaft:

$$\Psi_{k,c} := \{x \mid x \in \Omega \wedge x_k = c\}$$

Der *Verarbeitungsbereich* auf Dimension A_k ist ein Intervall $[a, b]$. Eine Schicht S^{-r} ist ein Teilraum von Ω, die in der Sortierrichtung durch den aktuellen Verarbeitungsbereich beschränkt ist. Falls die Schicht ein Teilraum innerhalb eines Anfragebereiches $Q = [[ql,qh]]$ ist, werden die übrigen Dimensionen durch Q beschränkt. Eine Schicht ist also wie folgt definiert:

Definition 4-3: Schicht S^{-r} in Basisraum Ω

Eine Schicht $S^{-r}_{[a,b],k,Q}(\Omega)$ im Basisraum Ω ist ein Teilbereich der Bereichsanfrage $Q = [[ql,qh]]$, wobei die Dimension k durch das Intervall $[a,b]$ bestimmt wird.

$$S^{-r}_{[a,b],k,[[ql,qh]]}(\Omega) = \{x \mid (x \in \Omega) \wedge (x \in [[ql,qh]]) \wedge (x_k \in [a,b])\}$$

Die Teilmenge einer Regionszerlegung Θ, die von einer Schicht $S^{-r}_{[a,b],k,[[ql,qh]]}(R)$ geschnitten wird, bezeichnen wir als Regionswelle W^r:

$$W^r_{[a,b],k,Q}(\Theta) = \{\rho \in \Theta \mid \rho \cap S^{-r} = \varnothing\} \qquad \text{Gl: 4-14}$$

Eine Seitenwelle W ist eine Menge von Seiten, die zu den entsprechenden Regionen der Regionswelle $W^r_{[a,b],k,Q}(\Theta)$ gehören und wie folgt definiert ist:

$$W_{[a,b],k,Q}(P_R) = \{p \in P_R \mid p \leftrightarrow \rho \text{ und } \rho \in W^r_{[a,b],k,Q}(\Theta)\} \qquad \text{Gl: 4-15}$$

Eine Scheibe S^{-t} ist eine Menge von Tupeln aus R, die innerhalb einer Schicht S^{-r} liegen.

Definition 4-4: Scheibe S^{-t} in Relation R

Gegeben ist eine Relation R, ein Verarbeitungsbereich $[a,b]$ auf der Dimension k und ein Anfragebereich $Q = [[ql,qh]]$. Eine Scheibe ist dann:

$$S^{-t}_{[a,b],j,[[ql,qh]]}(R) = \{x \mid (x \in R) \wedge (x \in [[ql,qh]]) \wedge (x_j \in [a,b])\}$$

Während Basisraum, Region, Regionszerlegung, Regionswelle, Anfragebereich und Schicht Teilmengen des d-dimensionalen Basisraums Ω sind, stellen Relation,

Seitenmenge, Seitenwelle, Seite, Ergebnismenge und Scheibe Mengen von Tupeln dar. Die Abbildung des Raums auf Tupelmengen ist durch die Funktion

$$\text{seite}(\rho) = p \qquad \text{Gl: 4-16}$$

und Tupelmenge auf den Raum durch die inverse Funktion

$$\text{region}(p) = \rho \qquad \text{Gl: 4-17}$$

definiert, da für jede *Region* $\rho \leftrightarrow p$ gilt (siehe Definition 3-23). Der Zusammenhang der einzelnen Teilräume und Teilmengen ist in Tabelle 4-1 zusammengefasst.

Raum		Relation von Tupeln	
Ω	Basisraum von Punkten	R	Relation von Tupeln
Θ	Regionszerlegung	P_R	Seitenmenge
W^r	Regionswelle	W	Seitenwelle
Q	Anfragebereich	E	Ergebnismenge
S^{-r}	Schicht	S^{-t}	Scheibe, Tupelmenge
ρ	Region	p	Seite
x	Punkt	x	Tupel

Tabelle 4-1: Raum-Tupel-Modell

x bezeichnet sowohl einen Punkt im Raummodell als auch ein Tupel im Relationen-Modell. Dies kommt daher, dass in dieser Arbeit ein Tupel als Punkt im mehrdimensionalen Raum angesehen wird (siehe Kapitel 3).

Da es sich beim *„sortierten Lesen“* um einen iterativen Algorithmus handelt, wird die generische Funktion *iter*( ) eingeführt, die für eine gegebene Variable *b* den Wert nach dem *i*-ten Iterationsschritt zurückgibt:

$$iter(a,i)=b \qquad \text{Gl: 4-18}$$

b bezeichnet somit den Wert der Variable *a* nach dem *i*-ten Iterationsschritt.

4.3.2 Algorithmus

Basierend auf dem formalen Modell kann nun das *„sortierte Lesen“* wie folgt beschrieben werden:

Gegeben ist eine Relation *R* und die entsprechende Zerlegung in die Seitenmenge P_R. Die Ergebnismenge *E*, die durch den Anfragebereich Q = [[*ql*, *qh*]] definiert ist, wird bezüglich Attribut A_k sortiert ausgegeben. Hierzu wird für jeden Iterationsschritt *i* eine neue Seitenwelle $W_{[iter(a,i),iter(b,i)],k,Q}(P_R)$ aus dem UB-Baum in den Arbeitsspeicher geladen, wobei

$$iter(b,i) < iter(a,i+1) \qquad \text{Gl: 4-19}$$

gilt. Der erste *Verarbeitungsbereich* [*iter*(*a*,0), *iter*(*b*,0)] wird durch die untere Grenze ql_k des Anfragebereichs Q=[[*ql*,*qh*]] bestimmt, also $a = ql_k$. Nachdem die *i*-te Seitenwelle $W_{[iter(a,i),iter(b,i)],k,Q}(P_R)$ in den Arbeitsspeicher geladen worden ist, werden die Tupel, die in der Seitenwelle enthalten sind, in den Cache eingefügt. Der Cache ist als Heap organisiert. Schlüssel des Heaps ist die Sortierdimension *k*. Alle Tupel, die zur aktuellen Scheibe $S^{-t}_{[iter(a,i),iter(b,i)],k,Q}(R)$ gehören, werden aus dem Cache entfernt und an den Verbraucher weitergereicht. Hierbei wird iterativ immer das Wurzelelement aus dem Heap entfernt. Dies führt zu einer sortierten Folge bezüglich der Sortierdimension *k*. Die obere Grenze des Verarbeitungsbereichs *iter*(*b*,*i*) bezeichnet die aktuelle Position der Sweepline. Die untere Grenze des nächsten Verarbeitungsbereichs ergibt sich wie folgt:

$$iter(a, i+1) = iter(b,i) + 1 \qquad \text{Gl: 4-20}$$

Die obere Grenze *iter*(*b*, i+1) des nächsten Verarbeitungsbereichs wird durch einen speziellen Algorithmus, der im nächsten Abschnitt beschrieben wird, bestimmt. Abbildung 4-4 zeigt die drei Iterationsschritte 1, 2 und 3, die den umrandeten Anfragebereich *Q* sortiert nach Attribut 1 abarbeiten.

Abbildung 4-5 zeigt die Kernroutine des sortierten Lesens. Hierbei werden folgende Variablen und Mengebezeichnungen benutzt:

Symbol	Beschreibung	Abstrakter Datentyp
`Q`	Anfragebereich: Teilraum vom Basisraum Ω	`Q` ⊆ Ω
`ql`	Untere Grenze des Anfragebereichs Q	`Punkt im Basisraum` Ω
`qh`	Obere Grenze des Anfragebereichs Q	`Punkt im Basisraum` Ω
k	Dimensionsnummer: Die Nummer identifiziert die Sortierdimension $k \in \{1, \ldots, d\}$	`Integer`
P_R`[]`	Seitenmenge, die alle Tupel der Relation *R* enthält und als UB-Baum organisiert ist	`UB-Baum`
`E[]`	Ergebnisfolge, die nach Dimension *k* sortiert ist	`Liste`
`cache[]`	Zwischenspeicher, in dem die Datensätze, die in den Anfragebereich fallen, zwischengespeichert werden. `cache` ist als Heap implementiert. `cache[0]` bezeichnet das kleinste Element im Heap	`Heap`
`a`	Untere Grenze des Verarbeitungsbereichs in der Sortierdimension *k*. `a` $\in \Omega_k$	`Integer`
`b`	Obere Grenze des Verarbeitungsbereichs in der Sortierdimension *k*. `b` $\in \Omega_k$	`Integer`
`grenze`	Attributswert der Sortierdimension *k* der oberen Grenze u des Anfragebereichs Q	`Integer`
`p`	Seite: Ein Array von Tupeln	`Seite`

`x`	Tupel	`Tupel`
$W_{[a,b],k,[[ql,qh]]}(P_R)$	Seitenwelle, die als Liste realisiert ist	`Liste von Seiten`
$W_{[a,b],k,[[ql,qh]]}(P_R)$`[n]`	*n*-te Seite in der Seitenwelle	`Seite`
`size_of_slice()`	Gibt die Breite der nächsten Scheibe in Sortierrichtung zurück. Es entspricht somit dem Wert `b – a` des nächsten Verarbeitungsbereichs [a,b]	`Integer`
`content(`*p*`)`	Liste von Tupeln, die auf Seite *p* enthalten sind	`Array von Tupel`
`null`	leeres Element, identifiziert z.B. das Ende einer Liste	
`n`	Index für die Seitenwelle	`Integer`
`m`	Index für die Ergebnisfolge	`Integer`

Tabelle 4-2

Marke	SORTIERTES LESEN

```
        input:      ql     : Untere Grenze des Anfragebereichs  Q
                    qh     : Obere Grenze des Anfragebereichs Q
                    k      : Sortierdimension
                    P_R[]  : Seitenmenge als UB-Baum organisiert, die R enthält

        output:     E[ ]   : Eine nach k sortierte Folge von Tupeln, die in Q enthalten sind
        begin

              cache[] = ∅;              //Initialisiere den Tupelcache
                                        // cache ist als Heap organisiert
              a,b = ql_k;               //Initialisiere den Verarbeitungsbereich
              grenze = qh_k;            // Setze die Abbruchbedingung
              m     = 0;                //Index für die Ergebnisfolge

              // Durchlaufe die Schleife solange, bis die Sweep-Hyperebene die obere
              // Grenze erreicht hat.
M1            while (a ≤ grenze)
                    //lese alle Seiten der aktuellen Seitenwelle W_[a,b],k,[[ql,qh]] aus
                    // dem UB-Baum und füge die Tupel, die den Anfragebereich Q
                    // und den Verarbeitungsbereich [a,b] schneiden, in den Tupelcache
                    // cache[] ein, der als Heap organisiert ist. Schlüssel des Heaps
                    //ist die Sortierdimension

                    n = 0;
M2                  while(p := W_[a,b],k,[[ql,qh]](P_R)[n])≠ null do
                          cache := cache∪{x| (x∈content(p))∧
                                             (x∈[[ql,qh]]) ∧
                                             (x_k∈[a,b])};
                          ν := ν + 1;
                    end
                    // gib alle Tupel x in sortierter Ordnung aus.
                    // Da der Tupelcache als HEAP
                    // organisiert ist, steht an cache[0] das kleinste Element
                    // des Caches.

M3                  while cache[0] ≠ null do
                          E[m]:= cache[0];
                          entferne das Tupel cache[0] aus dem HEAP;
                          erzeuge die Heap-Eigenschaft im cache[];
                          m = m + 1;
                    end
M4                  a := b + 1
                    b := a + size_of_slice()      //setze b auf den neuen
                                                  // Verarbeitungsbereich
              end while
        end
```

Abbildung 4-5:Sortiertes Lesen

4.3.3 Korrektheit des Algorithmus

In diesem Abschnitt wird ein formaler Korrektheitsbeweis für den Algorithmus, der in Abschnitt 4.3.2 vorgestellt wurde, angegeben. Die formale Spezifikation des Sortierproblems lautet:

Gegeben ist eine Folge von Tupeln

$$\langle x_{(0)},x_{(1)},x_{(2)},...,x_{(n-1)}\rangle \text{ mit } x_{(0)} \prec x_{(1)} \prec x_{(1)} \prec ... \prec x_{(n-1)} \qquad \text{Gl: 4-21}$$

Hierbei bezeichnet $\prec$ die Z-Ordnung. Der Sortieralgorithmus soll nun eine Permutation der Folge erzeugen, für die gilt:

$$\langle x_{\pi(0)},x_{\pi(1)},x_{\pi(2)},...,x_{\pi(n-1)}\rangle \text{ mit } x_{\pi(0),k} \le x_{\pi(1),k} \le x_{\pi(2),k} \le ... \le x_{\pi(n-1),k} \qquad \text{Gl: 4-22}$$

Das Symbol „≤ “ bezeichnet die Ordnungsrelation auf der Domäne von Attribut A_k. $x_{(i)}$ oder *get*($\langle R\rangle$,*i*) bezeichnet das *i*-te Element der Folge, x_k den Wert des Attributes A_k des Tupels *x*. $x_{(i),k}$ bestimmt somit den Wert des Attributs A_k des *i*-ten Elements der Folge. Des Weiteren gilt die Randbedingung, wonach die Tupelmenge in einem UB-Baum abgelegt ist. Somit existieren eine Seitenmenge P_R und eine entsprechende Regionszerlegung Θ, in der die Tupelmenge enthalten ist. Die Eingabe des Algorithmus ist somit die Seitenmenge P_R. Wir bezeichnen eine Folge $\langle R\rangle$ als sortierte Folge von Tupeln $x \in \langle R\rangle$, wenn sie mindestens nach einem Attribut A_k bezüglich einer Ordnungsrelation „≤“ sortiert ist.

Definition 4-5: Sortierte Folge ($\langle R\rangle$,≤,*k*)

Eine Folge $\langle R\rangle$ einer Relation *R* und einer Ordnungsrelation „≤“ ist eine *sortierte Folge* ($\langle R\rangle$,≤,*k*) genau dann, wenn gilt:

$$\forall i, j : i < j \Rightarrow x_{(i),k} \le y_{(j),k} \text{ mit } i,j \in \{0,...,|R|-1\}$$

|*R*| bezeichnet die Anzahl der Tupel, die in der Relation *R* enthalten sind. Für eine 2-dimensionale Tupelfolge und die Ordnungsrelation ≤ auf den natürlichen Zahlen ist somit $\langle(0,15),(0,13),(0,14),(3,12),(3,15),(15,0)\rangle$ eine sortierte Folge bezüglich Dimension 1. Für die Konkatenation von *n* Tupelfolgen $S_1^{-t} \circ S_2^{-t} \circ ... \circ S_n^{-t}$ schreibt man $\bigcirc_{i=1}^{n} S_i^{-t}$

Satz 4-1:

Sei ($\langle R\rangle$,≤,*k*) eine sortierte Folge, dann gilt:

$$x_{(i),k} < y_{(j),k} \Rightarrow i < j$$

Beweis:

Ergibt sich aus Definition 4-5.
q.e.d.

Satz 4-2: Korrektheit des sortierten Lesens

Gegeben ist eine Regionszerlegung Θ mit der entsprechenden Seitenmenge P_R für die Relation *R* und ein Anfragebereich *Q* = [[*ql*,*qh*]] sowie die

Sortierdimension *k*. Dann erzeugt der Algorithmus in Abbildung 4-5 eine sortierte Folge von Tupeln bezüglich der Sortierdimension *k*.

Beweis: Satz 4-2

Betrachten wir zunächst die WHILE-Schleife. Hierzu führen wir die Iterationsvariable *i* ein. Sie bezeichnet die aktuelle Anzahl der Ausführungen der WHILE-Schleife. Vor der Ausführung der Schleife ist die sortierte Folge $(\langle E\rangle,\leq, k) = \varnothing$ und die Iterationsvariable $i = 0$. Hierzu wird folgende Invariante, die bei jedem Iterationsschritt erfüllt sein muss, definiert:

Invariante:

Die *sortierte Teilfolge* $(\langle E\rangle, \leq, k)_{i-1}$ ist die Verknüpfung der disjunkten aufsteigend sortierten Scheibenfolge $(\langle S_1^{-t}, S_2^{-t},..., S_{i-1}^{-t}\rangle, \leq,k) = (\langle S_1^{-t}\rangle, \leq,k) \circ (\langle S_2^{-t}\rangle,\leq,k) \circ (\langle S_3^{-t}\rangle,\leq,k) \circ ... \circ (\langle S_{i-1}^{-t}\rangle,\leq,k) = \overset{i-1}{\underset{m=1}{O}}(\langle S_m^{-t}\rangle,\leq,k) = (\langle E\rangle,\leq,k)_{i-1}$

Hierbei bezeichnet „°“ die Konkatenation von Folgen, d.h. $\langle 1,2,3,4,5\rangle \circ \langle 6,7,8,9,10\rangle = \langle 1,2,3,4,5,6,7,8,9,10\rangle$.

$\overset{i-1}{\underset{m=1}{O}}(\langle S_m^{-t}\rangle,\leq,k)$ ist eine Kurzschreibweise.

1. Terminierung:

Zunächst wird die Terminierung des Algorithmus betrachtet. Dies ist gleichbedeutend mit der Terminierung der WHILE-Schleife an Marke **M1**[5]. Die Terminierung ist abhängig von der Lage des Verarbeitungsbereichs [*a*,*b*]. Die Abbruchbedingung lautet:

$$a \leq grenze \qquad \text{Gl: 4-23}$$

Die Variable grenze = qh_k ist somit der Wert der oberen Grenze des Anfragebereichs Q = [[*ql*,*qh*]] in der Sortierdimension *k*. Der Algorithmus terminiert, wenn die untere Grenze des Verarbeitungsbereichs *a* größer als die obere Grenze qh_k des Anfragebereichs in der Sortierdimension *k* ist.

Die obere Grenze *b* und die untere Grenze *a* des Verarbeitungsbereichs [*a*,*b*] wird mit dem Attributwert A_k der unteren Grenze *ql* des Anfragebereichs Q = [[*ql*,*qh*]] initialisiert:

$$a,b = ql_k \qquad \text{Gl: 4-24}$$

Fall 1: *qh* = *ql*

Aus $(qh = ql) \Rightarrow (qh_k = ql_k) \Rightarrow (a = qh_k)$. Da im ersten Durchlauf durch die WHILE-Schleife (**M1**) an Marke (**M4)** die untere Grenze *a* des Verarbeitungsbereichs auf die obere Grenze des Verarbeitungsbereichs *b* + 1 gesetzt wird, ist $a > qh_k$. Die Abbruchsbedingung Gl: 4-23 ist erfüllt. Der Algorithmus terminiert.

[5] Marke in Abbildung 4-5:Sortiertes Lesen

Fall 2: $qh > ql$

Es gilt:

$(iter(a, i+1) = iter(b, i) + 1) \wedge (iter(b, i) \geq iter(a, i)) \Rightarrow (iter(a, i+1) - iter(a, i) \geq 1)$

Somit ist a spätestens nach $(qh_j - ql_j)+1$ Iterationen größer als die Variable *grenze*. Der Algorithmus terminiert.

2. Induktionsanfang: i = 0
Die Invariante muss zu Beginn des Algorithmus wahr sein. Da die Ergebnisfolge E zu Beginn $\varnothing$ ist, ist die Invariante erfüllt.

3. Induktionsschritt: i -1⇒ i
Jetzt wird der Induktionsschritt betrachtet. Das bedeutet, dass die Invariante bei jedem Iterationsschritt wahr bleiben muss. Es ist also nun zu zeigen, dass

$$\left(\langle E\rangle, \leq, k\right)_{i-1} \Rightarrow \left(\langle E\rangle, \leq, k\right)_{i} \qquad \text{Gl: 4-25}$$

Bei der Ausführung der WHILE-Schleife wird die Seitenwelle $W_{[a,b],k,[[l,u]]}(P_R)$ aus dem UB-Baum gelesen und die Tupel, die den Anfragebereich[[*ql*,*qh*]] und den Verarbeitungsbereich [*a*, *b*] schneiden, in den Cache[] eingefügt.

```
while(p := W[a,b],k,[[ql,qh]](PR)[n])≠ null do
      cache := cache ∪{x|(x ∈ content(p))∧
                        (x ∈ [[ql,qh]])  ∧
                        (xk ∈ [a,b])  )};
            n := n + 1;
end
```

Der Cache ist als Heap organisiert. Der Schlüssel des Heaps ist die Sortierdimension *k*, so dass auf cache[0] das aktuell minimale Element steht. Alle Tupel werden aus dem Cache entfernt und bilden die sortierte Scheibe $(\langle S\rangle_i, \leq, k)$.

```
while cache[0] ≠ null do
      E[m]:= cache[0];
      entferne das Tupel cache[0] aus dem HEAP;
      erzeuge die Heap-Eigenschaft im cache[];
      m = m + 1;
end
```

Nach Induktionsannahme ist $(\langle E\rangle, \leq, k)_{i-1}$ eine nach Dimension k sortierte Teilfolge. Es gilt für die Elemente der Teilfolge $(\langle E\rangle, \leq, j)_{i-1}$ nach Gl: 4-19:

$$\forall x \in (\langle E\rangle, \leq, k)_{i-1} : x < iter(a, i). \qquad \text{Gl: 4-26}$$

Somit folgt für die aktuelle sortierte Scheibe S_i^{-t}:

$$(\langle E\rangle, \leq, k)_{i-1} \cap S_i^{-t} = \varnothing. \qquad \text{Gl: 4-27}$$

Daraus folgt nun:

$$(\langle E\rangle,\leq,k)_{i-1}\circ S_i=(\overset{i-1}{\underset{j=1}{O}}(\langle S_j^{-t}\rangle,\leq,k))\circ S_i= \qquad \text{Gl: 4-28}$$

$$\overset{i}{\underset{j=1}{O}}(\langle S_j^{-t}\rangle,\leq,k)=(\langle E\rangle,\leq,k)_{i-1}$$

q.e.d.

Der Algorithmus ist somit korrekt. Seine wesentlichen Bestandteile sind:

- Laden der UB-Baumseiten, die die aktuelle Scheibe schneiden.
- Sortieren der Tupel der aktuellen Scheibe nach der „Sortier-Dimension“.

4.4 SRQ-Algorithmus

Der SRQ-Algorithmus (Sorted-Range-Query-Algorithmus) wurde in [Bay97b] vorgestellt. In [Bay97b] wird der Algorithmus als UB-Cache bezeichnet. Der Autor dieser Arbeit hat eine Prototypimplementierung in das kommerzielle Datenbanksystem TransBase von Transaktion integriert (siehe Kapitel 8.3). Er ist jedoch nicht als Produkt verfügbar. Es wird gezeigt, dass das Verfahren für gleichverteilte Daten herkömmlichen Zugriffsverfahren überlegen ist. Für beliebige Datenverteilungen kann es jedoch zu Leistungseinbrüchen kommen.

4.4.1 UB Range Query

Da der UB-Baum, der die mehrdimensionale Clusterung durch die Z-Kurve ausnutzt, eine mehrdimensionale Zugriffsmethode ist, können Bereichsanfragen effizient abgearbeitet werden. Die Grunderkenntnis der Arbeiten [Bay96], [Bay97a], [FriM97], [Mar99] ist, dass der Bereichsanfrage-Algorithmus die Selektivität auf allen Schlüsselattributen ausnutzen kann.

4.4.1.1 Algorithmus

An dieser Stelle wird kurz die grundlegende Idee des *UB-Bereichsanfragealgorithmus* (*RQ*) vorgestellt, denn er bildet die Grundlage für den SRQ. Der *RQ-Algorithmus* ermöglicht es, nur diese Seiten aus einem UB-Baum in den Arbeitsspeicher zu laden, die auch vom Anfragebereich geschnitten werden. Hierbei wird jede Seite genau einmal gelesen.

Der Bereichsanfragealgorithmus geht wie folgt vor:

Um eine Bereichsanfrage Q = [[*ql*,*qh*]] abzuarbeiten, wird zunächst für die untere Grenze *ql* die Z-Adresse mit Z(*ql*) berechnet und die entsprechende Z-Region, die Z(*ql*) enthält, in den Arbeitsspeicher geladen. Da der UB-Baum auf einem B*-Baum basiert und jede Region ρ genau auf eine Sekundärspeicherseite abgebildet wird, kann die Adresse der Z-Region ρ mit folgender SQL-Anweisung bestimmt werden:

```
SELECT MIN(Z-regionaddress) FROM UB
WHERE Z(ql) <= Z-regionaddress
```

Abbildung 4-6: UB-Baum-Zugriff

Hierbei wird von einer Tabellen-Struktur, die in Abbildung 4-7 dargestellt ist, ausgegangen:

```
CREATE TABLE UB(
            Z-regionaddress       INT   NOT NULL,
            data                  BINCHAR(pagesize)
            )
```

Abbildung 4-7: Schema der UB-Tabelle in SQL

Alle Tupel, die *Q* schneiden, werden in die Ergebnismenge *E* eingefügt. Danach wird der nächste Schnittpunkt bezüglich der Z-Ordnung mit *Q* berechnet. Die Adresse der Z-Region, die diesen Schnittpunkt enthält, kann mit der obigen SQL-Anweisung berechnet werden. Die entsprechende Seite *p* wird in den Arbeitsspeicher geladen und die Tupel, die in der Schnittmenge

$$\text{content}(p) \cap Q \qquad \text{Gl: 4-29}$$

liegen, der Ergebnismenge *E* hinzugefügt. Dies wird so lange wiederholt, bis eine Z-Region geladen wird, die die obere Grenze *qh* enthält.

Betrachten wir nun die Berechnung des nächsten Schnittpunktes. In [Bay96] wird ein Algorithmus vorgestellt, der auf der hierarchischen Struktur der Z-Kurve basiert. Das Prinzip wird anhand eines 2-dimensionalen UB-Baums mit Ursprung in der linken oberen Ecke erläutert (siehe Abbildung 4-8).

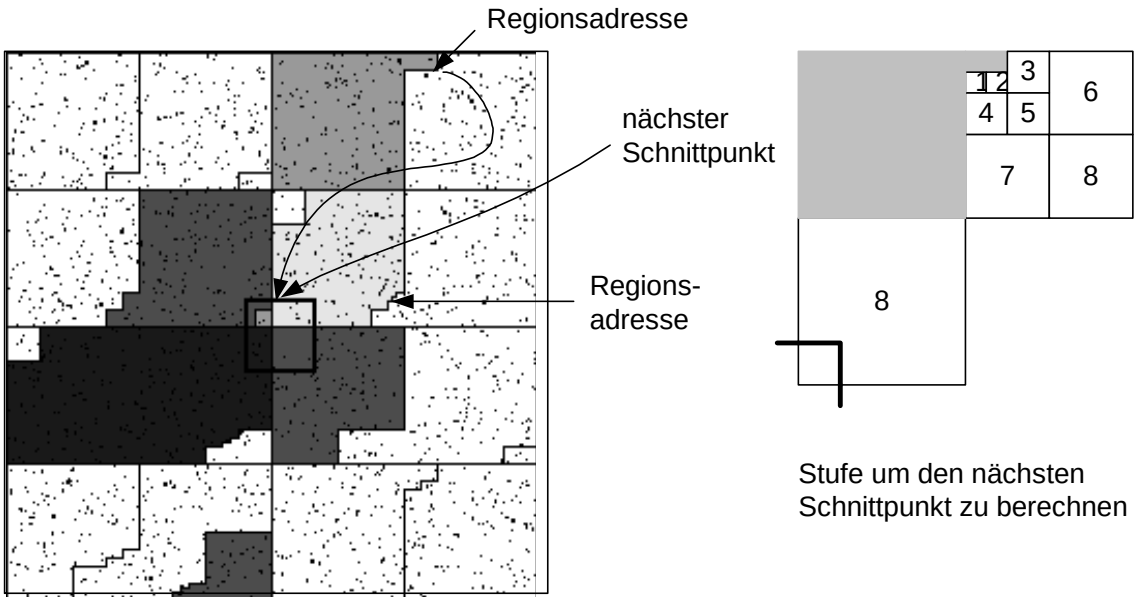


Abbildung 4-8: Nächster Schnittpunkt mit dem Anfragebereich *Q*

Um die Bereichsanfrage, die durch das schwarze Rechteck gekennzeichnet ist, abzuarbeiten, müssen 4 Regionen geladen werden (graue Regionen). Nachdem die zweite Region bezüglich der Z-Ordnung geladen wurde, liegt der Endpunkt der Z-Region nicht mehr in *Q*. Zum nächsten Schnittpunkt mit *Q* führt somit ein Sprung, der durch die geschwungene Line (siehe Abbildung 4-8) gekennzeichnet ist. Um den Endpunkt der Z-

Region ρ_2 genau zu spezifizieren, benötigt man 5 Adressstufen (Steps), somit 5 hierarchische Teilungen. Um den nächsten Schnittpunkt mit *Q* zu erhalten, geht man wie folgt vor: Man prüft, ob der nächste Unterwürfel auf Stufe 5 den Anfragebereich *Q* schneidet. Dieser Unterwürfel ist in Abbildung 4-8 mit der Nummer 1 gekennzeichnet. Da der Unterwürfel 1 nicht den Bereich *Q* schneidet, wird nun der nächste Bruder-Würfel (Nummer 2) in der Z-Ordnung getestet. Ein Bruder-Würfel ist ein Würfel auf derselben Stufe. Ein älterer Bruderwürfel ist hierbei der Würfel, dessen Z-Adresse größer als der aktuelle ist, das Umgekehrte gilt für den jüngeren Würfel. Sind alle Würfel einer Stufe untersucht und ist kein Schnittpunkt gefunden worden, geht man eine Hierarchiestufe höher. Den Würfel, der den soeben untersuchten Bereich abdeckt, bezeichnen wir als Vater. Da dieser Bereich bereits vollständig abgedeckt worden ist, geht man vom Vater zu seinem ältern Bruder und testet, ob der Anfragebereicht *Q* geschnitten wird. In unserem Beispiel ist das Würfel Nummer 3. Dieser Prozess wird solange wiederholt, bis ein Würfel den Bereich *Q* schneidet.

Betrachten wir nun den Prozess etwas genauer. Eine Z-Adresse wird in Stufen unterteilt (siehe Gl: 3-18). Die oberste Stufe wird mit 0, die kleinste Stufe wird mit *s*-1 gekennzeichnet. *s* ist die Anzahl der Bits, die benötigt wird, um den Wertebereich der jeweiligen Dimension A_i darzustellen (siehe Gl: 3-19). *s* bestimmt somit die Anzahl der Stufen einer Z-Adresse[6]. Der Wert einer Stufe gibt an, in welchem Teilraum das jeweilige Element liegt. Die Anzahl der Teilwürfel auf der Stufe *i* bestimmt sich aus der Anzahl der Dimensionen, die zu dieser Stufe beitragen. Da wir in unserem Modell nur den Basisraum Ω betrachten, in dem alle Dimensionen die Kardinalität *r* haben, ergeben sich somit

$$Teilwürfel = 2^d \qquad \text{Gl: 4-30}$$

Teilwürfel.

Im obigen Beispiel schneidet Teilwürfel 1 den Anfragebereich nicht. Somit wird der nächste Teilwürfel, also Teilwürfel 2, getestet. Teilwürfel 2 ist der nächste Teilwürfel nach der Z-Ordnung. Betrachten wir hierzu die Z-Adresse von Teilwürfel 1 und Teilwürfel 2.

Eine Z-Adresse wird hierzu in zwei Teile unterteilt, den *Präfix*, der mit $pre(\alpha,c)$ mit $c \in [0, s\text{-}1]$ bezeichnet wird, und den *Suffix*, der mit $suf(\alpha,c)$ mit $c \in [0, s\text{-}1]$ bezeichnet wird. *c* bestimmt die Stufe, ab der das Suffix der Z-Adresse beginnt. Das Suffix einer Z-Adresse geht somit von Stufe *c* bis Stufe (*s*-1). Der Wert ist wie folgt definiert:

$$pre(\alpha,c) = \begin{cases} \sum_{j=s-c}^{s-1} \sum_{i=1}^{d} x_{i,j} 2^{j \cdot d + i - 1} & ,c > 0 \quad \text{mit } x = Z^{-1}(\alpha) \\ 0 & ,c = 0 \end{cases} . \qquad \text{Gl: 4-31}$$

$$suf(\alpha,c) = \begin{cases} \sum_{j=0}^{s-c-1} \sum_{i=1}^{d} x_{i,j} 2^{j \cdot d + i} & ,c < s-1 \quad \text{mit } x = Z^{-1}(\alpha) \\ 0 & ,c = s-1 \end{cases} . \qquad \text{Gl: 4-32}$$

Hierbei bezeichnet *i* das Attribut A_i und *j* das Bit *j* (siehe Gl: 3-17 und Gl: 3-18). Für eine Z-Adresse gilt somit:

[6] Ohne Beschränkung der Allgemeinheit ist *s* in dieser Arbeit für alle Dimensionen gleich.

$$\alpha = per(\alpha, c) + suf(\alpha, c) \qquad \text{Gl: 4-33}$$

Beweis:

$$pre(\alpha, c) + suf(\alpha, c) \overset{x=Z^{-1}(\alpha)}{=} \sum_{j=s-c}^{s-1} \sum_{i=1}^{d} x_{i,j} 2^{j \cdot d + i - 1} + \sum_{j=0}^{s-c-1} \sum_{i=1}^{d} x_{i,j} 2^{j \cdot d + i - 1} =$$

$$\overset{(s-1-c)+1=s-c}{=} \sum_{j=0}^{s-1} \sum_{i=1}^{d} x_{i,j} 2^{j \cdot d + i - 1}$$

q.e.d.

Es wird nun das Konzept des Z-Würfels eingeführt.

Definition 4-6: Z-Würfel

Ein Z-Würfel $[\alpha| c]$ ist ein Teilraum des *d*-dimensionalen Basisraums Ω mit gleicher Kantenlänge in jeder Dimension, der genau durch ein Z-Intervall abgedeckt wird. Für das Z-Intervall gilt:

$$[pre(\alpha,c), pre(\alpha,c) + 2^{(s-c)d} - 1] \text{ mit } c \in [0, s\text{-}1]$$

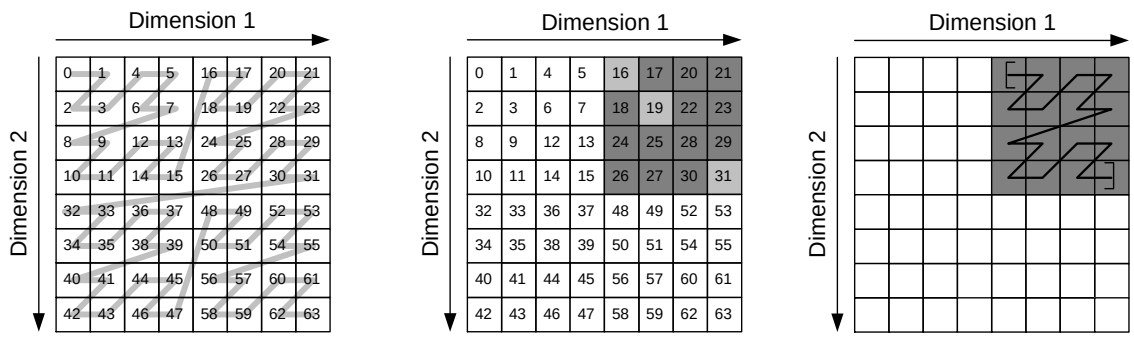
s bezeichnet die Anzahl der Stufen der Z-Adresse α

Beispiel 4-1:
Gegeben sei ein 2-dimensionaler Basisraum Ω. Die Stufentiefe *s* sei 3. Die Z-Adresse α sei α = 01.00.11 (binäre Darstellung). *c* hat den Wert 1. Für $x = Z^{-1}(\alpha)$ ergibt sich für x_1 =101 und x_2 = 001 (binäre Darstellung).
Für *pre*(α,c) ergibt sich somit:

$$pre(01.00.11,1) \overset{x_1=101; x_2=001}{=} \sum_{j=3-1}^{3-1} \sum_{i=1}^{2} x_{j,i} 2^{j \cdot d + i - 1} = 1 \cdot 2^{2 \cdot 2 + 1 - 1} + 0 \cdot 2^{2 \cdot 2 + 1} = 2^4 = 16$$

$$pre(\alpha, c) + 2^{(3-1)2} - 1 = 16 + 2^4 - 1 = 16 + 16 - 1 = 31$$

Man erhält somit das Z-Intervall [16,31]. Das Ergebnis ist in Abbildung 4-9 dargestellt.



(a) Z-Kurve/ 3-te Ebene (b) Punkte (c) Z-Intevall des Z-Würfe

Abbildung 4-9: Z-Würfel

Wie man in Beispiel 4-1 sieht, ist das Ergebnis ein Teilraum, der in beiden Dimensionen die gleiche Kantenlänge 2^2 besitzt.

Satz 4-3: Zusammenhang von Z-Intervall und Z-Würfel

Ist ein Z-Intervall $[\beta:\chi]$ das Z-Intervall eines Z-Würfels $[\beta|c]$ im Basisraum Ω, dann gilt:

1. $pre(\beta, c) = pre(\chi,c)$
2. $suf(\beta,c) = 0$
3. $suf(\chi,c) = 2^{(s-c)\cdot d} - 1$

Um Satz 4-3 zu veranschaulichen, betrachten wir die Bitrepräsentation eines Z-Würfels. Gegeben sei ein 2-dimensionaler Basisraum Ω mit einer Stufentiefe s = 4 und einem Z-Würfel [01.01.*.*|2]. Die ***.*** in der Z-Adresse verdeutlichen, dass die Werte beliebig gesetzt werden können. Das Präfix, also der Ursprung des Z-Würfels, ist somit der Wert 01.01. Für das Minimum ergibt sich:

$$pre(\beta|2) + suf(\beta|2)=01.01.00.00$$

Für das Maximum entsprechend

$$pre(\chi|2) + suf(\chi|2)=01.01.11.11$$

Beweis: Satz 4-3

1) $per(\beta,c) = pre(\chi,c)$: Ergibt sich direkt aus Definition 4-6. q.e.d.
2) $suf(\beta,c) = 0$: Ergibt sich direkt aus Definition 4-6. q.e.d.
3) $suf(\chi,c) = 2^{(s-c)\cdot d} - 1$: Ergibt sich direkt aus Definition 4-6. q.e.d.

Mit diesem Formalismus kann nun die Z-Würfel-Ursprungsstufe wie folgt definiert werden:

Definition 4-7: Z-Würfel-Ursprungsstufe ZU

Eine *Z-Würfel-Ursprungsstufe* für einen gegebenen Z-Würfel $[\alpha|c]$ mit $c \in [1,s-1]$ ist

$$ZU(\alpha,c)= \underbrace{\alpha_{d\cdot(c-1)+(d-1)}\alpha_{d\cdot(c-1)+(d-2)}\ldots\alpha_{d\cdot(c-1)}}_{\text{step c}} = \sum_{i=0}^{d-1} \alpha_{d\cdot(c-1)+i}\, 2^i$$

Basierend auf diesem Formalismus betrachten wir das Beispiel in Abbildung 4-8 nochmals. Da also der Z-Würfel 2 nicht Q schneidet, wird der nächste Z-Würfel in der Z-Ordnung betrachtet. Dazu muss auf die Ursprungsstufe des zweiten Z-Würfels 1 addiert werden. Die Z-Würfel-Ursprungsstufe hat den Wert (binäre Schreibweise):

$$ZU(01.01.00.00.11.*,5) = 11 \qquad \text{Gl: 4-34}$$

Da die Stufe jedoch bereits voll ist, kommt es zu einem Überlauf, d.h. der Z-Würfel hat bereits den maximalen Ursprungswert auf Stufe 4. Das liegt daran, dass der Wertebereich einer Stufe durch die Anzahl der Dimensionen, die zu dieser Unterteilungsstufe beitragen, bestimmt wird (siehe Definition 3-21). Somit ist der Wertebereich $[0,2^d\text{-}1]$. Für das 2-dimensionale Beispiel beträgt der Wertebereich demzufolge [0,3], was genau der vierten Stufe der Z-Adresse entspricht. Durch den Überlauf wird also der Wert der höheren Stufen beeinflusst. Diesen Vorgang kann man formal mit einer Addition beschreiben. Hierzu stellt man sich die Z-Adresse als binären String vor und addiert auf die Z-Würfel-Ursprungsstufe den Wert 1. In dem Beispiel wird auf dem binären String ab Stufe 4 der Wert 1 addiert.

$$01.01.00.00.11.\underbrace{**.\cdots.**}_{suf(a,c)} + \underbrace{2^{(s-c)d}}_{\substack{\text{Addition von}\\ \text{Wert 1 auf Stufe c-1}}} = 01.01.00.01.00.\underbrace{**.\cdots.**}_{suf(a,c)} \qquad \text{Gl: 4-35}$$

Nun könnte man iterativ diese Z-Adressen erzeugen und für jede dieser Adressen testen, ob sie den Anfragebereich schneiden. Dies würde dann zu

$$2^{(s\text{-}c)d} \qquad \text{Gl: 4-36}$$

Vergleichen führen.

Da dieser Algorithmus jedoch das Volumen des Z-Würfels exponentiell pro Stufe steigert, müssen im schlechtesten Fall

$$number\_of\_subcubes = \left(2^d - 1\right) \cdot steps \qquad \text{Gl: 4-37}$$

Z-Würfel untersucht werden, um den nächsten Schnittpunkt mit *Q* zu finden. Der genaue Algorithmus ist in [Bay96] zu finden.

E/A-Komplexität

Eine wesentliche Erkenntnis ist, dass zur Berechnung des nächsten Schnittpunktes keine zusätzliche E/A-Operation benötigt wird. *n* bezeichnet die Anzahl der geladenen Seiten. Die E/A-Komplexität für diesen Algorithmus ist somit linear zur Anzahl der geladenen Seiten, also O(*n*). Hier wurden jedoch nicht die E/A-Kosten für den Index-Zugriff berücksichtigt. Da es sich um einen B^*-Baum handelt, kommt noch ein logarithmischer Faktor hinzu, also

$$\log_m n, \qquad \text{Gl: 4-38}$$

der die Höhe des Baumes bestimmt. Für einen großen Verzweigungsgrad *m*, z.B. 100, und einen großen Datenbestand, wie z.B. $10^8 – 10^{10}$, beträgt die Höhe des Indexes zwischen 4 und 5 Knoten. Man sieht also, dass für einen großen Verzweigungsgrad die Baumhöhe sehr langsam steigt. Für das Data-Warehouse der GfK mit seiner Faktentabelle von 42.867.902 Tupeln ergibt sich bei einem Verzweigungsgrad von 100 eine Höhe von $\lceil 3.81 \rceil = 4$. Geht man davon aus, dass die ersten beiden Ebenen im Arbeitsspeicher liegen, hat man also noch einen konstanten Faktor von 2 Indexzugriffen. Somit ergibt sich für die E/A-Komplexität

$$c_{E/A} = n \underbrace{\log_m n}_{\text{für großes n}} \approx c \cdot n = O(n) \qquad \text{Gl: 4-39}$$

CPU-Komplexität

Der wesentliche Faktor der CPU-Komplexität ist die Bestimmung des nächsten Schnittpunktes mit dem Anfragebereich *Q* in Z-Ordnung.

Um also den nächsten Schnittpunkt zu finden, muss man maximal *s* (Anzahl der Stufen einer Z-Adresse) Stufen nach oben gehen. Pro Stufe muss man maximal

$$2^d\text{-}1 \qquad \text{Gl: 4-40}$$

ältere Brüder prüfen. An dieser Stelle ist noch einmal zu betonen, dass es sich hier nur um Rechenoperationen handelt und keine zusätzlichen E/A-Operationen benötigt werden. Ist nun der nächste Z-Würfel [β|c] gefunden, müssen die kartesischen Koordinaten des kleinsten Schnittpunktes mit dem Anfragebereich berechnet werden. Um die Berechnung der kartesischen Koordinaten besser zu verstehen, wird hier der Z-Würfel durch die kartesischen Koordinaten mit [β|c] = [[*wl*,*wh*]] repräsentiert. Diese Umrechnung ist eigentlich nicht notwendig, wird in der Prototypimplementierung nicht vorgenommen, es fallen also keine zusätzlichen CPU-Kosten an.

Die Bedingung, dass [[*wl*,*wh*]] $\cap$ *Q* = $\varnothing$ ist, kann formal so beschrieben werden:

$$\exists i : wh_i < ql_i \vee wl_i > qh_i \qquad \text{Gl: 4-41}$$

Die Bedingung, dass der Z-Würfel [[*wl*,*wh*]] den Anfragebereich *Q* schneidet, ist somit das Komplement der Formel :

$$\neg\exists i : wh_i < ql_i \vee wl_i > qh_i \qquad \text{Gl: 4-42}$$

Dies kann wie folgt umgeformt werden:

$$\forall i : wh_i \geq ql_i \wedge wl_i \leq qh_i \qquad \text{Gl: 4-43}$$

Somit ergibt sich für die kartesischen Koordinaten für den kleinsten Schnittpunkt *sp* mit dem Anfragebereich *Q* für die *i*-te Dimension:

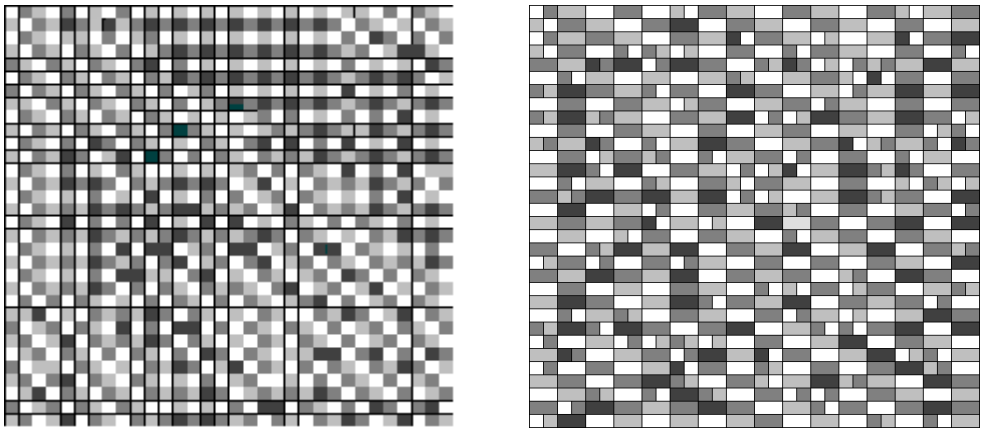
$$sp_i = \begin{cases} wl_i & , wl_i > ql_i \\ ql_i & , \text{sonst} \end{cases} \qquad \text{Gl: 4-44}$$

Mit Z(*sp*) wird nun eine Punktanfrage auf dem UB-Baum ausgeführt, um die nächste Region, die den Anfragebereich schneidet, zu laden. Die Anzahl der Vergleiche pro Stufe ist abhängig von der Anzahl der Dimensionen. Diese Anzahl steigt exponentiell. In [RamMF*00] haben wir einen Algorithmus beschrieben, der eine Bereichsanfrage in linearer Zeit bezüglich der Bitlänge der Z-Adresse abarbeitet. Einige Leistungsmessungen sind in [BayM98] zu finden.

4.4.1.2 Kostenmodell

Ein Kostenmodell für *idealisierte gleichverteilte* Daten wurde in [MarB00b],[Mar99] vorgestellt. Mit diesem Kostenmodell für UB-Bäume kann die Anzahl der Seiten abgeschätzt werden, die zur Abarbeitung einer Bereichsanfrage benötigt werden, d.h. die Seiten, die den Anfragebereich [[*y*,*z*]] schneiden.

Dieses Kostenmodell ist die Grundlage für die theoretische Analyse der hier vorgestellten Algorithmen.



(a) (b)

Abbildung 4-10: (a) Perfekte idealisierte gleichverteilte Partitionierung, (b) Statistisch gleichverteilte Partitionierung

Wenn man einen *d*-dimensionalen Basisraum Ω bezüglich jeder Dimension rekursiv in der Mitte teilt, erhält man $2^{e \cdot d}$ Unterwürfel, wobei *e* die Anzahl der rekursiven Unterteilungen ist. Hierbei ist die grundlegende Beobachtung, dass die daraus entstehende Partitionierung einem Schachbrett gleicht (Abbildung 4-10). Das bedeutet, dass die UB-Partitionierung eine Art Gitter erzeugt, wobei jede Region genau auf eine Zelle dieses Gitters abgebildet und somit als Z-Würfel angesehen werden kann. Hier stellt sich jedoch die Frage, unter welchen Bedingungen diese Art von Partitionierung entsteht. Dies führt zu dem Konzept der idealisierten gleichverteilten Zerlegung (Partitionierung).

Definition 4-8 Idealisierte gleichverteilte Zerlegung [Mar99]

Ein Menge von *P* Adressen ist idealisiert gleichverteilt (Abbildung 4-10 zeigt ein 2-dimensionales Beispiel) genau dann, wenn sich alle Adressen nur in den Bits *e* bis $e - \lceil \log_2 P \rceil$ unterscheiden, und alle Bit-Kombinationen auf den Bitpositionen $e - \lfloor \log_2 P \rfloor$ existieren und die Bitpositionen 0 bis $\lceil \log_2 P \rceil - 1$ gesetzt sind. Eine Zerlegung des Basisraums Ω, die durch eine Sequenz von idealisierten gleichverteilten Adressen erzeugt wird, ist eine idealisierte gleichverteilte Zerlegung.

Eine idealisierte gleichverteilte Zerlegung erzeugt also rechteckige Z-Regionen. Die Partitionierung des Basisraums Ω besteht nur aus Z-Würfeln, genau dann, wenn folgende Gleichung gilt:

$$(\log_2 P) \bmod d = 0 \qquad \text{Gl: 4-45}$$

Das bedeutet $P = 2^{e \cdot d}$ für ein $e > 0$, $e \in \mathbb{N}$. Falls Gl: 4-45 gilt, bezeichnen wir die Verteilung als *perfekte idealisierte gleichverteilte Zerlegung* des Basisraums Ω (siehe Abbildung 4-10.a Perfekte idealisierte gleichverteilte Zerlegung), ansonsten als *unperfekte Zerlegung*.

Allgemein hat der Teilraum ρ ein Volumen (siehe Definition 3-18) von

$$V(\rho) = 2^{-\lceil \log_2 P \rceil} \qquad \text{Gl: 4-46}$$

Im Folgenden wird der Logarithmus

$$l = \log_2 P \qquad \text{Gl: 4-47}$$

als Teilungstiefe bezeichnet. Da in den meisten Fällen die Anzahl der Seiten nicht genau einer 2er Potenz entspricht, wird das Konzept der totalen Teilungstiefe $l_\downarrow$ eingeführt. Hierbei bezeichnet die totale Teilungstiefe $l_\downarrow$ die Anzahl der kompletten hierarchischen Teilungen einer Z-Regionszerlegung. Für eine idealisierte gleichverteilte Z-Regionszerlegung mit *P* Seiten wird die totale Teilungstiefe wie folgt berechnet:

$$l_\downarrow(P) = \lfloor \log_2 P \rfloor \qquad \text{Gl: 4-48}$$

Die Anzahl der vollständigen Teilungen (kompletten Teilungen) wird auf die einzelnen Dimensionen gleichmäßig verteilt. Somit ist die vollständige Teilungstiefe pro Dimension

$$l_{\downarrow/\mathrm{d}} := \left\lfloor \frac{l_\downarrow(P)}{d} \right\rfloor \qquad \text{Gl: 4-49}$$

Falls $l_\downarrow$ (P) *mod* $d \neq 0$ und $l_\downarrow$ (P) = *l*(P) wird der Basisraum vollständig in einigen Dimensionen einmal öfter aufgeteilt als in anderen. Da die Adressenberechnung in der Prototypimplementierung die Ordnung d, d –1, d –2, … , 2, 1 der Dimension zur Berechnung der Adresse verwendet, werden die am weitesten rechts stehenden Dimensionen zuerst geteilt. Die vollständige Teilungstiefe l_j (P) bezüglich der Dimension *j* ist somit

$$l_j(d,P) = \begin{cases} l_{\downarrow/d}(d,P)+1 & , \text{wenn} \lfloor \log_2 P \rfloor \bmod d \leq j \\ l_{\downarrow/d}(d,P) & , \text{sonst} \end{cases} \qquad \text{Gl: 4-50}$$

Mit Gl: 4-50 kann man nun die Anzahl der Schichten $n_j(d, P, y_j, z_j)$, die den Anfragebereich [[*y*,*z*]] in Dimension *j* schneiden, berechnen.

$$n(y_j, z_j, l_j) = \begin{cases} 2^{l_j} - \lfloor \hat{y}_j 2^{l_j} \rfloor & , \text{wenn } \hat{z}_j = 1 \wedge \hat{y}_j \neq 1 \\ \lfloor \hat{z}_j 2^{l_j} \rfloor - \lfloor \hat{y}_j 2^{l_j} \rfloor + 1 & , \text{sonst} \end{cases} \qquad \text{Gl: 4-51}$$

Die Funktion ^ bezeichnet die Normalisierungs-Operation eines beliebigen Wertebereichs auf [0,1] und ist wie folgt definiert:

$$\hat{a} = \frac{a - a_{\min} + 1}{a_{\max} - a_{\min} + 1} \qquad \text{Gl: 4-52}$$

Falls $l\downarrow$ (P) *mod* $d \neq 0$ und $l\downarrow$ (P) $\neq l$(P) gilt, existiert mindestens eine Dimension, die in einigen Bereichen bereits $\left\lfloor l_\downarrow(P) / d \right\rfloor + 1$ Mal geteilt ist, andere Bereiche jedoch erst $\left\lfloor l_\downarrow(P) / d \right\rfloor$ Mal. Da Teilungen zunächst eine Stufe vervollständigen, bevor sie auf der nächsttieferen Stufe angewandt werden, kann nur eine Dimension beide Arten von Teilräumen besitzen. Wir bezeichnen diese Dimension als statistische Dimension. Die Wahrscheinlichkeit einer nicht vollständigen Teilung wird in diesem Modell berücksichtigt. Die vollständigen Teilungsebenen erzeugen nur $2^{l_\downarrow(P)}$ Seiten, so dass zusätzlich $P - 2^{l_\downarrow(P)}$ Seiten benötigt werden, um die Tabellengröße zu erzeugen. Diese Seiten werden aus den $2^{l_\downarrow(P)}$ Seiten durch Teilung bezüglich der Dimension j erzeugt. Deshalb ist die statistische Wahrscheinlichkeit $p_j(d,P)$ einer zusätzlichen Teilung in Dimension j:

$$p_j(d,P) = \begin{cases} \dfrac{P}{2^{\lfloor \log_2 P \rfloor}} - 1 & , \text{wenn } j = d - \left(\lfloor \log_2 P \rfloor \bmod d \right) \\ 0 & , \text{sonst} \end{cases} \qquad \text{Gl: 4-53}$$

Falls man unvollständig geteilte Dimensionen in die Rechnung einbezieht, kann Gl: 4-51 wie folgt verallgemeinert werden: Die durchschnittliche Anzahl von Scheiben, die den Bereich $[y_j,z_j]$ in Dimension j schneiden, ist somit

$$\begin{aligned} n_j(d, P, y_j, z_j) = n(y_j, z_j, l_j(d, P)) + ((n(y_j, z_j, l_j(d, P) + 1) \\ - n(y_j, z_j, l_j(d, P))) \cdot p_j(d, P) \end{aligned} \qquad \text{Gl: 4-54}$$

Da im Modell davon ausgegangen wird, dass die Schlüsselattribute voneinander unabhängig sind, ergibt sich für einen Anfragebereich [[y,z]] die Anzahl der zu ladenden Seiten $P_{ub\text{-}rq}$ durch Multiplikation der einzelnen Dimensionen wie folgt:

$$P_{ub-rq}(d, P, y, c) = \prod_{j=1}^{d} n_j(y_j, z_j, l_j(d, P)) \qquad \text{Gl: 4-55}$$

4.4.2 Algorithmus

Für die Implementierung des Algorithmus wird von folgenden Randbedingungen ausgegangen:

- Die Tabelle liegt als UB-Baum vor.
- Die Anzahl der Seiten, die im UB-Baum gespeichert sind, ist bekannt und wird mit *P* bezeichnet.

Die Grundidee des SRQ-Algorithmus beruht auf der Erkenntnis, dass der UB-Baum für eine idealisierte gleichverteilte vollständige Zerlegung nur aus Z-Würfeln besteht. Ist zusätzlich die genaue Anzahl der Seiten bekannt, die im UB-Baum gespeichert sind, kann mit der Gleichung Gl: 4-50 die minimale Breite einer Scheibe genau berechnet werden. Wir bezeichnen sie als *statische Scheibe*, da sie unabhängig von der Datenverteilung unter der Annahme der Gleichverteilung berechnet wird und die Breite der Scheiben mit Ausnahme der Anfangs- und Endscheibe gleich ist. „*Minimal*“ bedeutet in diesem Zusammenhang, dass die feinste Unterteilung in der Sortier-Dimension herangezogen wird.

$$breite = 2^{s-l_{\downarrow j}} \qquad \text{Gl: 4-56}$$

Man kann nun das Konzept der statischen Scheiben einführen.

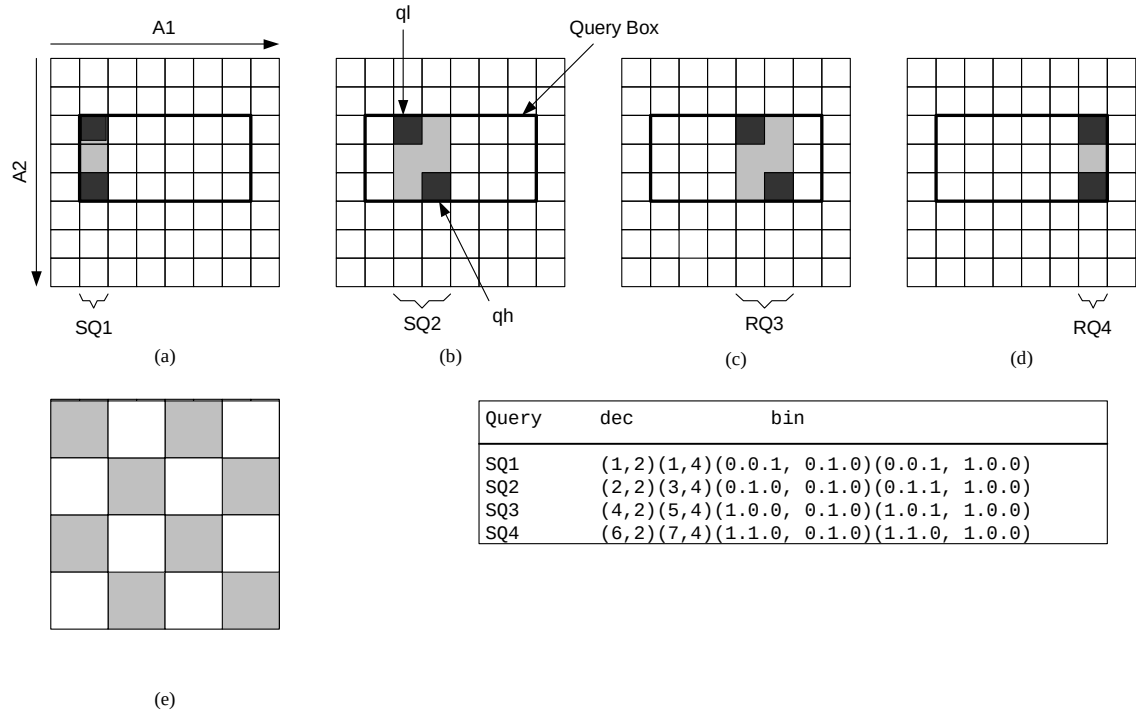
Definition 4-9: Statische Scheiben

Eine Sequenz von Scheiben $\langle S_1^{-t}, S_2^{-t}, ..., S_n^{-t} \rangle$ für einen Anfragebereich Q ist eine statische Scheibenmenge $\underline{S}$ genau dann, wenn gilt:

1) die Breite des Verarbeitungsbereichs $[a,b]$ der Sortierdimension k für alle $S_i^{-t} \in \underline{S}$ ist gleich außer für S_1^{-t} und S_n^{-t}
2) $Q = \bigcup_{i=1}^{n} S_i^{-t}$
3) $\forall x \in S_i^{-t}, y \in S_m^{-t} : x_k < y_k$ für $i < m$ und $i,m \in [i,n]$

Die Idee besteht nun darin, den Anfragebereich Q in eine Menge von statischen Scheiben aufzuteilen und die Sequenz von Scheiben $\underline{S}$ iterativ mit dem in Kapitel 4.4.1 beschriebenen Bereichsfrage-Algorithmus abzuarbeiten. Die Vereinigung der voneinander unabhängig sortierten Scheiben ergibt das sortierte Ergebnis.

Beispiel 4-2:
Abbildung 4-11e zeigt eine idealisierte gleichverteilte vollständige UB-Baum-Zerlegung mit 16 Seiten. Die vollständige Teilungstiefe pro Dimension ist $l_{\downarrow/d} = \log_2(16)/2 = 2$. Die maximale Auflösung r ist 8, so dass s = 3 ist. Somit ergibt sich für die minimale Breite $breite = 2^{s-l_{\downarrow/d}} = 2^{3-2} = 2$



Query	dec	bin
SQ1	(1,2)(1,4)(0.0.1, 0.1.0)(0.0.1, 1.0.0)	
SQ2	(2,2)(3,4)(0.1.0, 0.1.0)(0.1.1, 1.0.0)	
SQ3	(4,2)(5,4)(1.0.0, 0.1.0)(1.0.1, 1.0.0)	
SQ4	(6,2)(7,4)(1.1.0, 0.1.0)(1.1.0, 1.0.0)	

Abbildung 4-11: SRQ-Algorithmus

Neben der Breite muss noch die Lage der Scheiben bestimmt werden. Ziel ist es, die Lage der Scheiben so zu legen, dass bei einer idealisierten gleichverteilten UB-Baum-Zerlegung jede Region genau zu einer Scheibe gehört und somit genau nur einmal geladen wird. Die Lage der Scheiben orientiert sich somit an der rekursiven Unterteilung der Z-Kurve.

Abbildung 4-12 zeigt die Kernroutine des sortierten Lesens. Hierbei werden folgende Variablen und Mengebezeichnungen benutzt:

Symbol	Beschreibung	Abstrakter Datentyp
Q	Anfragebereich: Teilraum vom Basisraum Ω	Q ⊆ Ω
ql	Untere Grenze des Anfragebereichs Q	Punkt im Basisraum Ω
qh	Obere Grenze des Anfragebereichs Q	Punkt im Basisraum Ω
k	Dimensionsnummer: Die Nummer identifiziert die Sortierdimension $k \in \{1, \ldots, d\}$	Integer
P_R[]	Seitenmenge, die alle Tupel der Relation *R* enthält und als UB-Baum organisiert ist.	UB-Baum
E[]	Ergebnisfolge, die nach Dimension *k* sortiert ist	Liste
cache[]	Zwischenspeicher für die Tupel einer Scheibe	Array
sl	Untere Grenze der aktuellen Scheibe	Punkt im Basisraum Ω
sh	Obere Grenze der aktuellen Scheibe	Punkt im Basisraum Ω
P	Anzahl der Seiten, die im UB-Baum gespeichert sind	Integer
s	Anzahl der Stufen einer Z-Adresse	Interger
$l_{\downarrow/d}$	Vollständige Teilungstiefe pro Dimension	Integer
RQ(P_R,[[sl, sh]])	Bereichsanfrage-Algorithmus (siehe 4.4.1)	Array von Tupel
sort(cache,k)	Sortiert den Cache aufsteigend nach Attribut A_k	quick sort
|	ODER-Verknüpfung von Bits. z.B.: 0101 | 1111 :=1111	
&	UND-Verknüpfung von Bits, z.B.: 0101 & 1111 := 0101	
pow(2,s- $l_{\downarrow/d}$)	Die Funktion ist definiert: pow(2,s- $l_{\downarrow/d}$):= $2^{s-l_{\downarrow/d}}$	

Tabelle 4-3

SRQ-ALGORITHMUS

```
input:  ql      : Untere Grenze des Anfragebereichs Q
        qh      : Obere Grenze des Anfragebereichs  Q
        k       : Sortierdimension
        P_R[]   : Seitenmenge als UB-Baum organisiert, die R enthält
        d       : Anzahl der Dimensionen des UB-Baums
        P       : Anzahl der Seiten des UB-Baums
output: E[]     : Ein nach j aufsteigend sortierter Strom von Tupeln,
                  die in [[ql,qh]] enthalten sind
begin
    cache = ∅;                          //Initialisiere den Tupelcache
    sl = ql;                            //Initialisierung der ersten Scheibe
    sh = qh;                            //Initialisierung der ersten Scheibe
    l↓/d = ⌊log2(P)/d⌋ ;                // vollständige Teilungstiefe pro Dimension
    sh_k = ql_k ;                       // setze sh_k auf den Anfang der Sortierdimension
                                        // für die erste Scheibe
    sh_k= sh_k | (pow(2,s- l↓/d) –1);   // setze Bit von 0 bis s- l↓/d der Sortierdimension k
                                        // auf 1 (maximale Wert in der
                                        // Sortierdimension in der ersten Scheibe)
    do
        cache = RQ(PR,[[sl,sh]]); // Bereichsanfrage [[sl,sh]] auf  den UB-Baum
                                  // (aktuelle Scheibe)
        sort(cache,k) ;           // Sortiere Tupel der Scheibe nach Attribut k
        output (cache);           // liefere die sortierten Tupel an den Aufrufer
        sl_k = sl_k + pow(2,s- l↓/d); // bewege auf den Anfang der nächsten Scheibe
        slk = slk & ((pow(2,s) –1)  -     // setze  Bit 0 bis s- l↓/d der Sortierdimension
                (pow(2,s- l↓/d) –1));     //  k auf 0 (minimaler Wert in der
                                          // Sortierdimension  in der aktuellen
                                          // Scheibe
        sh_k =  sl_k;                     // bewege auf das Ende der nächsten
                                          // Scheibe
        sh_k = sh_k | (pow(2,s- l↓/d) –1); // setze Bit von 0 bis s- l↓/d der
                                          // Sortierdimension k auf 1(maximaler
                                          // Wert in der Sortierdimension
                                          // in der aktuellen Scheibe)

    while(sh_k ≤qh_k)
    cache = RQ(PR,[[sl,sh]]); // Bereichsanfrage [[a,b]] auf R
    sort(cache,k) ;           // Sortiere Tupel der Scheibe nach Attribut k
    output (cache);           // liefere die sortierten Tupel an den Aufrufer
end
```

Abbildung 4-12: SRQ-Algorithmus

4.4.3 Korrektheit

Die Korrektheit des Algorithmus ergibt sich aus der Invariante von Satz 4-2. Da für die Bereichsanfragen $[[sl,sh]]_i$, wobei *i* die i-te Iteration der WHILE-Schleife bedeutet, gilt:

$$(\forall a\ \forall b : a \in [sl_k,sh_k]_i \wedge b \in [sl_k,sh_k]_{i+1} \Rightarrow a < b) \text{ und } [sl_k,sh_k]_i \cap [sl_k,sh_k]_{i+1} = \varnothing \qquad \text{Gl: 4-57}$$

und die Schleife terminiert, da mindestens pro Durchgang sl_k um 1 erhöht wird, ist der Algorithmus korrekt und produziert eine sortierte Folge ($\langle E \rangle$, $\leq$, k).

4.4.4 Leistungsanalyse

Um den SRQ-Algorithmus zu bewerten, entwickeln wir zunächst eine Kostenfunktion für die Antwortzeit t_{ant} und die benötigte Cachegröße. Bei dieser theoretischen Analyse betrachten wir neben dem SRQ-Algorithmus einen konkatenierten Schlüsselindex (IOT, B*-Baum), das vollständige sequenzielle Lesen (FTS) und den Range-Query-Algorithmus auf UB-Bäumen (RQ).

Der Schnitt von mehreren Sekundärindexen (IV, Invertierung) wird nicht betrachtet, da er als Zugriffspfad für große Ergebnismengen, die in dieser Arbeit betrachtet werden, mit den oben genannt Zugriffspfaden nicht konkurrieren kann. Eine genaue Analyse ist in [FriM97], [BayM98] und [Mar99] zu finden.

Abgesehen vom SRQ-Algorithmus kommt nach der Lesephase, in der die Selektivitäten der einzelnen Dimensionen ausgewertet werden, eine Sortierphase. Somit bestehen die Gesamtkosten aus der Lesephase über den Zugriffspfad c_{lesen} und der Sortierphase $c_{sortieren}$.

4.4.4.1 E/A-Kostenmodell

Zunächst werden die Kosten, die durch E/A-Operationen verursacht werden, betrachtet. In Anlehnung an [HarR96] wird in dieser Arbeit ein Kostenmodell benutzt, das die Positionierungszeit t_π und die Datenübertragungszeit t_τ einer Festplatte berücksichtigt. Des Weiteren wird angenommen, dass pro E/A-Operation C Seiten vom Sekundärspeicher in den Arbeitsspeicher übertragen werden. Nach Gl: 3-6 beträgt die Zeit einer E/A-Operation

$$c_{E/A} = t_\pi + C\, t_\tau \qquad \text{Gl: 4-58}$$

Diesen Vorgang bezeichnen wir als *Prefetching* (Definition 3-10). Des Weiteren nehmen wir an, dass das Lesen c_l und Schreiben c_s von Daten gleich lange dauert. Für das „Sequenzielle Lesen“ bzw. „Schreiben“ mit einem *Prefetchingfaktor C* ergibt sich somit folgende Kostenfunktion, um n Seiten zu lesen bzw. zu schreiben:

$$c_l(n) = c_s(n) = c_{l/s}(n) = \left\lceil \frac{n}{C} \right\rceil \cdot c_{E/A} \qquad \text{Gl: 4-59}$$

Mit diesem Kostenmodell als Grundlage berechnen wir nun die Kosten für das „externe Sortieren“ nach Attribut A_j für eine mehrdimensionale Bereichsanfrage Q = [[ql,qh]] auf der Relation R bestehend aus P Seiten ($P >> C$) mit der Selektivität $S_{E(Q),R}$ (siehe Definition 3-33), einem Arbeitsspeicher der Größe M und einem Mischfaktor m. P_E ist die Anzahl der Seiten, die benötigt werden, um die Ergebnismenge $E(R,Q)$, die durch Q bestimmt wird, aufzunehmen. Für P_E mit Gl: 3-27 gilt:

$$P_E = S_{E(Q),R} P = P \prod_{i=1}^{d} S_i \qquad \text{Gl: 4-60}$$

Das externe Sortieren (Abschnitt 4.2) unterteilt sich in eine Initialisierungsphase, in der die einzelnen Initialisierungsläufe für die Mischphase erzeugt werden, und eine Sortierphase, in der das Sortieren durch Verschmelzen stattfindet. Mit den Selektivitäten auf den einzelnen Dimensionen ergibt sich somit für die Anzahl der Seiten P_{misch} der Mischphase des externen Sortierens für Gl: 4-12

$$P_{misch} = \begin{cases} \underbrace{2P_E \log_m \frac{P_E}{M}}_{\text{Mischphase}} & , \log_m \frac{P_E}{M} > 1 \\ \underbrace{2P_E}_{\text{Mischphase}} & , \text{sonst} \end{cases} \qquad \text{Gl: 4-61}$$

Die Kosten sind somit

$$c_{misch} = c_{E/A} \left\lceil \frac{P_{misch}}{C} \right\rceil \qquad \text{Gl: 4-62}$$

Die Initialisierungsphase P_{init} wiederum besteht aus einem Lese- und einem Schreibvorgang, in dem P_{lesen} und $P_{schreiben}$ Seiten gelesen bzw. geschrieben werden. Da nur Tupel, die zum Ergebnis E beitragen, wieder auf den Sekundärspeicher geschrieben werden, kann man $P_{schreiben} = P_E$ setzen. Die Kosten c_s, die dabei entstehen sind:

$$c_s = \left\lceil \frac{P_E}{C} \right\rceil c_{E/A} \qquad \text{Gl: 4-63}$$

Somit sind die Sortierkosten $c_{E/A\text{-}sort}$:

$$c_{E/A-sort} = c_s + c_{misch} = \left\lceil \frac{P_E}{C} \right\rceil c_{E/A} + \left\lceil \frac{P_{misch}}{C} \right\rceil c_{E/A} \le c_{E/A} \left(\left\lceil \frac{P_E + P_{misch}}{C} \right\rceil + 1 \right) \qquad \text{Gl: 4-64}$$

$$\underset{P_E + P_{misch} >> C}{\approx} \frac{c_{E/A}}{C} (P_E + P_{misch})$$

Zunächst müssen jedoch die Daten über einen Zugriffspfad in den Arbeitsspeicher M geladen werden. Da beim Lesen über die Zugriffspfade der Relation R nicht die volle Selektivität zum Tragen kommt, werden zu viele Seiten gelesen und somit gilt:

$$P_{lesen} \ge P_E \qquad \text{Gl: 4-65}$$

Zugriffspfad „vollständiges sequenzielles Lesen“:

Um die Relation R zu lesen erhält man für das vollständige sequentielle Lesen die Kosten $c_{E/A\text{-}fts}$:

$$c_{\text{E/A -fts}} = c_{E/A} \cdot \left\lceil \frac{P}{C} \right\rceil \qquad \text{Gl: 4-66}$$

Kombiniert man nun die Kosten des Zugriffspfads und des externen Sortierens, erhält man folgende Kostenfunktionen $c_{\text{E/A-fts-sort}}$:

$$c_{\text{E/A-fts-sort}} = c_{E/A-fts} + c_{E/A-sort} \le c_{E/A}\left(\left\lceil \frac{P + P_E + P_{misch}}{C} \right\rceil + 2\right) \qquad \text{Gl: 4-67}$$

$$\overset{P+P_E+P_{misch} >> C}{\approx} \frac{c_{E/A}}{C}(P + P_E + P_{misch})$$

Konkatenierten Schlüsselindex

Benutzt man einen konkatenierten Schlüsselindex, kann nur die Selektivität S_i des ersten Schlüsselattributs ausgenutzt werden. Somit ergibt sich für das Lesen der Relation R folgende Kostenfunktion $c_{\text{E/A-iot}}$:

$$c_{\text{E/A-iot}} = \underbrace{S_i \cdot P}_{P_{lesen}} \cdot c_{E/A} \qquad \text{Gl: 4-68}$$

Die Kosten des Zugriffspfads und des externen Sortierens sind dann:

$$c_{\text{E/A-iot-sort}} = c_{EA-iot} + c_{E/A-sort} = S_i \cdot P \cdot c_{E/A} + c_{E/A}\left(\left\lceil \frac{P_E + P_{misch}}{C} \right\rceil + 1\right) \qquad \text{Gl: 4-69}$$

$$= c_{E/A}\left(S_i \cdot P + \left\lceil \frac{P_E + P_{misch}}{C} \right\rceil + 1\right)$$

UB-Baum mit RQ-Algorithmus

Für den UB-Baum nehmen wir an, dass die d-dimensionale Relation R als d-dimensionaler UB-Baum organisiert ist. Für eine idealisierte gleichverteilte vollständige UB-Zerlegung ergibt sich für den RQ-Algorithmus nach dem Kostenmodell $c_{E/A\text{-}ub}$ aus Kapitel 4.4.1.2

$$c_{E/A-ub} = c_{E/A} \cdot P_{ub-rq} = c_{E/A} \cdot \prod_{j=1}^{d} \underbrace{n_j(d, P, y_j, z_j)}_{\text{Anzahl der schneidenden Z-Regionen in } A_j} \qquad \text{Gl: 4-70}$$

Mit den Sortierkosten $c_{sortieren}$ erhält man $c_{\text{E/A-ub-sort}}$:

$$c_{E/A-ub-sort} = c_{E/A-ub} + c_{E/A-sort} =$$
$$= c_{E/A} \cdot \prod_{j=1}^{d} \underbrace{n_j(d,P,y_j,z_j)}_{\text{Anzahl der schneidenden Z-Regionen in } A_j} + c_{E/A}\left(\left\lceil \frac{P_E + P_{misch}}{C} \right\rceil + 1\right) \approx \quad \text{Gl: 4-71}$$
$$\approx c_{E/A} \cdot P_E + \frac{c_{E/A}}{C}(P_E + P_{misch}) = c_{E/A}\left(P_E + \frac{P_E + P_{misch}}{C}\right)$$

UB-Baum mit SRQ-Algorithmus

Wie beim RQ-Algorithmus ist die *d*-dimensionale Tabelle *R* als UB-Baum organisiert. Es wird von einer idealisierten Gleichverteilung ausgegangen. Da der SRQ-Algorithmus die *Lese-* und *Sortierphase* miteinander verzahnt, fällt die nachträgliche Sortierung weg. Hierbei wird jedoch vorausgesetzt, dass die aktuelle Scheibe vollständig im Arbeitsspeicher gehalten wird. Somit ergibt sich für die Lesephase des SRQ-Algorithmus $c_{E/A\text{-}ub}$:

$$c_{E/A-ub} = c_{E/A} \cdot P_{ub-rq} = c_{E/A} \cdot \prod_{j=1}^{d} \underbrace{n_j(d,P,y_j,z_j)}_{\text{Anzahl der schneidenden Z-Regionen in } A_j} \approx c_{E/A} P \cdot \prod_{i=1}^{d} S_i \quad \text{Gl: 4-72}$$

Mit der Sortierung ergibt sich somit für den SRQ-Algorithmus $c_{E/A\text{-}SRQ}$:

$$c_{E/A-SRQ} = c_{E/A} \cdot \prod_{j=1}^{d} \underbrace{n_j(d,P,y_j,z_j)}_{\text{Anzahl der schneidenden Z-Regionen in } A_j} + c_{E/A}\left\lceil \frac{P_E}{C} \right\rceil = \quad \text{Gl: 4-73}$$
$$= c_{E/A}\left(\left\lceil \frac{P_E}{C} \right\rceil + \prod_{j=1}^{d} \underbrace{n_j(d,P,y_j,z_j)}_{\text{Anzahl der schneidenden Z-Regionen in } A_j}\right) \approx c_{E/A}\left(\left\lceil \frac{P_E}{C} \right\rceil + P_E\right)$$

In Tabelle 4-4 ist noch einmal das vollständige Kostenmodell, das für die theoretische Analyse herangezogen wird, zusammengestellt:

Bezeichner	Beschreibung
t_π	Durchschnittliche Positionierungszeit
t_τ	Durchschnittliche Block-Übertragungszeit
c_{lesen}	Kosten der Lesephase
$c_{E/A\text{-}sort}$	Kosten der Sortierphase
$O_{E/A}$	Asymptotische E/A-Komplexität
C	Prefetching-Faktor, liest C Seiten auf einmal
$c_{E/A}$	Kosten einer E/A-Operation
$c_l(n)$	Kosten für n E/A-Operationen (Lese-Operationen)
$c_s(n)$	Kosten für n E/A-Operationen (Schreibe-Operationen)
$S_{E(Q),R}$	Selektivität
M	Arbeitsspeicher
P	Anzahl von Seiten einer Tabelle

$P_{ub\text{-}rq}$	Anzahl der Seiten, die durch den Bereichsanfrage-Algorithmus gelesen werden
R	Relation
M	Mischfaktor
P_E	Anzahl der Seiten der Ergebnismenge E
P_{misch}	Anzahl der Seiten bei der Mischphase
c_{misch}	Kosten der Mischphase
c_s	Kosten der Schreibphase
$c_{E/A\text{-}fts}$	E/A-Kosten für FTS
$c_{E/A\text{-}fts\text{-}sort}$	E/A-Kosten für externes Sortieren mit FTS
$c_{E/A\text{-}iot}$	E/A-Kosten für IOT
$c_{E/A\text{-}iot\text{-}sort}$	E/A-Kosten für externes Sortieren mit IOT
$c_{E/A\text{-}ub}$	E/A-Kosten für UB-Baum
$c_{E/A\text{-}ub\text{-}sort}$	E/A-Kosten für externes Sortieren mit UB-Baum
$c_{E/A\text{-}SRQ}$	E/A-Kosten für SQR

Tabelle 4-4: *E/A*-Kostenmodell

4.4.4.2 Theoretische Analyse

In der theoretischen Analyse betrachten wir zunächst die zwei wesentlichen Kennzahlen, um die Anfrage in Abbildung 4-1 abzuarbeiten:

- Die Antwortzeit t_{ant}, die benötigt wird, um eine Bereichsanfrage Q sortiert nach Attribut A_k abzuarbeiten.
- Die erste Antwortzeit $t_{ant\text{-}1}$, die benötigt wird, um das erste Teilergebnis zu liefern.

Um diese Kennzahlen genau zu analysieren, gehen wir von dem in Kapitel 3.2.1.2 beschriebenen Kostenmodell aus. Heutige Betriebssysteme benutzen einen Prefetchfaktor $C = 8$. Somit werden bei einer E/A-Operation (wahlfreier Zugriff) vom Sekundärspeicher 8 Seiten geholt. Der Arbeitsspeicher (*Sortercache* M_{cache}), der für die Sortierung vom System bereitgestellt wird, sei 32 MB groß. Eine Gleichverteilung der Daten wird angenommen. Die betrachtete Tabelle bestehet aus 2097152 Seiten (Seitengröße 2K), was 4 GB entspricht.

Prefetchfaktor C (Seiten)	8
Seitengröße (KB)	2
Sortiercache M_{cache} (MB); (Seiten)	32 ; 16000
Tabellengröße (GB); (Seiten)	4 ; 2097152
Datenverteilung	Gleichverteilung

Tabelle 4-5: Analysescenario

Bei der Analyse der Kennzahlen werden folgende Parameter variiert:

- Dimensionalität d der Tabelle
- Selektivität S_i der einzelnen Dimensionen

Bei der Variation der oben genannten Parameter werden die übrigen auf einen konstanten Wert gesetzt. Der Einfluss von Caches auf die Kennzahlen werden bei der theoretischen

Analyse nicht berücksichtig. Sie stellen jedoch ein interessantes weiterführendes Forschungsgebiet dar.

4.4.4.2.1 Antwortzeit t_{ant} unter Variation des Selektivität des Anfragebereichs *Q*

Für unsere Kostenanalyse betrachten wir zunächst den 3-dimensionalen Fall. Wir nehmen für das externe Sortieren einen Mischfaktor *m* von 2 an. Hierbei werden *d*-1 Dimensionen auf eine konstante Selektivität eingeschränkt. Wir bezeichnen sie als *c*%-*Messung*. Diese Art des Selektivitätswachstums eignet sich besonders gut, um die Leistung von Zugriffspfaden in Abhängigkeit von der Selektivität des Anfragebereichs *Q* darzustellen.

Beispiel 4-3:
Abbildung 4-13 zeigt eine 50%-Messung für eine 2-dimensionalen Basisraum (c = 50). Hierbei wird Attribut A_2 auf 50 Prozent gesetzt und A_1 auf 25 Prozent, 50 Prozent und 75 Prozent gesetzt. Die einzelnen Anfragebereiche *Q* sind durch die gestrichelten Linien angedeutet.

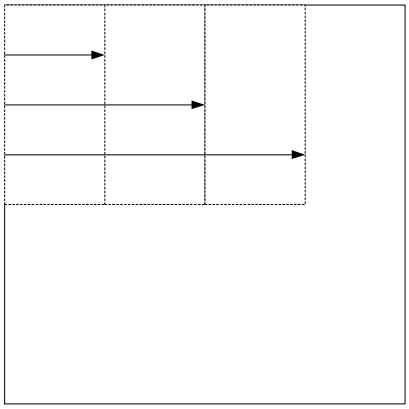


Abbildung 4-13: *c*%-Messung

Struktur der Algorithmen

Die Struktur der Algorithmen besteht, wie bekannt, aus der Selektionsphase, die durch einen Zugriffspfad unterstützt wird, und einem Nachfilterprozess, in dem die Tupel auf den Anfragebereich *Q* geprüft und gegebenenfalls der Ergebnismenge *E* hinzugefügt werden. Danach kommt eine Sortierphase, in der die Ergebnismenge *E* nach der Dimension *k* sortiert wird. Die E/A-Kosten, die bei dieser Struktur auftreten, sind in Abbildung 4-14 dargestellt. Sie zeigt eine 100 Prozent-Messung für einen 3 dimensionalen Basisraum **Ω**. Hierbei wird angenommen, dass die Dimension, auf der die Restriktion existiert, nicht identisch mit Sortierdimension *k* ist.

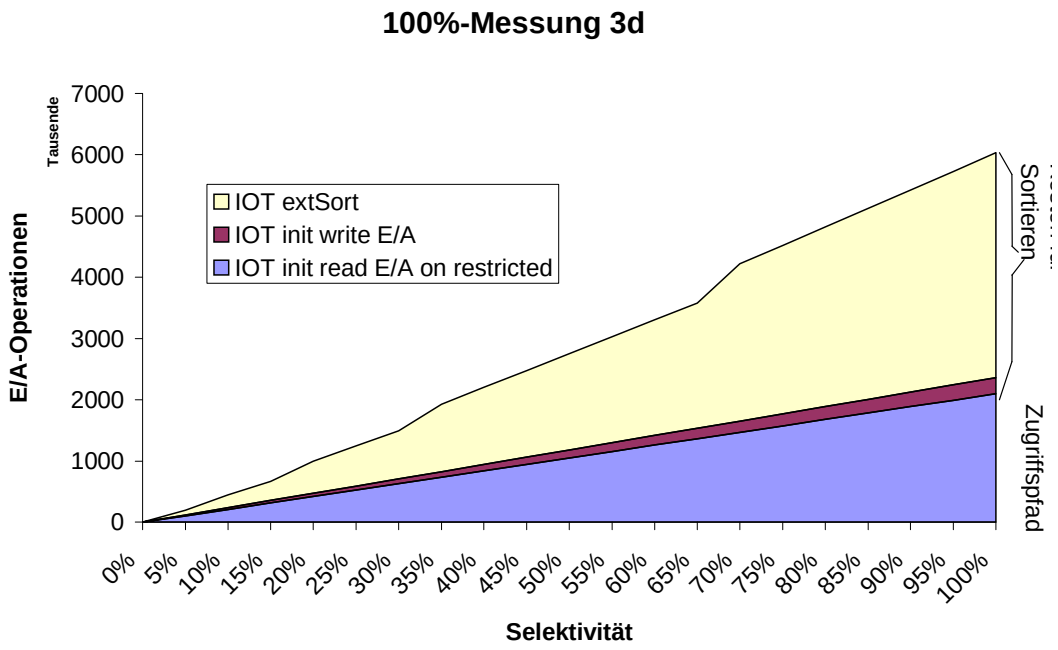


Abbildung 4-14: Bereichsanfrage mit externem Sortieren (Simulation)

Selektionsphase

Bei der Analyse wird die Selektionsphase durch die Zugriffspfade IOT und UB-Baum unterstützt. Steht kein Zugriffspfad zur Verfügung muss ein FTS ausgeführt werden. Die Kosten, die durch das Lesen über den Zugriffspfad entstehen, sind in Abbildung 4-14 durch die untere Fläche dargestellt. Ein sehr ausführlicher Vergleich der einzelnen Zugriffspfade für Bereichsanfragen ist in [Mar99], [FriM97] zu finden. Es werden hier nur die wesentlichen Ergebnisse noch einmal präsentiert.

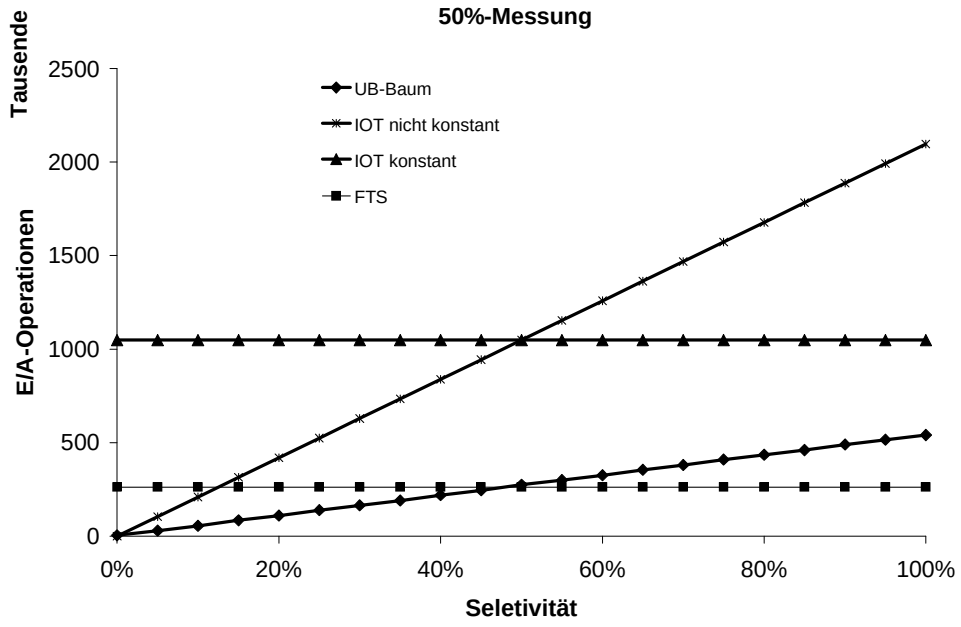


Abbildung 4-15: 50%-Messung 3d (Simulation)

Abbildung 4-15 zeigt eine Simulation der E/A-Kosten einer 50%-Messung auf einem 3-dimensionalen Basisraum Ω. Für die Simulation der E/A-Kosten wurde die E/A-Kostenfunktion aus Kapitel 4.4.4.1 verwendet. Es wird Dimension A1 von 0% bis 100% variiert, Dimension A2 und A3 werden auf 50% gesetzt.

Die E/A-Kosten für den Bereichsanfrage-Algorithmus auf dem UB-Baum steigen linear mit der Selektivität. Der IOT auf A1 steigt auch linear, da er die Restriktion auf der Dimension vollständig ausnutzen kann. Da der IOT auf A1 die Restriktionen auf A2 und A3 nicht ausnutzen kann, ist die Ausführung auf dem UB-Baum dem IOT auf A1 überlegen. Für dieses Beispiel erhält man eine Leistungssteigerung von Faktor 4.

Der IOT auf A2 zeigt ein konstantes E/A-Verhalten, da die Restriktion auf A1 nicht ausgenutzt werden kann. Da auf A2 eine 50% Restriktion existiert, werden ca. 1000000 Seiten geladen. Der Schnittpunkt Abbildung 4-15 zwischen dem IOT auf A1 und IOT auf A2 liegt bei einer Selektivität von 50 %. Der IOT auf A1 und der IOT auf A2 können jeweils die gleiche Restriktion auf der jeweiligen Dimension nutzen.

Der FTS zeigt auch ein konstantes Verhalten. Er liest die ganze Tabelle. Da er Prefetchfaktor C ausnutzt, werden pro E/A-Operation 8 Seiten geladen. Um alle Seiten zu laden, benötigt der FTS nur 262144 E/A-Operationen. Somit ist der FTS dem IOT auf Dimension A1 bereits ab ca. 13,5 % und dem UB-Baum ab ca. 50% überlegen.

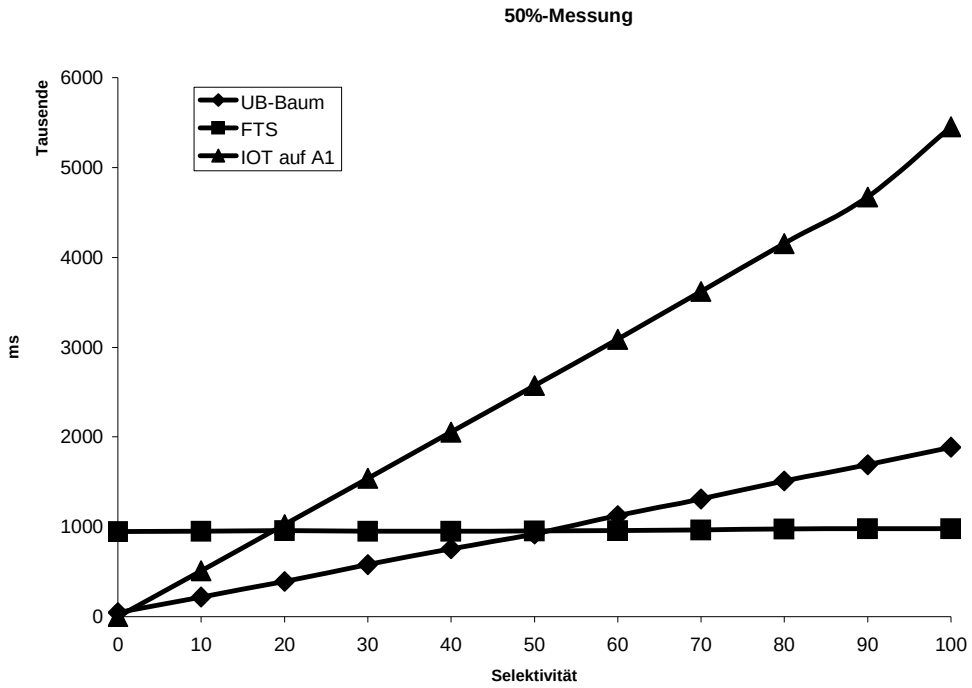


Abbildung 4-16: 50%-Messung auf einer 1G Datenbank mit gleichverteilten Daten

Abbildung 4-16 zeigt die Antwortzeit einer 50%-Messung. Die Messung ist auf einer 1 GB großen 3-dimensionalen Datenbank mit 2 KB Seiten ausgeführt worden. Die Daten sind gleichverteilt. Hierbei handelt es sich um die Gesamtkosten, d.h. die E/A-Kosten und die CPU-Kosten. Da der Graph das gleiche Verhalten wie die Simulation der E/A-Kosten zeigt, ist die Ausführung des Bereichsanfrage Algorithmus E/A-Bound und nicht CPU-Bound. Für die CPU Komplexität sei auf 4.4.4.4 verwiesen.

Sortierphase

In der Sortierphase wird die Ergebnismenge in die Zielordnung übergeführt. Wird das sortierte Lesen eingesetzt, ist die Sortierphase mit der Selektionsphase verzahnt.

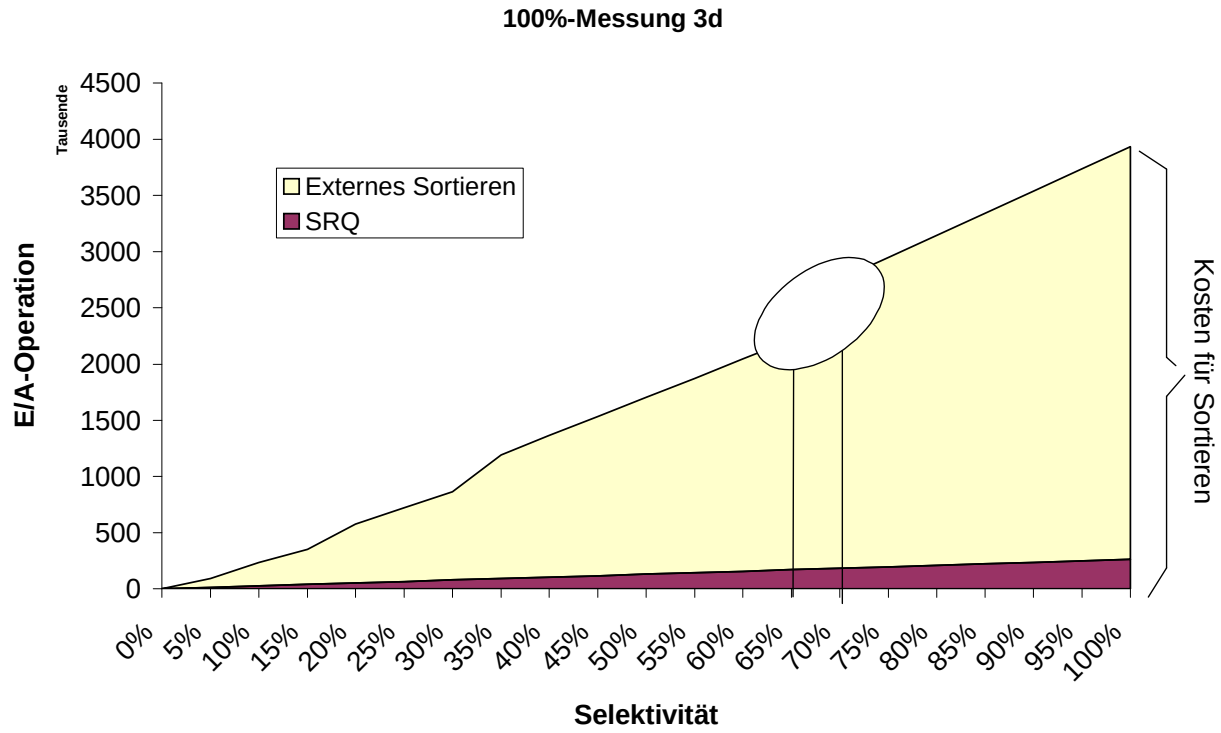


Abbildung 4-17: Externes Sortieren, SRQ

Abbildung 4-17 zeigt die Kosten für das externe Sortieren sowie die Sortierkosten, die beim SRQ-Algorithmus anfallen. Da beim SRQ-Algorithmus die Sortierung mit einem internen Sortierverfahren, in der Prototypimplementierung mit dem Heap-Sort, erzeugt wird, sind die E/A-Kosten gleich der Größe des Ergebnisses in Seiten.

Beim externen Sortieren wird der Initiallauf, d.h. das Erzeugen der Initialläufe, und die Verschmelzungsphase unterschieden. Die E/A-Kosten für die Initialläufe entsprechen den E/A-Kosten des SRQ-Algorithmus. Da in diesem Beispiel ein Verschmelzungsgrad von $m = 2$ angenommen wird, sind die E/A-Kosten in der Verschmelzungsphase relativ hoch. Um die 4 GB Tabelle vollständig zu sortieren, werden bereits 7 Verschmelzungsphasen benötigt, d.h. auf jede Seite wird 14 mal zugegriffen.

Selektivität [%]	SRQ [E/A]	Initiallauf [E/A]	Externes Sortieren [E/A]	Externes Sortieren gesamt	Anzahl der Mischphasen
0	0	0	0	0	0
5	13.107	13.107	78.643	91.750	2
10	26.214	26.214	209.715	235.930	3
15	39.322	39.322	314.573	353.894	4
20	52.429	52.429	524.288	576.717	4
25	65.536	65.536	655.360	720.896	5
30	78.643	78.643	786.432	865.075	5
35	91.750	91.750	1.101.005	1.192.755	5
40	104.858	104.858	1.258.291	1.363.149	5
45	117.965	117.965	1.415.578	1.533.542	5
50	131.072	131.072	1.572.864	1.703.936	6
55	144.179	144.179	1.730.150	1.874.330	6
60	157.286	157.286	1.887.437	2.044.723	6
65	170.394	170.394	2.044.723	2.215.117	6
70	183.501	183.501	2.569.011	2.752.512	6
75	196.608	196.608	2.752.512	2.949.120	6
80	209.715	209.715	2.936.013	3.145.728	6
85	222.822	222.822	3.119.514	3.342.336	6
90	235.930	235.930	3.303.014	3.538.944	6
95	249.037	249.037	3.486.515	3.735.552	6
100	262.144	262.144	3.670.016	3.932.160	7

Tabelle 4-6: E/A-Kosten externes Sortieren, SRQ-Algorithmus

Tabelle 4-6 zeigt die jeweiligen E/A-Kosten für das externe Sortieren und den SRQ-Algorithmus. Es sind die jeweiligen logischen Seiten aufgeführt.

Selektionsphase und Sortierphase

Kombiniert man die Selektionsphase mit der Sortierphase, erhält man den Umfang des SRQ-Operators, der das sortierte Lesen verwirklicht.

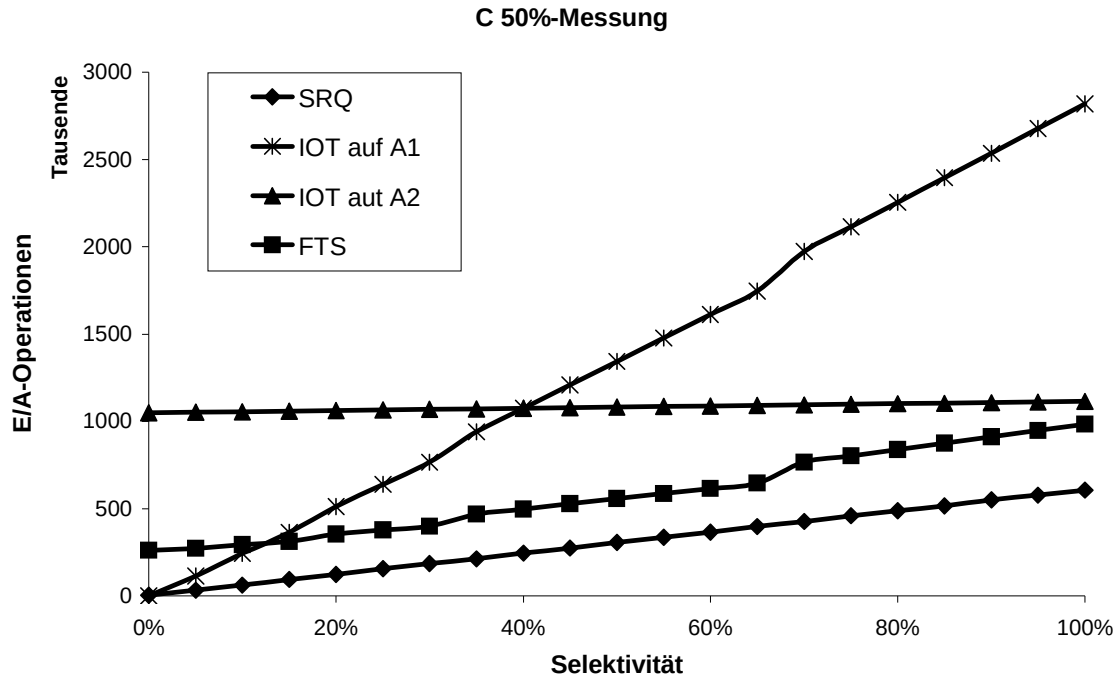


Abbildung 4-18: Bereichsanfrage mit Sortierung auf einem 3-dimensionalen Basisraum mit verschiedenen Zugriffspfaden

Abbildung 4-18 zeigt eine Simulation der E/A-Kosten einer 50%-Messung auf einen 3-dimensionalen Basisraum Ω. Für die Simulation wurde das Kostenmodell aus Kapitel 4.4.4.1 verwendet. Bei der Simulation wird die Selektivität der Dimension A1 von 0% bis 100% variiert. Die Selektivität von Dimension A2 und A3 wird auf 50% festgesetzt. Das Ergebnis soll nach Dimension A2 sortiert ausgegeben werden.

Die SRQ-Algorithmus ist allen Konkurrenten überlegen. Da die Sortierung der Scheiben intern ausgeführt und keine zusätzliche E/A-Operationen benötigt werden, sind die E/A-Kosten des SRQ-Algorithmus linear zur Selektivität aller Dimensionen und somit linear zu der Ergebnismenge. Jede Seite muss genau ein Mal gelesen und geschrieben werden. Die Kostenfunktion für den IOT hat eine Komplexität von O (S_i P lg P_A). Da der IOT auf A1 nur die Selektivität auf Dimension A1 ausnutzen kann, werden S_i P Seiten geladen. Hierzu kommt noch der logarithmische Faktor, das externe Sortieren. Deutlich kann man die Sprünge in den E/A-Kosten erkennen, wie z.B. bei einer Selektivität von ca. 70%, die durch einen zusätzlichen Mischdurchgang verursacht werden. Auch der FTS hat eine E/A-Komplexität von O(P lg P_A). Da der FTS den Prefetchfaktor C voll ausnutzen kann, ist der FTS in Verbindung mit dem externen Sortieren dem IOT auf A1 bei einer Selektivität von ca. 12% auf Dimension A1 überlegen. Der IOT auf A2 zeigt ein konstantes E/A-Verhalten, da er zum einen die Selektivität auf A1 nicht ausnutzen kann, zum anderen jedoch bereits nach A2 sortiert ist. Somit entstehen keine weiteren E/A-Kosten für die Sortierung.

4.4.4.2.2 Erste Antwortzeit $t_{Ant\text{-}1}$

Die erste Antwortzeit $t_{Ant\text{-}1}$, d.h. die Zeit, die benötigt wird, bis das erste Teilergebnis bereitgestellt werden kann, ist sehr entscheidend für die gesamte Leistung bei der Anfrageverarbeitung, da es einen großen Einfluss auf die Parallelisierungsmöglichkeit hat.

Bei der Parallelisierung in der Anfrageverarbeitung unterscheidet man üblicherweise zwischen Parallelisierung zwischen Anfragen (inter query parallelism), Parallelisierung zwischen Operatoren (interoperation parallelism) und Parallelisierung innerhalb eines Operators (intraoperation parallelism) [YuM98]. In der Parallelisierung zwischen Anfragen werden die einzelnen Anfragen auf verschiedene Rechnerkerne verteilt, so dass der Durchsatz erhöht wird. In der Parallelisierung zwischen Operatoren werden einzelne Operatoren einer Anfrage auf verschiedene Rechnerkerne verteilt und parallel abgearbeitet. Wird ein Operator selbst noch unterteilt, spricht man von Parallelisierung eines Operators.

Es gibt hierbei drei Eigenschaften um Parallelisierung zu erreichen, Unabhängigkeit der Operatoren, Pipelining und Partitionierung.

- *Unabhängigkeit der Operatoren*: Bei Operatorbäumen einer Anfrage sind manche Operatoren von einander unabhängig und können demzufolge parallel abgearbeitet werden. Hat man z.B. eine Verbundoperation von vier Relationen R_1 ,R_2 ,R_3 ,R_4, kann man z.B. zunächst R_1, R_2 und R_3, R_4 parallel abarbeiten und dann die Ergebnisse kombinieren.

- *Pipelining*: Das Ergebnis eines Operators ist oft die Eingabe eines anderen Operators im Operatorbaum. Kann der zweite Operator erst mit seiner Arbeit beginnen, bis der erste Operator vollständig abgearbeitet worden ist, spricht man von einer *sequenziellen Verarbeitung*. Kann jedoch der zweite Operator bereits mit Teilergebnissen des ersten Operators weiterarbeiten, handelt es sich um *Pipelining*.

Da üblicherweise mehrere Operatoren aufeinander aufbauen, können z.B. durch Pipelining die ersten Teilergebnisse bereits weiter verarbeitet werden. Hierdurch wird der temporäre Speicherbedarf reduziert, so dass Zwischenergebnisse teilweise nicht explizit gespeichert werden müssen. Eine häufig zur Optimierung verwendete Heuristik zielt darauf ab, die Größe der Zwischenergebnisse zu minimieren [Mit95].

- *Partitionierung*: Die Partitionierung basiert auf dem Prinzip „Teile und Löse“ (Divide and Conquer). Hierbei wird die Eingabe in Teilmengen aufgeteilt und für jede Teilmenge das Ergebnis parallel berechnet, danach werden die Teilergebnisse wieder zusammengefügt.

Das sortierte Lesen erfüllt hierbei die beiden zuletzt genannten Eigenschaften.

Da das „externe Sortieren“ die Vorsortierung nicht ausnutzt, entstehen sortierte Läufe einer Relation R, die bezüglich der globalen Sortierung von R nicht sortiert sind, d.h. für zwei Läufe $(\langle L_1\rangle,\leq)$, $(\langle L_2\rangle,\leq) \subset R$ gilt *nicht*:

$$\left(\forall x \in L_1, y \in L_2 : x < y\right) \vee \left(\forall x \in L_1, y \in L_2 : x > y\right) \qquad \text{Gl: 4-74}$$

Somit ist es nicht möglich *Pipelining* einzusetzen. Die Bedingung Gl: 4-74 ist jedoch für das sortierte Lesen erfüllt. Da zusätzlich die Scheiben (Läufe) in aufsteigender bzw. absteigender Reihenfolge erzeugt werden (siehe Satz 4-2), produziert das sortierte Lesen kontinuierlich Teilergebnisse, die bereits weiterverarbeitet werden können.

Die Antwort Zeit für das sortierte Lesen ist von drei Faktoren abhängig:

- Anzahl der Dimensionen d
- Anzahl der Seiten P
- Selektivität $S_{\neq k}$ ohne die Sortier-Dimension k

Für das sortierte Lesen ergibt sich somit eine Antwortzeit $t_{Ant\text{-}1}$ wie folgt. Nach Gleichung Gl: 4-54 schneiden das Intervall $[y_i,x_i]$ in Dimension i $n_i(d,P,y_i,x_i)$ Scheiben. Da die Dimensionen orthogonal zu einander sind, ergibt sich für die Anzahl der Regionen $M_{ub\text{-}s}$, die in einer Scheibe der Sortier-Dimension k enthalten sind:

$$M_{ub-s} = \prod_{i=1,i\neq k}^{d} n_i(d,P,y_i,x_i) \approx P \prod_{i=1,i\neq k}^{1} S_i \qquad \text{Gl: 4-75}$$

$M_{ub\text{-}s}$ Seiten müssen somit vom Sekundärspeicher geladen und dann intern sortiert werden, bis das erste Teilergebnis vom SRQ-Operator zurückgegeben werden kann.

4.4.4.3 Speicherplatzbedarf

Das hier betrachtete Komplexitätsmaß sei nun der Bedarf am Arbeitsspeicher M_c, der benötigt wird, um die Anfrage Q nach Attribut j zu sortieren. Es wird hier wieder von gleichverteilten Daten ausgegangen. Da das sortierte Lesen den Anfragebereich in Scheiben aufteilt, die nacheinander verarbeitet werden, ist die Cachegröße für das sortierte Lesen abhängig zum einen von der Scheibenbreite und zum anderen von der Selektivität in den Nicht-Sortierdimensionen. Geht man davon aus, dass keine Restriktionen auf den Dimension existieren, ergibt sich für die Cachegröße folgender Satz:

Satz 4-4:

Gegeben sei ein d-dimensionaler UB-Baum mit P Seiten. Dann benötigt das sortierte Lesen einen Cache der Größe:

$$cache_{sortierte\_Lesen}(P,d) = \sqrt[d]{P^{d-1}}$$

Beweis:

$$cache_{sortiertes\_Lesen}(P,d) = \frac{P}{2^{recursive\_splits_{\dim}}} = \frac{P}{2^{\frac{\log_2 P}{d}}} = \sqrt[d]{P^{d-1}}$$

q.e.d.

Abbildung 4-19 zeigt die verschiedenen Cachegrößen für das sortierte Lesen. Hierbei sind Cachegrößen für einen 2- bis 5-dimensionalen UB-Baum dargestellt. Ist der IOT auf die Sortierdimension angelegt worden, benötigt der IOT keinen Cache.

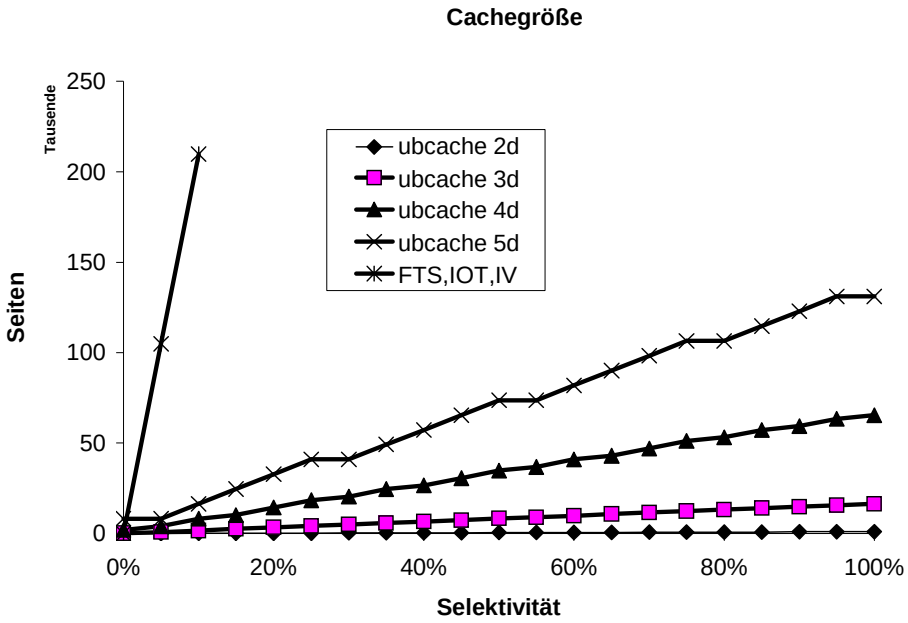


Abbildung 4-19: Cachegröße

4.4.4.4 CPU-Komplexität

Das hier betrachtete Komplexitätsmaß sei nun die Anzahl der Vergleichoperationen, die benötigt wird, um die Anfrage *Q* nach Attribut *j* zu sortieren. *m* sei die Anzahl der Tupel, die in dem Anfragebereich liegen.

Für Sortieren durch Verschmelzen erhält man eine CPU-Komplexität von

$$c_{CPU-sort} = O(m \log m) \qquad \text{Gl: 4-76}$$

Die CPU-Kosten für das sortierte Lesen bestehen im Wesentlichen aus zwei Teilen.

- CPU-Kosten der Bereichsanfrage, $c_{CPU\text{-}RQ}$
- CPU-Kosten für die Sortierung, $c_{CPU\text{-}sort\text{-}RQ}$

Nach [Mar99] ist die CPU-Komplexität des Bereichsanfrage-Algorithmus linear abhängig von der Anzahl der Regionen, die den Anfragebereich schneiden. Dies entspricht nach dem Kostenmodell aus Abschnitt 4.4.1 der Selektivität des Anfragebereichs *Q* (siehe Gl: 4-2, Gl: 3-28 und Gl: 4-70) und der Anzahl der Bits *b*, die eine Z-Adresse repräsentieren.

$$c_{CPU-RQ} = O(b \cdot P \cdot \prod_{i=0}^{d-1} S_i) = O(b \cdot \frac{m}{F_{Füllungsgrad} \cdot \kappa} \prod_{i=0}^{d-1} S_i) \qquad \text{Gl: 4-77}$$

Für einen bestimmten UB-Baum ist somit der Bereichsanfragealgorithmus linear zu der Anzahl der Regionen, die den Anfragebereich schneiden.

$$c_{CPU-RQ} = O\left(\frac{m}{F_{Füllungsgrad} \cdot \kappa} \prod_{i=0}^{d-1} S_i \right) = O(m) \qquad \text{Gl: 4-78}$$

Das sortierte Lesen partitioniert die Datenmenge für die Sortierphase in Scheiben, die jeweils einzeln sortiert werden. n_k ist die Anzahl der Scheiben, die in der Sortierdimension k erzeugt werden, um den Anfragebereich abzuarbeiten (siehe Gl: 4-54). Somit ergit sich für das Sortieren einer Scheibe eine CPU-Komplexität von

$$c_{CPU-sort-RQ} = O\left(\frac{m}{n_k} \log \frac{m}{n_k} \right) \qquad \text{Gl: 4-79}$$

Diese CPU-Kosten entstehen n_k Mal und man erhält:

$$c_{CPU-sort-RQ} = O\left(m \log \frac{m}{n_k} \right) \qquad \text{Gl: 4-80}$$

Werden die Selektivitäten der Nicht-Sortierdimensionen konstant gehalten, ist somit die CPU-Komplexität des sortierten Lesens linear zur Ergebnisgröße m. Das sortierte Lesen gehört somit zu den adaptiven Sortieralgorithmen [EstW92] und ist somit dem extern Sortieren überlegen.

4.4.5 Leistungsmessungen

In diesem Abschnitt werden einige Leistungsmessungen präsentiert, die den Bereichsanfrage-Algorithmus in Kombination mit externem Sortieren dem SRQ-Algorithmus (siehe Kapitel 4.4) gegenüberstellen.

Die Messungen wurden auf einer Sun Ultra 1 mit 143 MHz von Sun Microsystems durchgeführt. Der Arbeitsspeicher M betrug 64 MB. Als Sekundärspeicher wurde eine 18.2 GB IBM Ultrastar 18XP Festplatte eingesetzt. Die durchschnittliche Zugriffszeit ist mit 7,5 ms angegeben.

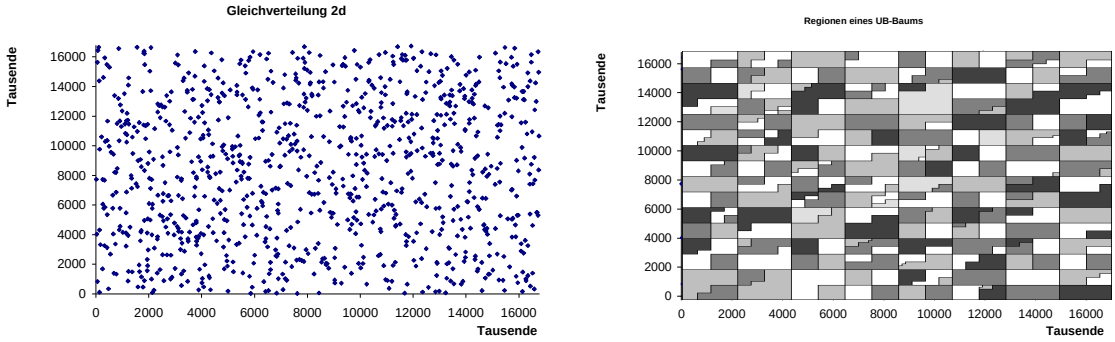


Abbildung 4-20: UB-Baumpartitionierung für gleichverteilte Daten

Für die Messungen wurde eine 2-dimensionale, 1,38 GB große Datenbank mit TransBase aufgebaut (siehe Tabelle 4-7) mit einer Seitengröße von 2 KB.

Name	Dimensionen	Wertebereich	Verteilung	Anzahl der Tupel	Anzahl der Seiten	Datenbank-größe [GB]
DB2d2Gh	2	$[0,2^{24}-1]$	gleichverteilt	2.400.000	682.908	1.38

Tabelle 4-7

Abbildung 4-21 zeigt eine 50%-Messung der Form:

```
SELECT x1,x2 FROM table
WHERE x1 BETWEEN a AND b AND
      x2 BETWEEN c AND d
ORDER BY x1
```

Die Selektivität der Sortier-Dimension x_1 wurde auf 20%, 40% und 60% festgelegt. Die Dimension x_2 jeweils auf 50%. Für die jeweilige Selektivität wurde eine Messung mit dem SRQ-Algorithmus und mit dem RQ-Algorithmus in Verbindung mit Sortieren durchgeführt.

Die Ergebnismenge P_E umfasst bei der 20%-Messung 68526 Seiten. Bei einer Seitengröße von 2 KB ergibt das eine Ergebnisgröße von ca. 136 MB. Für die 60%-Messung ergibt sich eine Ergebnismenge P_E von ca. 400 MB. Somit gilt $P_E > M$, das externe Sortieren muss eingesetzt werden. Nach Gl: 4-12 ist das externe Sortieren linear genau dann, wenn der logarithmische Faktor $\log_m \frac{P_E}{M}$ kleiner gleich 1 ist. Setzen wir nach Gl: 4-10 den Mischgrad m auf $m = \left\lfloor \frac{M}{2C} - 2 \right\rfloor = \left\lfloor \frac{32768}{2 \cdot 8} - 2 \right\rfloor = 2046$, so ist der logarithmische Faktor für die drei Messungen kleiner 1. Die E/A-Kosten $c_{E/A\text{-}ub\text{-}sort}$ sind linear (siehe Kostenmodell Tabelle 4-4). Dies entspricht den Messergebnissen aus Tabelle 4-8. Die Bereichsanfrage in Kombination mit Sortieren (RQ_Sort) wächst annähernd linear. Der logarithmische Faktor wird jedoch durch die CPU-Kosten verursacht.

Operator	Selek-tivität $x1$ [%]	Ergebnis-Tupel	Scheiben	gelesene Seiten	mehrmals gelesene Seiten [%]	Antwort-zeit [s]	Erstes Teilergebnis[s]
SRQ	20	239643	13	68760	0,34	1240	77
RQ_Sort	20	239643	1	68526	0	1504	1321
SRQ	40	479889	26	137340	0,37	2460	78
RQ_ Sort	40	479889	1	136825	0	3024	2741
SRQ	60	720050	39	206107	0,38	3866	77
RQ _Sort	60	720050	1	205317	0	4801	4382

Tabelle 4-8

Betrachtet man die Verarbeitungszeit pro Seite beim *RQ-Sort* wächst sie von 21,7 ms auf 23,5 ms bei einer Vergrößerung der Ergebnismenge P_E um Faktor 3.

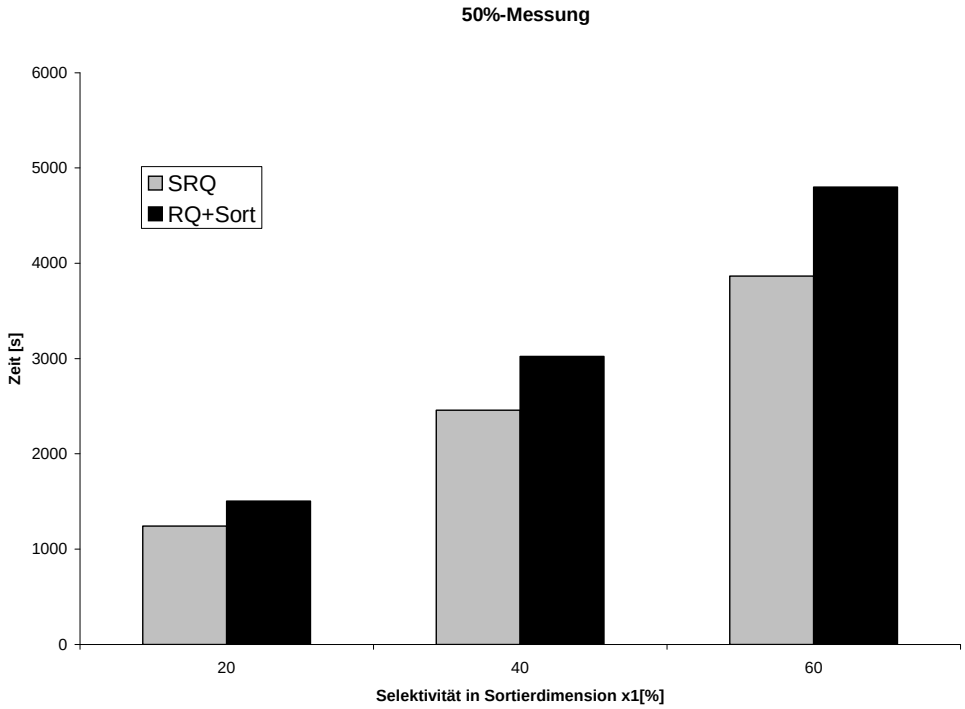


Abbildung 4-21 TransBase SRQ und RQ + Sortieren

Die E/A-Kosten $c_{ub\text{-}SRQ}$ für den SRQ-Algorithmus (siehe Kostenmodell Tabelle 4-4) ergeben sich aus den Lesekosten des UB-Baums und aus dem Schreiben der Ergebnismenge P_E. Eine Scheibe des SRQ-Algorithmus besteht aus ca. 5290 Seiten. Bei einer Seitengröße von 2 KB ergibt sich somit ein Cachebedarf von 10,6 MB. Eine Scheibe kann somit vollständig im Arbeitsspeicher durch einen Heapsort sortiert werden. Die E/A-Kosten für das Lesen aus dem UB-Baum sind höher als das Schreiben, da der *Prefetchfaktor* C nicht vollständig ausgenutzt werden kann. Wie beim RQ-Sort-Algorithmus wachsen die E/A-Kosten linear. Die CPU-Kosten $c_{CPU\text{-}SRQ}$ ergeben sich aus der Summe der Sortier-Kosten der einzelnen Scheiben. Da für diese Messung gleichverteilte Daten benutzt wurden, kann man davon ausgehen, dass in jeder Scheibe gleich viele Tupel liegen und somit die Kosten $c_{CPU\text{-}Scheibe}$ konstant sind. Da die Anzahl der Scheiben linear von der Anzahl der Ergebnistupel n abhängt (siehe Tabelle 4-8), sind die CPU-Kosten $c_{CPU\text{-}SRQ}$ auch linear. Betrachtet man wieder das Verhältnis Verarbeitungszeit pro Seite, erhält man für die drei Messungen eine Verarbeitungszeit für eine Ergebnisseite von ca. 18 ms. Dies entspricht somit dem Kostenmodell.

Vergleicht man die E/A-Kosten beider Algorithmen, so ist der SRQ-Algorithmus dem RQ-Sort-Algorithmus überlegen. Die E/A-Kosten für den SRQ-Algorithmus liegen bei 80 bis 82% der E/A-Kosten des RQ-Algorithmus (siehe Tabelle 4-8) bei diesem 2-dimensionalen UB-Baum. Eine Reduzierung der E/A-Kosten auf 50% ist nur dann möglich, wenn der Prefetchfaktor *C* gleich 1 ist. Sei $e = \frac{c_{E/A-SRQ}}{c_{E/A-ub-sort}}$, dann ergibt sich für den Prefetchfaktor:

$$
\begin{aligned}
& c_{E/A-SRQ} && = && e \cdot c_{E/A-ub-sort} \\
\Leftrightarrow \quad & c_{E/A}\left(P_E + \frac{P_E}{C}\right) && = && e \cdot c_{E/A}\left(P_E + \frac{3P_E}{C}\right) \\
\Leftrightarrow \quad & c_{E/A}P_E + \frac{c_{E/A}P_E}{C} && = && e \cdot c_{E/A} \cdot P_E + \frac{3 \cdot e \cdot c_{E/A} \cdot P_E}{C} \\
\Leftrightarrow \quad & c_{E/A} \cdot P_E(1-e) && = && \frac{c_{E/A}P_E}{C}(3e-1) \\
\Leftrightarrow \quad & C && = && \frac{c_{E/A}P_E(3e-1)}{c_{E/A}P_E(1-e)} \\
\Leftrightarrow \quad & C && = && \frac{(3e-1)}{(1-e)}
\end{aligned}
$$

Gl: 4-81

Für die Messung ergibt sich für *e* ein Wert von ungefähr 0,81. Damit ergibt sich für den Prefetchfaktor *C*:

$$C = \frac{3 \cdot 0{,}81 - 1}{1 - 0{,}81} = \frac{1{,}43}{0{,}19} \approx 7{,}52$$

Gl: 4-82

Ein weiterer wesentlicher Vorteil des SQR und des Tetris-Algorithmus (siehe Kapitel 5) liegt in den *E/A*-Kosten $c_{E/A\text{-SQR-ant1}}$ für die Bereitstellung des ersten Teilergebnises.

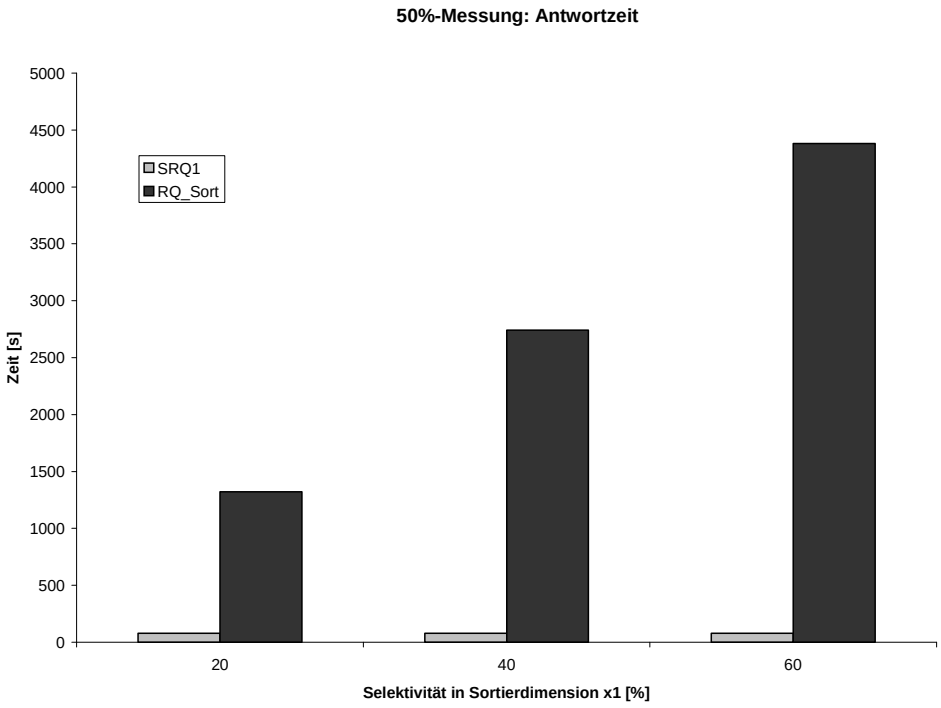


Abbildung 4-22

Dies ist in Abbildung 4-22 dargestellt. Die Kosten $c_{E/A\text{-SQR-ant1}}$ für das erste Teilergebnis werden durch die Kosten der Verarbeitung der ersten Scheibe bestimmt. Da die Selektivität in x_2 bei den Messungen konstant bleibt, wird die Anzahl der Seiten, die eine Scheibe schneiden, nicht verändert. Sie beträgt bei allen drei Messungen ca. 5282 Seiten pro Scheibe (siehe Tabelle 4-9). Somit sind die Kosten für das erste Teilergebnis $c_{E/A\text{-SQR-ant1}}$ unabhängig von der Selektivität in der Sortierdimension x_1. Die Antwortzeit des ersten Teilergebnises mit dem SRQ-Algorithmus liegt bei allen drei Messungen bei 77-78 s. Demgegenüber ist die Antwortzeit des *RQ-Sort*-Algorithmus bei einer Selektivität auf Dimension x_1 von 20% bei 13021 s, bei einer Selektivität auf Dimension x_1 von 40 % bei 2741s und bei einer Selektivität auf Dimension x_1 von 60 % bereits bei 4382 s.

%-Messung	SRQ Seite / Scheibe	Erstes Teilergebnis SRQ [s]	Erstes Teilergebnis RQ_Sort [s]	Faktor RQ_Sort / SRQ
20	5289	77	1.321	17,16
40	5282	78	2.741	35,14
60	5285	77	4.382	56,91

Tabelle 4-9

Der Grund liegt darin, dass es sich beim Sortieren um eine blockende Operation handelt, d.h. es kann erst das erste Tupel der Ergebnismenge *E* an den Aufrufer zurückgegeben werden, wenn das Ergebnis vollständig sortiert ist.

Das Ergebnis kann erst vollständig sortiert werden, wenn die letzte Seite, die zum Anfragebereich *Q* gehört, aus dem UB-Baum geladen worden ist und das externe Sortieren mit der letzen Mischphase beginnt. Somit kann das erste Teilergebnis durch den SRQ-Algorithmus bei einer Selektivität auf Dimension x_1 von 20% 17 Mal schneller und bei einer Selektivität auf Dimension x_1 von 60% sogar 56 Mal schneller bereitgestellt werden als mit dem RQ-Sort-Algorithmus.

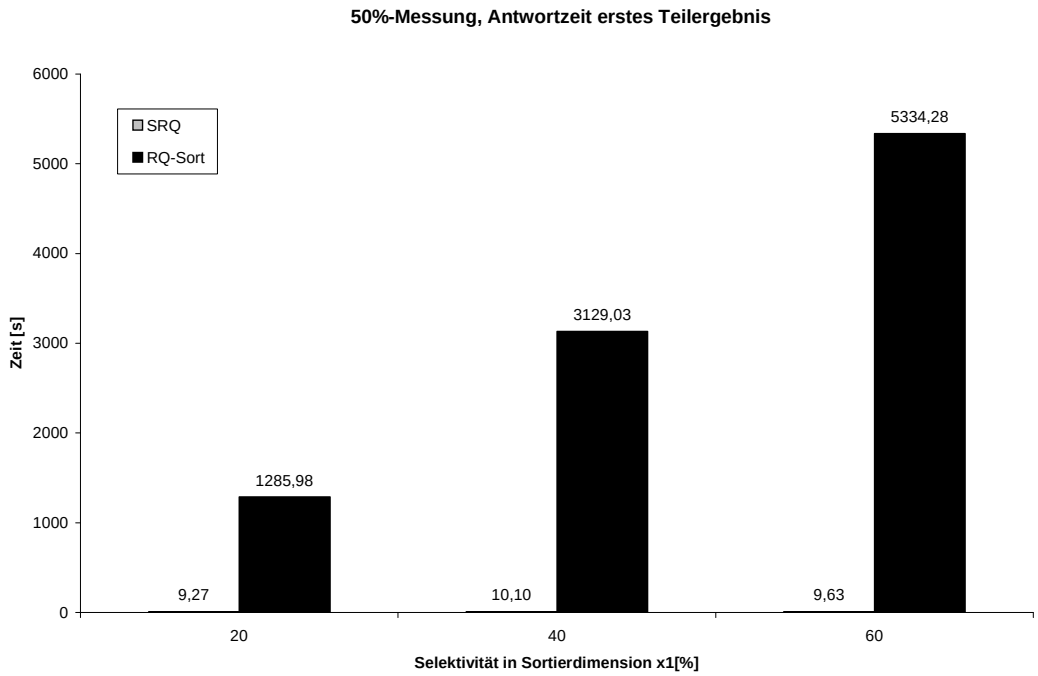


Abbildung 4-23

Die Antwortzeit für das erste Teilergebnis kann durch den SRQ-Algorithmus soweit optimiert werden, dass die minimale Scheibenbreite gelesen wird. Die minimale Scheibenbreite wird durch die vollständige Teilungstiefe pro Dimension $l_{\downarrow/d}$ bestimmt.

Durch die minimale Schreibenbreite in Sortierrichtung x_1 müssen weniger Seiten in den Arbeitsspeicher geladen und sortiert werden, um das erste Teilergebnis zu erzeugen. Somit werden die E/A- und CPU-Kosten auf das Minimum reduziert. Abbildung 4-23 zeigt die Antwortzeiten des SRQ und RQ-Sort für eine 50%-Messung. Für den SRQ wurde die minimale Seitenbreite benutzt. Das erste Teilergebnis liefert der SRQ-Algorithmus nach 9 bis 10 Sekunden bei einer Selektivität auf Dimension x_1 von 20%, 40% und 60%.

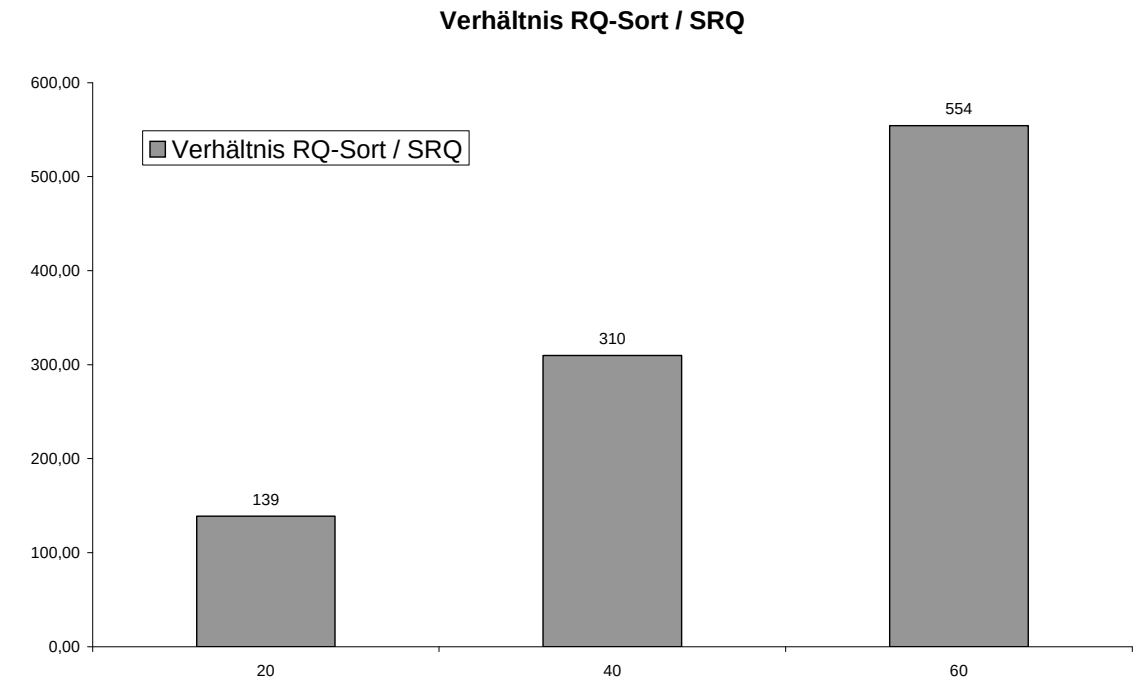


Abbildung 4-24

Der RQ-Sort-Algorithmus benötigt bei einer Selektivität auf Dimension x_1 von 60% 5334 Sekunden. Somit ist der SRQ-Algorithmus 554 Mal schneller (siehe Abbildung 4-24).

Operator	%-Messung	Ergebnis-tupel	Anzahl der Scheiben	gelesene Seiten	mehrmals gelesene Seiten %	Antwortzeit [s]	Erstes Teilergebnis [s]	RQ_Sort / SRQ
SRQ	20	239.643	205	130.461	90	1.622	9	139
RQ_Sort	20	239.643	1	68.526	0	1.329	1.285	
SRQ	40	479.889	410	261.191	91	4.551	10	310
RQ_Sort	40	479.889	1	136.825	0	3.253	3.129	
SRQ	60	720.050	615	391.695	91	8.795	9	554
RQ_Sort	60	720.050	1	205.317	0	5.487	5.334	

Tabelle 4-10

In Tabelle 4-10 sind die einzelnen Messwerte zusammengefasst. Betrachtet man die Antwortzeit für die gesamte Ergebnismenge *E* ist der RQ-Sort-Algorithmus dem SRQ-Algorithmus überlegen (siehe Abbildung 4-25). Durch das Fransenphänomen [FriM97], [Mer99] kann eine Region mehrere Scheiben schneiden, so dass sie mehrmals geladen wird.

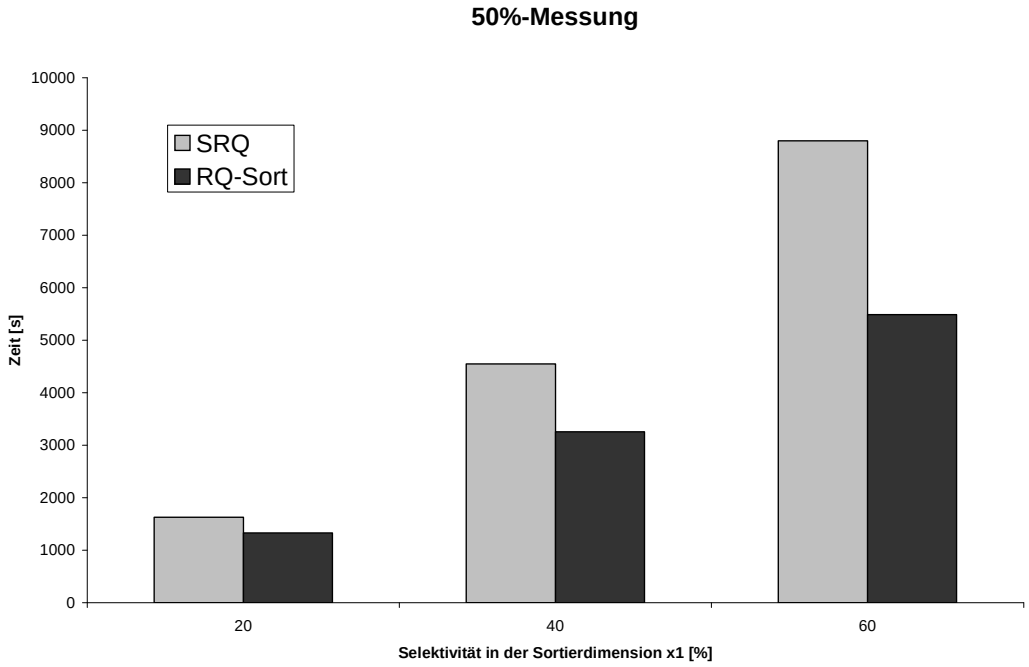


Abbildung 4-25:

Bei dieser Messung lädt der SRQ-Algorithmus zwischen 90 und 91% mehr Seiten als der RQ-Algorithmus, der jede Seite, die in den Anfragebereich *Q* fällt, genau einmal liest. Dies führt zu einer wesentlichen Erhöhung der E/A-Kosten. Da beim Lesen der Prefetchfaktor *C* nicht optimal ausgenutzt werden kann, sind die E/A-Kosten pro Seite höher als beim RQ-Sort-Algorithmus.

50%-Messen	20%		40%		60	
Anzahl der Zugriffe	Seite	%	Seite	%	Seite	%
6	1	0	2	0	2	0
5	3	0	6	0	9	0
4	1.431	2	2.805	2	4.211	2
3	8.302	12	16.705	12	25.121	12
2	41.019	60	82.505	60	123.456	60
1	17.770	26	34.804	25	52.519	26

Tabelle 4-11: Anzahl der Zugriffe auf eine Seite in Seiten und %

Tabelle 4-11 zeigt die Anzahl der Zugriffe auf eine Seite durch den SRQ-Algorithmus. Es werden z.B. 60% der Seiten, die den Anfragebereich *Q* schneiden, genau zwei Mal gelesen, d.h. sie schneiden zwei Scheiben.

4.4.6 TPC-H Benchmark

In diesem Kapitel werden Leistungsmessungen für den Verbundoperator vorgestellt. Hierzu wurden die Daten des *TPC-H* Benchmark herangezogen, der im folgenden Abschnitt kurz erläutert wird.

Der TPC-Benchmark H (TPC-H) ist ein Benchmark für *Decision-Support-Systeme*. Er besteht aus einer Menge von geschäftsorientierten *ad-hoc-Anfragen* sowie konkurrierenden Updates Anweisungen. Die Anfragen werden auf künstlichen Daten ausgeführt. Die Anfragen und die Datenverteilung haben eine große Relevanz in einem großen Bereich der Industrie. Dieser Benchmark betrachtet große Datenmengen, auf denen komplexe Anfragen zu kritischen Geschäftsprozessen ausgeführt werden.

Schema

Das TPC-H Datenbankschema ist ein 3 dimensionales Schneeflocken-Schema mit der Lineitem Tabelle als zentrale Faktentabelle. Auf die Faktentabellen können drei Sichten genutzt werden, die Dimension *Kunde*, *Lieferant* und *Produkt*. Die Dimension *Kunde* besteht aus den Hierarchieobjekten *Order*, *Customer*, *Nation* und *Region*, die Dimension *Lieferant* aus den Hierarchieobjekten *Partsupp*, *Supplier*, *Nation* und *Region*, die Dimension *Produkt* aus *Partsupp* und *Part*. Jedes Hierarchieobjekt besteht aus einer Tabelle, die zusätzliche *Attribute* enthält. Eine genaue Beschreibung der Attribute ist in [TCP01]zu finden.

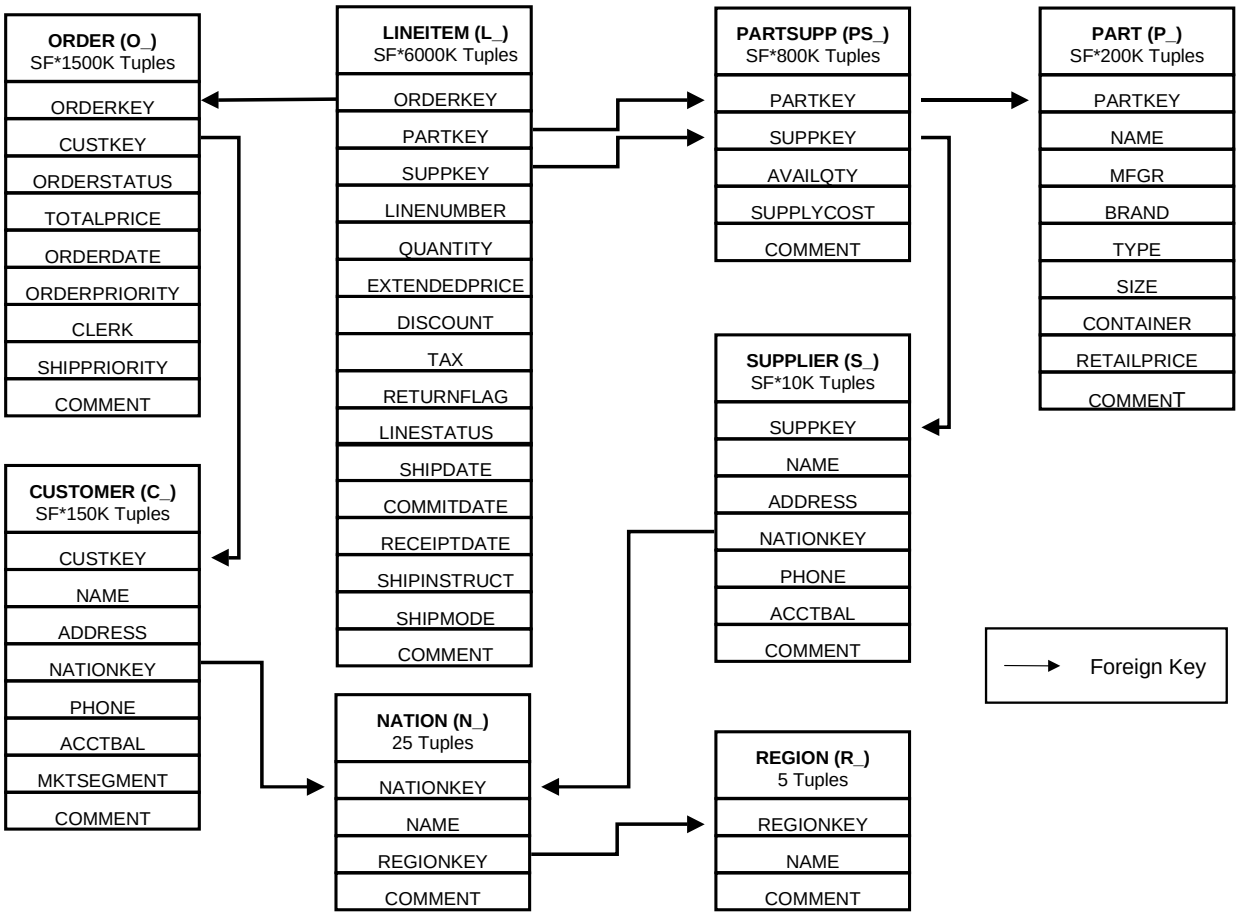


Abbildung 4-26: TPC-Schema

4.4.6.1 Messung

Für die Leistungsmessungen wurde die Faktentabelle *Lineitem* als UB-Baum organisiert. Die Leistungsmessungen wurden mit der Prototypimplementierung (siehe Kapitel 8) durchgeführt. Als Datenbanksystem wurde Oracle 8i verwendet. Für den Vergleich wurden ein „Index Orgnized Table“ (IOT), der die Tabelle als einen clusternden B*-Baum speichert, und eine „Full Table Scan“ (FTS), der die ganze Tabelle liest, herangezogen. Hierzu wurde eine SUN ULTRA SPARC II mit einen Arbeitsspeicher von 512MB und einer 4 GB Festplatte mit einer durchschnittlichen Positionierungszeit t_π(siehe Definition 3-8) von 8 ms und einer Übertragungszeit t_τ (siehe Definition 3-7) von 0,7 ms. pro Seite verwendet. Für den Vergleich des SRQ-Algorithmus mit dem IOT und FTS wurde die Query *Q3* und *Q4* des TPC-H Benchmark [TCP01]herangezogen.

4.4.6.2 Verbundoperator und Restriktionen

Die Anfrage *Q3* des TPC-H Benchmark besteht aus Einschränkungen und einem Verbundoperator, der drei Relationen verbindet.

```
SELECT L_ORDERKEY, ,O_ORDERDATE, O_SHIPPRIORITY,
       SUM(L_EXTENDEDPRICE*(1- L_DISCOUNT)) AS REVENUE
FROM CUSTOMER, ORDER, LINEITEM
WHERE
       C_MKTSEGMENT = 'FOOD' AND
       C_CUSTKEY = O_CUSTKEY AND
       L_ORDERKEY = O_ORDERKEY AND
       O_ORDERDATE < DATE 1.5.98 AND
       L_SHIPDATE > DATE 1.6.98
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
ORDER BY REVENUE DESC, O_ORDERDATE
```

Abbildung 4-27: Query *Q*3 des TPC-H Benchmark

Der Standard-Operatorbaum sowie der Operatorbaum mit sortiertem Lesen ist in Abbildung 4-28 abgebildet. Es werden die in Kapitel 3.1 einführten Symbole für die Opertoren der relationalen Algebra verwendet. Zusätzlich bezeichnet ω den Sortieroperator und γ den Gruppierungsoperator. $M_{<Schlüssel>}$ bezeichnet den Verschmelzungs-Operator, der zwei sortierte Tupelströme, die nach <Schlüssel> sortiert sind, miteinander verschmelzt.

Die Query ohne Sortiertes Lesen (siehe Abbildung 4-28) wird wie folgt verarbeitet: Zunächst werden die Restriktionen auf die einzelnen Tabellen ausgeführt. Danach wird ein Hash-Join oder Sorted-Merge-Join auf das Zwischenergebnis angewandt. Wir gehen hier nicht näher auf die Problematik ein, ob ein Sort-Merge-Join oder ein Hash-Join die bessere Wahl ist.

Die Reihenfolge der Joins ist durch die Größe der einzelnen Tabellen bestimmt. Da die Lineitem Tabelle 4 mal größer als die ORDER Tabelle und sogar 40 mal größer als die CUSTOMER Tabelle ist, wird sie erst im zweiten Join verarbeitet. Das Zwischenergebnis des zweiten Join wird dann gruppiert und sortiert.

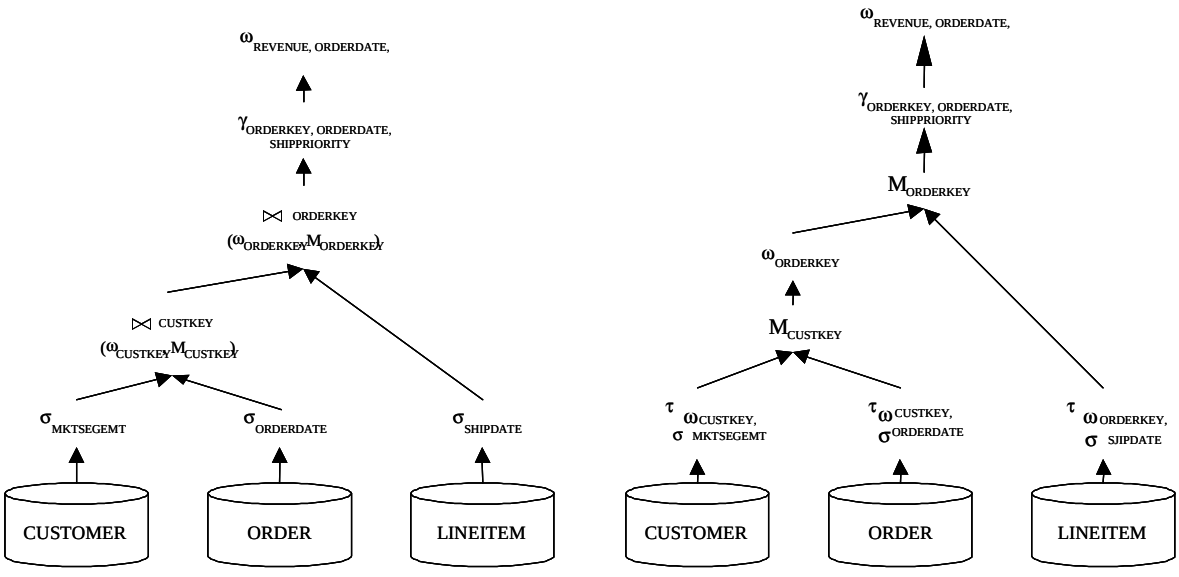


Abbildung 4-28: Operatorbaum für Query Q3 des TPC-H Benchmark

Werden die Tabellen als UB-Baum organisiert, sind die jeweiligen Attribute der Tabellen CUSTOMER(CUSTKEY, MKTSEGMENT), ORDER(ORDERKEY, CUSTKEY, ORDERDATE) und LINEITEM(SHIPDATE, ORDERKEY) Bestandteil des Schlüssels. Abbildung 4-28.b und Abbildung 4-29 zeigt die Verarbeitung des Operators des sortierten Lesens $\tau_{\sigma,\omega}$, der die Bereichsanfrage und das Sortieren miteinander kombiniert. Hierbei wird die Bereichsanfrage sortiert nach dem Join-Attribut gelesen. Dieser sortierte Strom von Tupeln wird an den Merge -Operator weitergereicht.

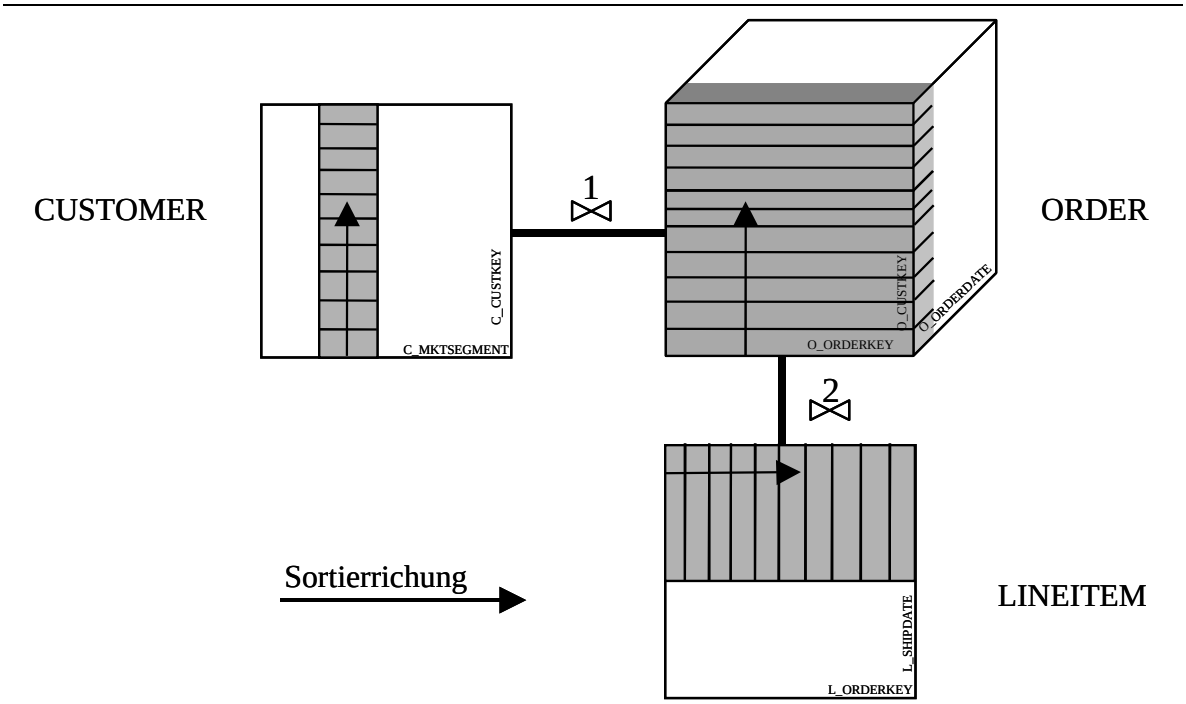


Abbildung 4-29

Für den sortierten Tabellenzugriff in Anfrage Q3 des TPC-H Benchmark werden verschiedene Skalierungsfaktoren der Tabellen betrachtet. Für die Messung wurde der Skalierungsfaktor 0.1 bis 1.0 gewählt. Für die LINEITEM Tabelle ergibt sich bei einem Skalierungsfaktor von 1 eine Tabellengröße von 1302 MB.

Es wird hier nicht näher darauf eingegangen ob ein HASH-JOIN oder eine SORT-MERG-JOIN die bessere Wahl für die Verarbeitung ist [Mer83],[DewKO+84]. Für die Messung wird ein großer Arbeitsspeicher eingesetzt. Da nach [CheHH+91] der Hash-Join und der Sort-Merge-Join in Systemen mit großem Arbeitsspeicher ähnliche Leistungsverhalten aufweisen. wird in dieser hier der Sort-Merge-Join verwendet.

Die LINEITEM Tabelle ist der primäre Flaschenhals der Anfrage Q3. In der Messung wird somit nur noch die Tabelle LINEITEM betrachtet. Für die Messungen wurde die LINEITEM Tabelle in drei Varianten aufgebaut:

- IOT mit SCHIPDATE als Schlüssel
- IOT mit ORDERKEY als Schlüssel
- UB-Baum mit Schlüssel SCHIPDATE, ORDERKEY und CUSTKEY

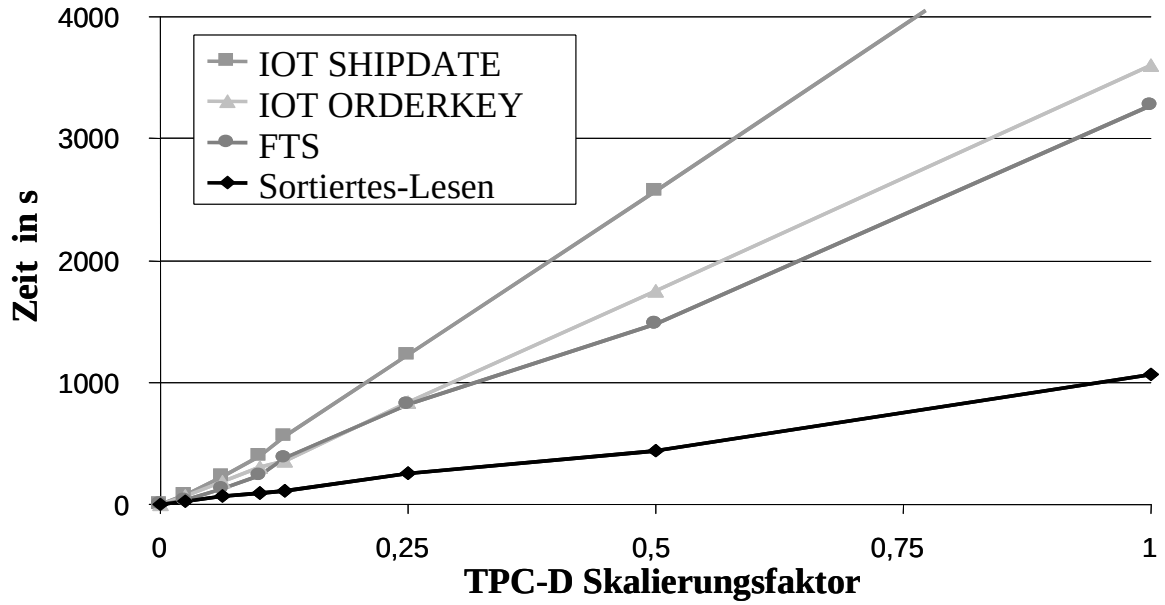


Abbildung 4-30: Antwortzeit Q3

Abbildung 4-30 und Tabelle 4-12 zeigen eindeutig, das das sortierte Lesen das beste Antwortverhalten aufzeigt. Die 50% Restriktion auf SHIPDATE erzeugt keine ausreichende Selektivität auf dem IOT auf SHIPDATE um mit dem UB-Baum zu konkurrieren. Der vorsortierte IOT auf ORDERKEY benötigt keine Sortierung (Merge-Sort) und zeigt somit ein ähnliches Antwortverhalten wie der FTS mit Sortieren (Merge-Sort). Das sortierte Lesen ist 3 mal schneller als der FTS und IOT. Besonders hervorzuheben ist die Antwortzeit des ersten Teilergebnisses des sortierten Lesens, das bereits nach ein paar Sekunden geliefert wird, 2 bis 3 Größenordnungen schneller als der FTS oder IOT.

Tabellengröße	33 MB	81 MB	131MB	163MB	326MB	651MB	1302MB
Skalierungsfaktor	(0.025)	(0.0625)	(0.1)	(0125)	(0.25)	(0.5)	(1)
SRQ t_{Ant-1}	0,3s	0,5s	0,7s	1,1s	1,3s	1,3s	3,3s
SRQ t_{ant}	23.1s	64.4s	92.5s	106.2s	257.5s	441.2s	1062.2s
IOT tant ORDERKEY	64,7s	184,3s	306,7s	356,2s	834,3s	1753,6s	3604,1s
IOT tant SHIPDATE	72.5s	226.9s	401.3s	554..3s	1223.7s	2569.8s	5286.4s
FTS-Sort	34.1s	126.7s	234.0s	381.1s	816.5s	1479.4s	3276.4s
Scheiben	64	128	128	128	256	256	512
SRQ Cache	0.3.MB	0.3MB	0.9MB	1.1MB	1.4MB	2.1MB	2.6MB
IOT /FTS Cache	17MB	40MB	65MB	81MB	183MB	326MB	751MB

Tabelle 4-12

Der IOT on ODERKEY benötigt keine Zwischenspeicher, da die Tabelle bereits in Zielordnung organisiert ist. Das sortierte Lesen benötigt einen relativ geringen temporären Speicher für die Scheiben, da der temporäre Speicher entsprechend der $\sqrt{\text{Tabellengröße}}$ wächst. Bei einer 1306 MB Tabellengröße und einer Restriktion von 50% auf SHIPDATE benötigt das sortierte Lesen nur 2.6 MB. Dem gegenüber wächst der temporäre Speicher für den FTS und IOT linear zur Ergebnismenge *E*. Hierbei gehen wir davon aus, dass die

gesamte Ergebnismenge auf einmal mit einem internen Sortierverfahren, wie z.B. Quick-Sort [Knu98v3] oder Heap-Sort [Knu98v3] sortiert wird. Somit beträgt der temporäre Speicher 751 MB.

4.4.6.3 Verbundoperator, Gruppierung und Restriktionen

Bereichsanfragen, Join und Gruppierung werden effizient durch das sortierte Lesen unterstützt. Im Allgemeinen kann das externe Sortieren vermieden werden, da durch die Ausnutzung der mehrdimensionalen Restriktionen und das Lesen von Scheiben geringe Mengen von Seiten auf einmal gelesen werden müssen. Dies wird an Hand von Anfrage Q4 gezeigt.

```
SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT
FROM ORDER
WHERE  O_ORDERDATE >= DATE '[Date95]' AND
            O_ORDERDATE < DATE '[Date95]' +
            INTERVAL '3' MONTH AND
            EXISTS (   SELECT *
                       FROM LINEITEM
                       WHERE
                            L_ORDERKEY = O_ORDERKEY AND
                            L_COMMITDATE < L_RECEIPTDATE )
GROUP BY O_ORDERPRIORITY
ORDER BY O_ORDERPRIORITY
```

Abbildung 4-31

Es wird angenommen, dass die ORDER-Tabelle (ORDERDATE, ORDERPRIORITY, ORDERKEY) und die LINEITEM (COMMITDATE, RECEIPTDATE, ORDERKEY) als 3-dimensionaler UB-Baum organisiert ist. Die Anfrageverarbeitung der Anfrage Q4, die das sortierte Lesen einsetzt, ist in Abbildung 4-32 dargestellt. Die Anfrage gruppiert die Tupel der ORDER Tabelle. Hierbei werden die Tupel durch das Attribut ORDERDATE auf ein Intervall und zusätzlich noch durch eine Existenzabfrage auf Lineitem eingeschränkt. Um diese Anfrage effizient zu verarbeiten, muss die Tabelle ORDER sortiert bezüglich des ODERKEY Attributs verarbeitet werden. Zusätzlich muss die 3,5% Restriktion auf ORDERDATE berücksichtigt werden. Zur Auswertung der Existenzrestriktion wird die LINEITEM Tabelle sortiert nach ORDERDATE verarbeitet und ein Semi-Join mit der ORDER Tabelle ausgeführt. Durch das sortierte Lesen wird die Restriktion COMMITDATE < RECEIPTDATE effizient ausgewertet. Hierbei wird der Anfragebereich sortiert nach ORDERKEY verarbeitet. Wird jede ORDERDATE-Scheibe in ORDERPRIORTY-Ordnung verarbeitet, können die CPU Operationen reduziert werden, da für die Gruppierung nicht mehr sortiert werden muss.

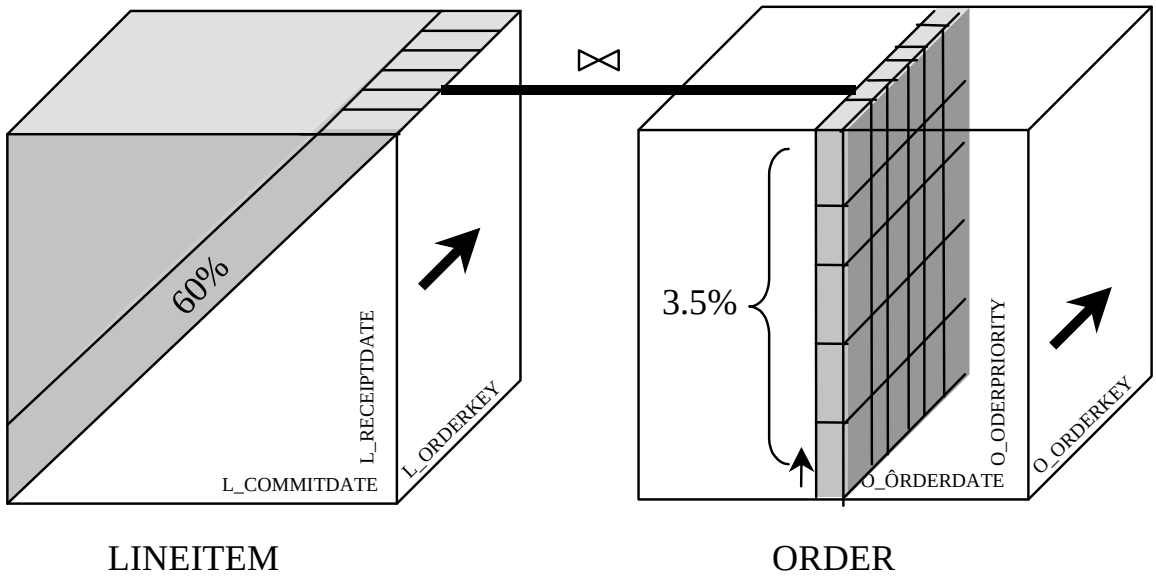


Abbildung 4-32

In Tabelle 4-13 und Abbildung 4-33 sind Antwortzeiten und Cachegröße für die ORDER-Tabelle aufgelistet. Die Verarbeitung von Anfragen mit Restriktionen der Art COMMITDATE < RECEIPTDATE wurde in der Prototypimplementierung nicht realisiert und stellt somit ein weiteres Forschungsgebiet da.

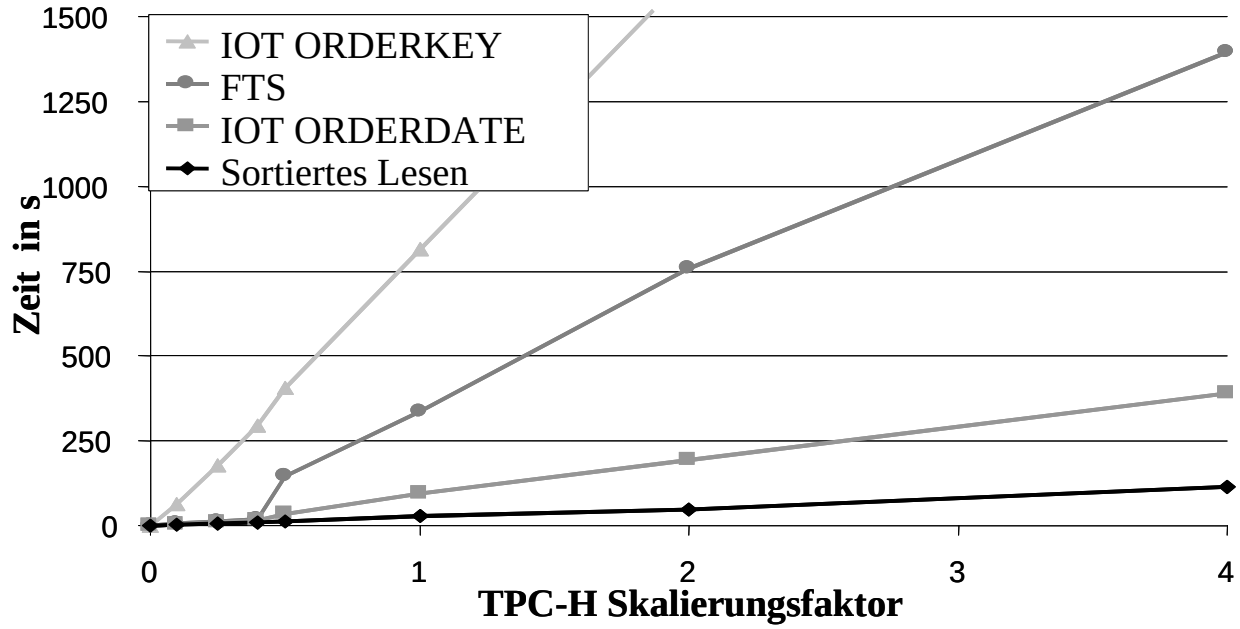


Abbildung 4-33: Antwortzeit ORDER Tabelle, Selektivität 3,5% (Q4)

Die Selektivität auf Tabelle ORDER für den IOT auf ORDERDATE ist so groß, dass er dem FTS und dem IOT auf ODERKEY überlegen ist. Das sortierte Lesen ist durch seine Ausnutzung der mehrdimensionalen Clusterung und Partitionierung eindeutig allen Konkurrenten überlegen, da es die Sortierung und Restriktion gleichzeitig ausnutzt. Obwohl der IOT auf ORDERDATE das Datenvolumen auf 3,5% einschränkt, wurde durch das sortierte Lesen eine Leistungssteigerung um Faktor 3 erreicht. Gegenüber dem FTS

könnte eine Leistungssteigung um Faktor 11 und gegenüber dem IOT auf ORDERKEY sogar ein Faktor von 30 erreicht werden.

Tabellengröße Skalierungsfaktor	32 MB (0.1)	79 MB (0.25)	131MB (0.4)	161MB (0.5)	322 MB (1)	750MB (2)	1498MB (4)
Erste Antwortzeit sortiertes Lesen	0,2s	0,4s	0,1s	0,1s	0.1s	0.2s	0.3s
Scheibenanzahl	64	128	128	128	256	256	512
IOT_ORDERKEY	62.0s	178.4s	295.9s	406.9s	813.8s	1627.5s	3254.9s
IOT_ORDERDATE	3.2s	9.2s	16.8s	34.3s	95.4s	194.2s	390.4s
FTS-Sort	5.2s	12.5s	19.9s	146.4s	335.2s	758.6s	1396.7s
Sortiertes Lesen	3.3s	7.8s	10.5s	12.2s	29.7s	47.8s	113.9s
Cache sortiertes Lesen	0.1MB	0.1MB	0.2MB	0.2MB	0.2MB	0.2MB	0.3MB
Cache IOT/FTS	1.3MB	3.2MB	5.2MB	6.4MB	12.9MB	30.1MB	60.1MB

Tabelle 4-13

4.5 Bewertung und Zusammenfassung

Um eine Relation zu sortieren oder einen Verbundoperator auszuwerten, sollten Restriktionen, die auf den einzelnen Attributen existieren, ausgenutzt werden, um so die E/A- und CPU-Kosten zu reduzieren. Das sortierte Lesen benutzt die hervorragende mehrdimensionale Zugriffsstruktur UB-Baum und eine Sweepline-Technik, um Bereichsanfrage und Sortieren zu kombinieren. Hierdurch kann die Selektionsphase, d.h. die Auswertung der mehrdimensionalen Bereichsanfrage und Sortierphase der Verarbeitung in einem Schritt durchgeführt werden. Es wurde gezeigt, dass das sortierte Lesen für mehrdimensionale Anfragen, die typisch in relationalen Datenbanksystemen sind, eine lineare E/A-Komplexität und sublineare Speicher-Kosten für den benötigten Cache hat. Ein wesentlicher Vorteil besteht darin, dass das sortierte Lesen das Teilergebnis kontinuierlich erzeugt, da die Scheiben in Sortierrichtung geladen werden und somit bereits sortiert werden können, obwohl der Rest der Daten noch nicht geladen worden ist. Der Merge-Sort-Algorithmus hingegen kann erst das Ergebnis erzeugen, wenn alle Daten geladen worden sind. Durch das sortierte Lesen geht der Sortier-Operator von einem blockenden zu einem nicht blockenden Operator über. Verglichen mit den existierenden Techniken kann das erste Teilergebnis durch das sortierte Lesen erheblich schneller erzeugt werden. Dies führt zu einer besseren Antwortzeit. Die Technik des sortierten Lesens ermöglicht somit den Einsatz des Pipelinings in der Anfrageverarbeitung, die ein Sortieroperator enthält. Die Benchmark-Ergebnisse des TPC-H des sortierten Lesens zeigen die Praxisrelevanz dieser Technik. Die zwei hier dargestellten Queries zeigen beeindruckend den Performanzgewinn des sortierten Lesens, der durch die Partitionierungstechnik des UB-Baums entsteht.

So konnte z.B. die Verarbeitungszeit für *Q*3 gegenüber den FTS und IOT um Faktor 3 gesteigert werden. Das hier vorgestellte Kostenmodell basiert auf gleichverteilten Daten. Sind die Daten perfekt gleichverteilt, und man verwendet die minimale Breite der Scheibe im SRQ-Algorithmus, wird fast jede Region zweimal geladen. Dies liegt am Fransen-Phänomen. Verbreitert man die Scheibenbreite um Faktor 2, kann das Problem

entsprechend reduziert werden. Für nicht gleichverteilte Daten kommt es zu einer Vergrößerung der benötigten Cachegröße. Dies kann im schlimmsten Fall dazu führen, dass die gesamte Ergebnismenge in einer Scheibe enthalten ist. Das folgende Kapitel führt einen Algorithmus ein, der dieses Problem löst.

Let´s look at the records

Al Smith (1928)

5 Sortiertes Lesen mit variablen Scheiben, der Tetris-Algorithmus

In diesem Kapitel wird nun ein Algorithmus vorgestellt, der nicht mehr auf dem Bereichsanfrage-Algorithmus basiert. Er benötigt keine Statistik über die Datenbankgröße und deren Datenverteilung, wie es der SRQ-Algorithmus benötigt. Die Grundidee wurde in [MarB98] vorgestellt. Der Autor dieser Arbeit hat eine Methode entwickelt, die es ermöglicht, die Idee effizient umzusetzen [MarZB99]. Die wesentliche Innovation des Autors dieser Arbeit besteht in der Einführung einer neuen Ordnung, der Tetris-Ordnung, die das mehrdimensionale Schnittproblem mit der Sweepline auf ein eindimensionales Problem reduziert.

5.1 Problemstellung

Der SRQ-Algorithmus, der auf statisch vorberechneten Scheiben basiert, hat folgende Probleme:

- Bei nicht gleichverteilten Daten kann es zu einer nicht optimalen Partitionierung des Raumes kommen, so dass eine Scheibe nicht mehr vollständig in den Arbeitsspeicher passt. Somit ist der Vorteil des internen Sortierens hinfällig und statt dessen muss z.B. externes Sortieren durch Verschmelzen benutzt werden.

- Durch die statische Aufteilung in Teilbereichsanfragen und deren jeweilige Verarbeitung durch Bereichsanfragen werden Regionen, die an der Grenze zwischen zwei aufeinander liegenden Teilbereichen liegen, mindestens zweimal gelesen.

5.2 Grundidee

Um diesen Problemen zu begegnen wird nun die Grundidee des allgemeinen *Tetris-Algorithmus* [MarB98], [MarZB99] vorgestellt. Die Grundidee besteht darin, die Sweepline nur soweit in Sortierrichtung zu verschieben, bis wieder eine vollständige Scheibe im Arbeitsspeicher enthalten ist. Alle Daten, die kleiner als die Sweepline sind, bilden den statischen Teil und können somit *sortiert* und als *Teilergebnis* bereits weitergereicht werden. Die Tupel, die gleich oder größer als die Sweepline sind, gehören noch zum dynamischen Bereich und werden somit im Cache gehalten. Da Tupel, die zum dynamischen Bereich gehören, im Arbeitspeicher gehalten werden, braucht eine Region auch nur genau einmal in den Arbeitsspeicher geladen werden. Infolgedessen reduzieren sich auch die E/A-Operationen auf den Daten-Segmenten des UB-Baums und stimmen im Wesentlichen mit den E/A-Kosten einer reinen Bereichsanfrage überein. Der SRQ-Algorithmus wird somit um einen Cachebereich, in dem die dynamischen Tupel gehalten werden, erweitert.

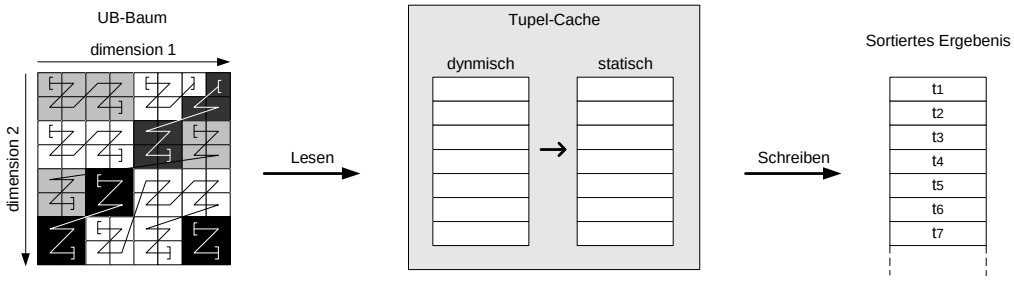


Abbildung 5-1: Struktur des Tetris-Algorithmus

Abbildung 5-1 zeigt die Struktur des Tetris-Algorithmus. Der Tupelcache besteht jetzt aus zwei Teilen: dem dynamischen und statischen Teil. Die wesentliche Innovation des Tetris-Algorithmus besteht darin, die Regionen durch eine *neue Ordnung* (Tetris-Ordnung) aus dem UB-Baum in Sortierrichtung zu lesen, d.h. es wird jedem Punkt des Basisraums Ω eine Tetris-Adresse $T_k(x)$ zugeordnet. k bezeichnet hierbei die Sortierdimension.

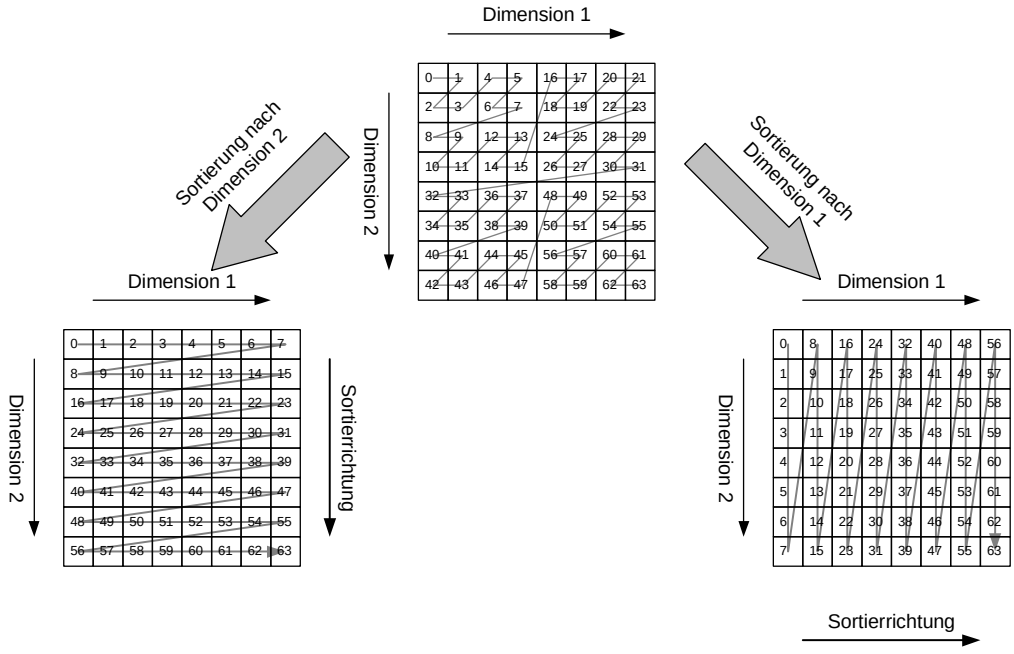
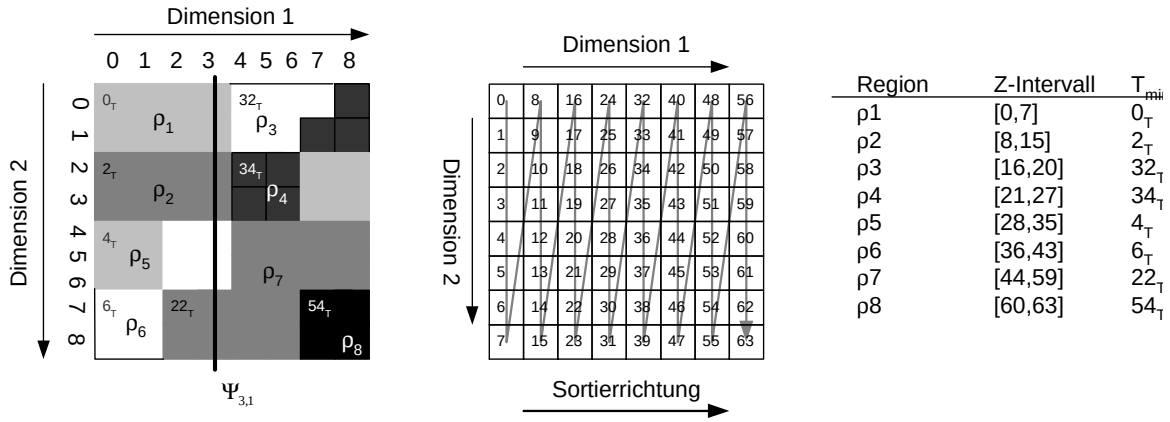


Abbildung 5-2:Tetris-Ordnung im 2-dimensionalen Fall

Abbildung 5-2 zeigt die Tetris-Ordnungen für den 2-dimensionalen Fall. Für jede Sortierrichtung existiert genau eine T-Ordnung. Jedem Punkt im Basisraum Ω wird eindeutig eine Ordnungszahl zugeordnet.

Basierend auf der T-Ordnung kann nun die Sweep-Hyperebene $\Psi_{k,c}$, die den dynamischen und statischen Bereich trennt, als Punkt repräsentiert werden, da jede Sweep-Hyperebene $\Psi_{k,c}$ als ein T-Intervall dargestellt ist.

Die Sweep-Hyperebene $\Psi_{1,3}$ (siehe Abbildung 5-3.a) z.B. besteht aus einem zusammenhängenden T-Intervall $[24,31]_T$ auf der T-Ordnung T_1. Somit hat eine Sweepline immer einen minimalen und einen maximalen Wert. Da die Sweepline nicht zum statischen Bereich gehört, wird das Minimum der Sweepline als Separator zwischen dem dynamischen und statischen Bereich herangezogen. Somit ist der Separator zwischen dem statischen und dynamischen Bereich die T-Adresse 24_T. Damit ist ein n-dimensionales Schnittproblem auf ein 1-dimensionales Problem reduziert. Der Tetris-Algorithmus liest nun die Regionen nicht in Z-Ordnung sondern in T-Ordnung aus dem UB-Baum. Liest man die Regionen in aufsteigender T-Ordnung iterativ aus dem UB-Baum, entspricht das dem Sweepline-Prinzip und man kann so alle Tupel, die zum statischen Bereich gehören, sortiert an den Aufrufer weiterreichen. Für den Cache kann hier z.B. eine Heap-Struktur verwendet werden. Da die Tetris-Adressen $T_k(x)$ eine 1-dimensionale Ordnung auf Ω erzeugen, die dem Prinzip der *aufsteigenden Scheiben* entspricht, kann man nun iterativ die nächste minimale Scheibe $S^{-t}_{[a,b],k,Q}(R)$ abarbeiten. Das Prinzip der *aufsteigenden Scheiben* bedeutet, dass ein gelesenes Tupel bezüglich der Sortierdimension *k* größer oder gleich seinem Vorgänger ist. Die Tetris-Ordnung erfüllt diese Anforderung.



Region	Z-Intervall	T_{min}
ρ1	[0,7]	0_T
ρ2	[8,15]	2_T
ρ3	[16,20]	32_T
ρ4	[21,27]	34_T
ρ5	[28,35]	4_T
ρ6	[36,43]	6_T
ρ7	[44,59]	22_T
ρ8	[60,63]	54_T

(a) Z-Partitionierung (b) T-Adresse (c) Region, Z-Interval, Tmin

Abbildung 5-3: Regionszerlegung, T-Ordnung, Z-Intervalle, T_{min}

Liest man z.B. die Regionen der Regionszerlegung Θ, die in Abbildung 5-3.a dargestellt sind, sortiert nach Dimension 1, läuft der Tetris-Algorithmus die T-Kurve (siehe Abbildung 5-3.b) T_1 entlang und lädt eine Seite in den Arbeitsspeicher, sobald die entsprechende Region das erste Mal von der T-Kurve T_1 geschnitten wird. Abbildung 5-3.a zeigt jeweils die minimale T-Adresse T_{min} zu einer Region. Der Wert bestimmt somit die Reihenfolge, wann eine Region verarbeitet wird. Somit erhält man folgende Regionsfolge: $\rho_1, \rho_2, \rho_5, \rho_6, \rho_7, \rho_3, \rho_4, \rho_8$.

Die Daten, die durch die Ausdehnung einer Z-Region nach der Sortierung zu früh vom Sekundärspeicher geladen worden sind, werden im dynamischen Cache zwischengespeichert. Sobald ihr T-Wert kleiner als die T-Adresse des Separators der Sweep-Hyperebene ist, können die Tupel aus dem Cache entfernt werden und als

Teilergebnis an den Aufrufer weitergeleitet werden. Der Tetris-Algorithmus terminiert, sobald die maximale T-Adresse des Anfragebereichs erreicht ist.

5.3 Formales Modell

Wir erweitern hierzu zunächst das Raum-Tupel-Modell aus Tabelle 4-1 um den Begriff Unterraum und Z-Unterraum.

Definition 5-1: Unterraum (Subspace), Z-Unterraum

Ein Unterraum ist ein Teilgebiet des Basisraums Ω, d.h. $\upsilon \subseteq \Omega$. Ein Unterraum muss weder rechteckig noch zusammenhängend sein.

Ein Z-Unterraum $\upsilon = [\alpha:\beta]$ ist ein Unterraum υ, der durch ein zusammenhängendes Z-Intervall $[\alpha,\beta]$ beschrieben wird.

Ein Z-Unterraum ist somit ein beliebiges Z-Intervall im Basisraum Ω. Zu beachten ist hier, dass die Kapazität noch nicht festgelegt ist, d.h. die Anzahl der Datensätze, die in den Z-Unterraum fallen, ist unspezifiziert.

Basierend auf Definition 5-1 wird nun die Menge Φ eingeführt. Φ bezeichnet die Menge von Z-Intervallen[7], deren Bild $Z^{-1}(\Phi)$ genau den noch nicht verarbeiteten Anfrageraum *Q* im Basisraum Ω abdeckt. Noch nicht verarbeitet heißt in diesem Zusammenhang, dass die Regionen und somit die Seiten, die diesen Raum abdecken, noch nicht geladen worden sind. Die Schreibweise $Z^{-1}(\Phi)$ ist eine Kurzschreibweise für:

$$\{Z^{-1}(\chi))|\ \chi \in [\alpha,\beta] \text{ und } [\alpha,\beta] \in \Phi\} \qquad \text{Gl: 5-1}$$

Ohne Beschränkung der Allgemeinheit wird im formalen Modell nur die minimale T-Adresse betrachtet. Die maximale T-Adresse und die entsprechenden Sätze können direkt aus dem hier dargestellten formalen Modell hergeleitet werden.

Wir bezeichnen die minimale T_k-Adresse von Φ als *Eventpunkt* τ_k. *k* bezeichnet die Sortierdimension. Hierzu wird für jedes Z-Intervall, das Element von Φ ist, die minimale T_k-Addresse berechnet. Formal heißt das:

$$\tau_k = min\{T_k(Z^{-1}(\chi))|\ \chi \in [\alpha,\beta] \text{ und } [\alpha,\beta] \in \Phi\} \qquad \text{Gl: 5-2}$$

[7] Es handelt sich hier um beliebige Z-Intervalle, die mehrere Regionen oder auch nur einen Teil einer Region umfassen.

Raum		Relation	
Ω	Basisraum von Punkten	R	Relation von Tupeln
Θ	Regionszerlegung	P_R	Seitenmenge
W^r	Regionswelle	W	Seitenwelle
$Z^{-1}(\Phi)$	Nicht verarbeiteter Anfragebereich von Q		
Q	Anfragebereich	E	Ergebnismenge
S^{-r}	Schicht	S^{-t}	Scheibe
ρ	Region	p	Seite
x	Punkt	x	Tupel

Tabelle 5-1: Raum-Tupel-Modell

Tabelle 5-1 fasst das Raum-Tupel-Modell zusammen.

5.4 Tetris-Ordnung

Der RQ-Algorithmus liest die Seiten vom Sekundärspeicher in den Arbeitsspeicher in Z-Ordnung. Die Z-Ordnung entspricht jedoch nicht der Zielordnung, so dass man das vollständig sortierte Ergebnis erst erzeugen kann, wenn alle Tupel, die zur Ergebnismenge E gehören, bereits gelesen worden sind. Das Ziel ist somit, die Regionen des Basisraums Ω in Zielordnung zu lesen. Hierzu wird das Konzept der *aufsteigend sortierten Scheibenfolge* ($\langle S_1^{-t}, S_2^{-t}, ..., S_n^{-t} \rangle, \leq_{dim}$) herangezogen.

Folgende Randbedingungen für die Zielordnung existieren:

- Der Beitrag (siehe Definition 5-2) der Sortier-Dimension muss maximal sein.
- Die Zielordnung soll eine totale Ordnung auf Ω erzeugen, um einen eindeutigen Separator zwischen dem *statischen* und *dynamischen Bereich* zu spezifizieren.

Der Beitrag einer Dimension zu einer Ordnung ist wie folgt definiert:

Definition 5-2: Beitrag einer Dimension zu einer Ordnung

Der Beitrag *con*(*ord*, *i*) einer Dimension *i* zu einer Ordnung *ord* ist:

$$con(ord, i) = \frac{\max\{ord(0,0,...,xi,...,0) \mid x_i \in D_i\}}{\max\{ord(x) \mid x \in \Omega\}}$$

Hierbei bezeichnet *ord*(*x*) die Ordnungszahl (Adresse) für das Element *x* bezüglich der Ordnung *ord*.

Der Beitrag einer Dimension zur Ordnungszahl ist abhängig von der Wertigkeit seiner Bits innerhalb der Ordnungszahl. Eine Dimension hat demzufolge den größten Beitrag zur Ordnungszahl, wenn sie die höchstwertigen Bits in sich vereint. Für einen *d*-dimensionalen Basisraum Ω und eine Auflösung $r = 2^s$ in jeder Dimension muss für die Sortier-Dimension *k* gelten:

$$T_k(0,0,...,x_k,...,0) = \sum_{j=0}^{s-1} x_{k,j}\, 2^{j+s\cdot(d-1)} \qquad \text{Gl: 5-3}$$

da α_i mit $i \in \{s \cdot d - 1,..., s(d-1)\}$ die höchstwertigen Bits der Adresse sind. Reduziert man ein *d*-dimensionales Tupel *x* um die Dimension *i*, dann schreiben wir:

$$x|\mathrm{i} := x_1,..,x_{i-1},x_{i+1},..,x_d \qquad \text{Gl: 5-4}$$

Wir definieren die Tetris-Adresse $T_k(x)$ dementsprechend wie folgt:

Definition 5-3: Tetris-Adresse

Die Tetris-Adresse $T_k(x)$ für einen *d*-dimensionalen Basisraum Ω mit einer Auflösung *r* ($s = \log_2 r$) und der Sortier-Dimension *k* ist die Ordnungszahl:

$$T_k(x) = \underbrace{\left(\sum_{j=0}^{s-1} x_{k,j}\, 2^{j+s\cdot(d-1)}\right)}_{\text{Sortierdimension}} + Z(x \mid k)$$

$$= \underbrace{\left(\sum_{j=0}^{s-1} x_{k,j}\, 2^{j+s\cdot(d-1)}\right)}_{\text{Sortierdimension}} + \underbrace{\sum_{j=0}^{s-1}\sum_{i=k+1}^{d} x_{i,j}\, 2^{j\cdot(d-1)+i-2} + \sum_{j=0}^{s-1}\sum_{i=1}^{k-1} x_{i,j}\, 2^{j(d-1)+i-1}}_{Z(x|k)}$$

Die Struktur der binären Darstellung einer T-Adresse ist somit die Binärdarstellung des Wertes der Sortierdimension *k* konkateniert mit der reduzierten Z-Adresse $Z(x|k)$:

$$x_k \circ Z(x \mid k) = \qquad \text{Gl: 5-5}$$

$$= \underbrace{\overbrace{x_{k,s-1}...x_{k,0}}^{\text{Sortier-Dimension}}}_{\text{T-Stufe 0}} \cdot \underbrace{\overbrace{x_{d,s-1}...x_{k+1,s-1}x_{k-1,s-1}...x_{1,s-1}}^{Z(x|k)}}_{\text{T-Stufe 1}} \cdot ... \cdot \underbrace{x_{d,0}...x_{k+1,0}x_{k-1,0}...x_{1,0}}_{\text{T-Stufe } s}$$

Nach Gl: 3-18 hat eine Z-Adresse *s* Stufen, wohingegen eine T-Adresse aus *s + 1* T-Stufen besteht, da die T-Stufe 0 durch den Wert der Sortier-Dimension belegt ist.

Eine Z-Adresse eines *d*-dimensionalen Basisraums Ω mit einer Auflösung von *r* mit $s = \log_2 r$ besteht aus

$$n = s \cdot d \qquad \text{Gl: 5-6}$$

Bits. Die Binärdarstellung einer T-Adresse besteht aus *s* Bits für die Sortier-Dimension und *s* (*d*-1) Bits für die reduzierte Z-Adresse $Z(x|k)$. Es gilt:

$$s + s(d-1) = s + sd - s = sd \qquad \text{Gl: 5-7}$$

Dies entspricht der Anzahl der Bits einer Z-Adresse. Abbildung 5-4 zeigt die Tetris-Adressen für den 2-dimensionalen Basisraum Ω.

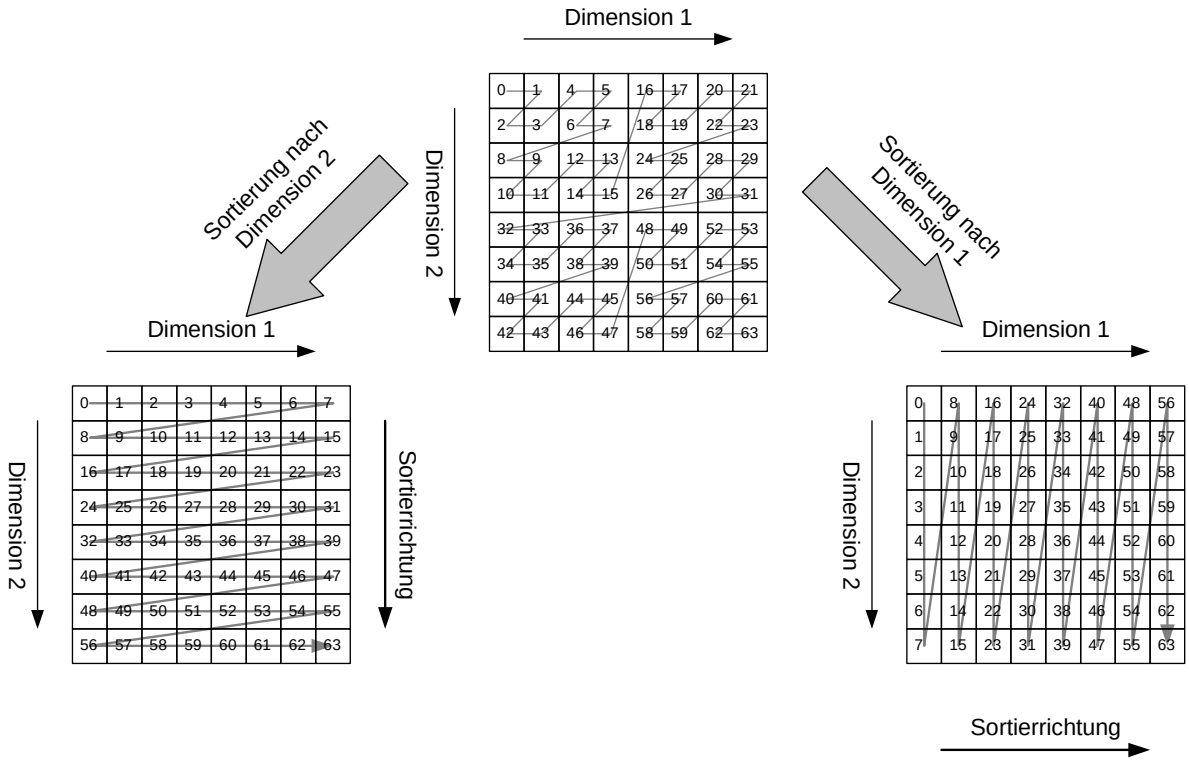


Abbildung 5-4: Tetris-Ordnung für den 2-dimensionalen Fall

Satz 5-1: $T^{-1}_{k,i}(\alpha)$

Die *inverse Funktion* $T^{-1}_{k,i}(\alpha)$ berechnet aus der Tetris-Adresse α_T mit der binären Repräsentation $\alpha = \alpha_{s\cdot d-1}\alpha_{s\cdot d-2}...\alpha_0$ das Attribut x_i des Tupels $x = (x_1,...x_d)$:

$$x_i = T^{-1}_{k,i}(\alpha) = \begin{cases} \sum_{j=0}^{s-1} \alpha_{s\cdot(d-1)+j}\, 2^j & , \quad k = i \\ \sum_{j=0}^{s-1} \alpha_{j(d-1)+(i-1)}\, 2^j & , \quad k < i \\ \sum_{j=0}^{s-1} \alpha_{j(d-1)+(i-2)}\, 2^j & , \quad k > i \end{cases}$$

Beweis: Satz 5-1

Nach Definition 5-3 ist die binäre Darstellung der Tetris-Adresse

$$T_k(x) = \alpha_{s\cdot d-1}\alpha_{s\cdot d-2}...\alpha_0 \qquad \text{Gl: 5-8}$$

$$= \underbrace{\alpha_{(s-1)+s(d-1)}...\alpha_{s(d-1)}}_{\text{Sortierdimension}} \underbrace{\alpha_{(s-1)\cdot(d-1)+d-2}...\alpha_{(s-1)\cdot(d-1)+(k+1)-2}}_{i>k}$$

$$\underbrace{\alpha_{(s-1)\cdot(d-1)+(k-1)-1}...\alpha_{(s-1)\cdot(d-1)+1-1}}_{i<k}...\alpha_0$$

1) *i* = *k*

Die Bits an Position $\underbrace{\alpha_{(s-1)+s(d-1)}...\alpha_{s(d-1)}}_{\text{Sortierdimension}}$ sind die Sortierdimension x_k:

$$x_k = \sum_{j=0}^{s-1} \alpha_{j+s(d-1)} 2^j \qquad \text{Gl: 5-9}$$

2) $i < k$

Die binäre Darstellung für die Dimension, die kleiner als die Sortierdimension k ist, entspricht den Bits $\alpha_{(s-1)\cdot(d-1)+(i-1)}\alpha_{(s-2)(d-1)+(i-1)}....\alpha_{i-1}$ der Tetris-Adresse α. Setzt man die Bits zu Basis 2 erhält man:

$$x_i = \sum_{j=0}^{s-1} \alpha_{j(d-1)+(i-1)} 2^j \qquad \text{Gl: 5-10}$$

3) $i > k$

Die binäre Darstellung für die Dimension, die größer als die Sortierdimension k ist, entspricht den Bits $\alpha_{(s-1)\cdot(d-1)+(i-2)}\alpha_{(s-2)(d-1)+(i-2)}....\alpha_{i-2}$ der Tetris-Adresse α. Setzt man die Bits zu Basis 2, erhält man:

$$x_i = \sum_{j=0}^{s-1} \alpha_{j(d-1)+(i-2)} 2^j \qquad \text{Gl: 5-11}$$

Wendet man die obigen Formeln an, um $x_1, x_2, ..., x_d$ aus der Tetris-Adresse α zu berechnen, erhält man die *inverse Funktion* $(x_1, x_2, ..., x_d) = T^{-1}_{k,i}(\alpha)$
q.e.d.

Satz 5-2: $T_k(\Omega)$

Der Wertebereich W für die Tetris-Funktion T_k angewendet auf den d-dimensionalen Basisraum Ω mit $\Omega_i = \{0,..., 2^s - 1\}$ ist $W = \{0,..., 2^{s \cdot d} - 1\}$:

$$T_k : \Omega \rightarrow T_k(\Omega) = \{0,...,2^{s \cdot d} - 1\}$$

Beweis: Satz 5-2

1) Minimum

Nach Definition 5-3 gilt $T_k(0,...,0) = 0$. Es gilt:

$$\forall x \in \Omega \setminus (0,...,0): T_k(x) > 0 \qquad \text{Gl: 5-12}$$

da mindestens ein x_i existiert, dessen binäre Repräsentation ein Bit gesetzt hat. Somit ist $T_k(0,...,0) = 0$ das Minimum von $T_k(x)$.

2) Maximum

Nach Definition 5-3 ist $T_k(r-1,...,r-1) = 2^{s-1+s\cdot(d-1)+1} - 1 = 2^{s-1+s\cdot d-s+1} - 1 = 2^{s\cdot d} - 1$.

3) Nach Satz 5-1 ist $T_k^{-1}(\alpha)$ die inverse Funktion zu $T_k(x)$, so dass $T_k(x)$ eine bijektive Abbildung von Basisraum Ω nach $\mathbb{N}$ ist. Da die Kardinalität des Basisraums Ω nach Gl: 3-16:

$$|\Omega| = \prod_{i=1}^{d} r_i = \prod_{i=1}^{d} \log_2 s = \prod_{i=1}^{d} 2^s = 2^{s \cdot d} \qquad \text{Gl: 5-13}$$

ist, besteht der Wertebereich von T_k aus $2^{s \cdot d}$ verschiedenen Werten. Mit dem Minimum und Maximum von T_k ergibt sich somit der Wertebereich $\{0,...,2^{s \cdot d} - 1\}$ q.e.d.

Wir bezeichnen das Bild T_k^{-1} als *Tetris-Kurve*.

Satz 5-3: T-Kurve

Die Tetris-Kurve (T-Kurve) ist eine raumfüllende Funktion.

Beweis: Satz 5-3

Die Funktion T_k^{-1} bildet $D = \{0,...,2^{s \cdot d} - 1\} \subset \mathbb{N}$ auf Ω ab. Nach Satz 5-1 ist T_k^{-1} die bijektive Abbildung von $D \rightarrow \Omega$.
q.e.d.

Basisräume mit unterschiedlichen Kardinalitäten in den Dimensionen führen zu komplexeren Funktionen für die *T-Kurve*. Die grundlegenden Eigenschaften, auf denen die hier vorgestellten Algorithmen basieren, bleiben jedoch erhalten. Ohne Beschränkung der Allgemeinheit benutzen wir einen Basisraum Ω mit gleichen Kardinalitäten r für alle Dimensionen, da die grundlegenden Ideen besser verständlich sind. Abbildung 5-5 zeigt die T-Kurve T_3 für den 3-dimensionalen Fall.

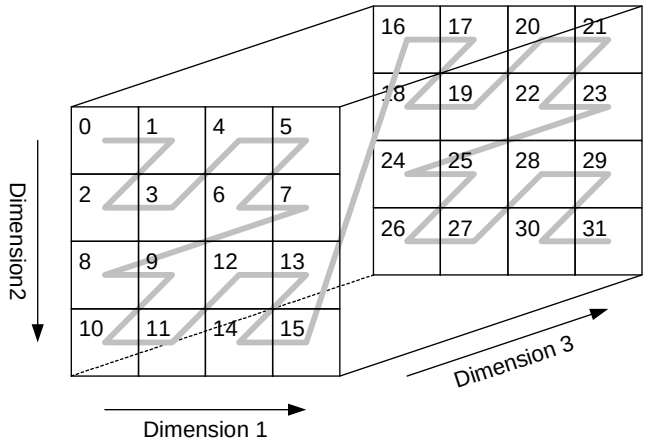


Abbildung 5-5: Tetris-Kurve für den 3-dimensionalen Fall

5.5 Berechnung der Tetris-Adresse

Im vorherigen Abschnitt wurde die Formel T_k angegeben, die aus kartesischen Koordinaten eines Tupels des Basisraums Ω die *T-Adresse* berechnet. Da die Algorithmen mit *Z-Adressen* arbeiten, muss eine Funktion angegeben werden, die eine *Z-Adresse* in eine *T-Adresse* umwandelt. Dies wird durch die Komposition $T_k \circ Z^{-1}$ erfüllt:

$$\alpha_T = T_k(Z^{-1}(\alpha_z)) \qquad \text{Gl: 5-14}$$

Die Abbildungsfunktion, die eine Z-Adresse direkt auf eine T_k-Adresse abbildet, bezeichnen wir mit f_k. Der Zusammenhang ist in Abbildung 5-6 dargestellt.

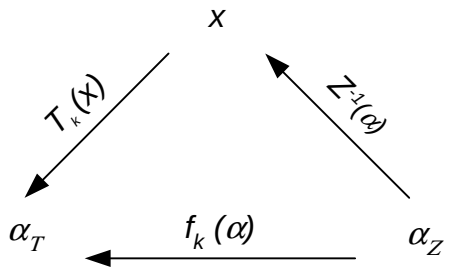


Abbildung 5-6: Transformationsfunktionen

Satz 5-4:

Gegeben sei ein *d*-dimensionaler Basisraum Ω. *s* sei die Anzahl der Stufen der Z-Adresse α_Z. Dann bildet :

$$f_k(\alpha_Z) = \underbrace{\sum_{j=0}^{s-1} \alpha_{(k-1)+j\cdot d}\, 2^{j+(s-1)d}}_{Sortier-Dimension} + \underbrace{\sum_{j=0}^{s-1}\sum_{i=k+1}^{d} \alpha_{(j\cdot d)+i-1}\, 2^{j(d-1)+i-2}}_{i>k \wedge k<d} + \underbrace{\sum_{j=0}^{s-1}\sum_{i=1}^{k-1} \alpha_{(j\cdot d)+i-1}\, 2^{j(d-1)+i-1}}_{i<k \wedge k>1}$$

auf die entsprechende T-Adresse ab.

Wird nur der Beitrag einer Dimension *k* zur Z-Adresse betrachtet, schreibt man $Z(x_k)$. Hierbei sind die übrigen Dimensionen unspezifiziert. Um das *i-te* Bit in einem Binärstring zu referenzieren, schreibt man α_i mit $i \in \{0, ..., n\}$ und $\alpha_i \in \{0,1\}$. Mit

$$\alpha_i 2^j \qquad \text{Gl: 5-15}$$

beschreibt man die Bitpermutation, die Bit *i* auf Bit *j* abbildet.

Beweis:

1) x_k

Nach Definition 5-3 und Gl: 5-5 liegen auf Bitposition $s(d-1)$ bis $sd-1$ die s Bits der Sortierdimension k. Es muss also gelten:

$$T_k(x_k) = f_k(Z(x_k))$$

$$\Leftrightarrow \sum_{j=0}^{s-1} x_{k,j} 2^{j+s(d-1)} = f_k(\sum_{j=0}^{s-1}\sum_{i=k}^{k} x_{i,j} 2^{j\cdot d+i-1})$$

$$\Leftrightarrow \sum_{j=0}^{s-1} x_{k,j} 2^{j+s(d-1)} = f_k(\sum_{j=0}^{s-1} x_{k,j} 2^{j\cdot d+k-1})$$

$$\Leftrightarrow \sum_{j=0}^{s-1} x_{k,j} 2^{j+s(d-1)} = f_k(\underbrace{\alpha_{(k-1)+d(s-1)}\ldots\alpha_{(k-1)+d(s-2)}\ldots\alpha_{k-1}}_{\text{Binärdarstellung der Z-Adresse } Z(x_k)})$$

$$\Leftrightarrow \sum_{j=0}^{s-1} x_{k,j} 2^{j+s(d-1)} = \sum_{j=0}^{s-1} \alpha_{(k-1)+j\cdot d} 2^{j+s(d-1)}$$

$$\Leftrightarrow \sum_{j=0}^{s-1} x_{k,j} 2^{j+s(d-1)} = \sum_{j=0}^{s-1} x_{k,j} 2^{j+s(d-1)}$$

2) $Z(x|k)$

Nach Definition 5-3 besteht die T-Adresse auf Bitposition $s\cdot d-(s+1)$ bis 0 aus der reduzierten Z-Adresse $Z(x|k)$.

Fall 1: $i < k$

Die Bitposition innerhalb einer Stufe bleibt erhalten, wobei die Länge einer Stufe um eins reduziert wird. Somit erhält man:

$$\sum_{j=0}^{s-1}\sum_{i=1}^{k-1} \alpha_{(\underbrace{j\cdot d}_{\text{Stufe}})+i-1} 2^{j(d-1)+i-1}$$

Fall 2: $i > k$

Die Bitposition innerhalb einer Stufe wird um eins reduziert und somit die Länge einer Stufe. Man erhält:

$$\sum_{j=0}^{s-1}\sum_{i=k+1}^{d} \alpha_{(\underbrace{j\cdot d}_{\text{Stufe}})+i-1} 2^{j(d-1)+i-2}$$

Kombiniert man die obigen Fälle erhält man die Funktion f_k.
q.e.d.

Die Bitstruktur der *T*-Adresse besteht aus dem Sortierattribut konkateniert mit den Z-verschränkten Bits ohne die Bits, die zum Sortierattribut gehören.

$$T_j(x) = x_j \circ Z(x|k) \qquad \text{Gl: 5-16}$$

Aus Satz 5-4 und Gl: 5-16 folgt, dass die Berechnung der T-Adresse aus einer Z-Adresse aus folgenden Operationen besteht:

- Extrahierung der Sortierdimension aus den Stufen der Z-Adresse.

- Setzen des Sortierattributs auf T-Stufe 0 der T-Adresse.
- Verschiebe alle Bits, deren Dimension größer als die Sortierdimension ist, auf das nächste niederwertige Bit.
- Füge die Stufe *n* auf T-Stufe n+1 ein.

Abbildung 5-7 zeigt einen Algorithmus, der eine Z-Adresse in eine T-Adresse umwandelt. Hierbei handelt es sich um die Prototyp-Implementierung in C.

Bezeichnung	Beschreibung
AD	Bezeichnet eine Z-Adresse oder T-Adresse. Sie besteht aus einem Array von Integer und Metadaten
DUB	Metadaten des UB-Baums
adTet.Steps	Anzahl der Steps der T-Adresse
ExtAttr	Sortierdimension
DUB_stepLength(stepNo)	Gibt die Anzahl der Dimensionen zurück, die zu dem Step stepNo beitragen
AD_stepGetDim(DUB * dub, AD * ad, int step, int posinstep)	Gibt den Bitwert der Dimension in einem Step zurück
AD_stepSet1Dim(DUB * dub, AD * ad, int step, int dim)	Setzt das Bit der Adresse *ad* in Stufe *Step* an der Dimension *dim* auf den Wert 1.
AD_stepSet0Dim(DUB * dub, AD * ad, int step, int dim)	Setzt das Bit der Adresse *ad* in Stufe *Step* an der Dimension *dim* auf den Wert 0.
ADTET_reducePart	Setzt den Pointer auf den Anfang der reduzierten Z-Adresse der T-Adresse
AD_set0	initialisiert eine Z-Adresse oder T-Adresse
AD_copy	kopiert eine Z-Adresse oder T-Adresse
AD_setSteps	setzt die Anzahl der Stufen einer Z-Adresse oder T-Adresse

Tabelle 5-2

Tabelle 5-2 enthält die verwendeten Funktionen und Typen, die im Algorithmus AD_zToTetAddress verwendet werden.

```
Status AD_zToTetAddress(DUB * dub, AD * res, AD * ad, int extAttr)
{
   long stepNo;
   long posInStep = 0;
   int bitValue = 0;
   AD adTet;


   AD_set0(dub, &adTet);


   /* first copy the z-address to adtet.*/
   memcpy(&(adTet.binlong[GräKK+01]), &(ad->binlong[0]),
   (sizeof(AD) - 2));
   adTet.steps = ad->steps;

   for (stepNo = 0; stepNo < DUB_adSteps(dub); stepNo++)
   {
      /* test if a step exist for sortAttr */
      if (extAttr < DUB_stepLength(dub, stepNo))
      {
         /* first save the current extract Bit */
         bitValue = AD_stepGetDim(dub, ADTET_reducePart(&adTet),
                                  stepNo, extAttr);
         /*movevthe bit between the maximum and extAttr to the right */
         for (posInStep = extAttr;
             posInStep < (DUB_stepLength(dub, stepNo) - 1);
             posInStep++)
         {
            if (AD_stepGetDim(dub, ADTET_reducePart(&adTet), stepNo,
                                    (posInStep + 1)) == 1)
            {
               AD_stepSet1Dim(dub, ADTET_reducePart(&adTet),
                                 stepNo, posInStep);
            }
            else
            {
               AD_stepSet0Dim(dub, ADTET_reducePart(&adTet),
                                       stepNo, posInStep);
            }
         }
         /* set the maximum bit of the step to 0 because      */
         /*  the extract bit is                               */
         /*  moved to the first step of the address           */
         AD_stepSet0Dim(dub, ADTET_reducePart(&adTet), stepNo,
                     (DUB_stepLength(dub, stepNo) - 1));
         if (bitValue == 1)
            AD_stepSet1Dim(dub, ADTET_extractPart(&adTet),
                    0, ((CUB_tBitLen(dub, extAttr) - 1) - stepNo));
         else
            AD_stepSet0Dim(dub, ADTET_extractPart(&adTet),
                    0, ((CUB_tBitLen(dub, extAttr) - 1) - stepNo));
      }

   }

   AD_set0(dub, res);
   AD_copy(dub, res, &adTet);
   AD_setSteps(dub, res, DUB_adSteps(dub) + 1);
   return OK;

}
```

Abbildung 5-7: Algorithmus zur Berechnung der T-Adresse

Die CPU-Komplexität ist O(*n*). *n* bezeichnet hier die Anzahl der Bits, die zur Repräsentation der T-Adresse benötigt werden.

5.6 Eigenschaften der T-Adresse

Betrachten wir hier zunächst den 2-dimensionalen Fall für einen Basisraum Ω mit einer Auflösung von 4x4 mit dem Ursprung in der oberen linken Ecke (siehe Abbildung 5-8). Die einzelnen Z-Adressen sind in der binären Darstellung abgebildet, wobei die Stufen durch einen Punkt voneinander getrennt sind. Abbildung 5-8 zeigt den jeweiligen Beitrag für die Dimension 1 und Dimension 2 zur Z-Adresse. Hier ist besonders zu bemerken, dass die Bits der Z-Adresse z.B. an Position 1 der Dimension 1 entlang der Dimension 2 immer den konstanten binären Wert 01 haben.

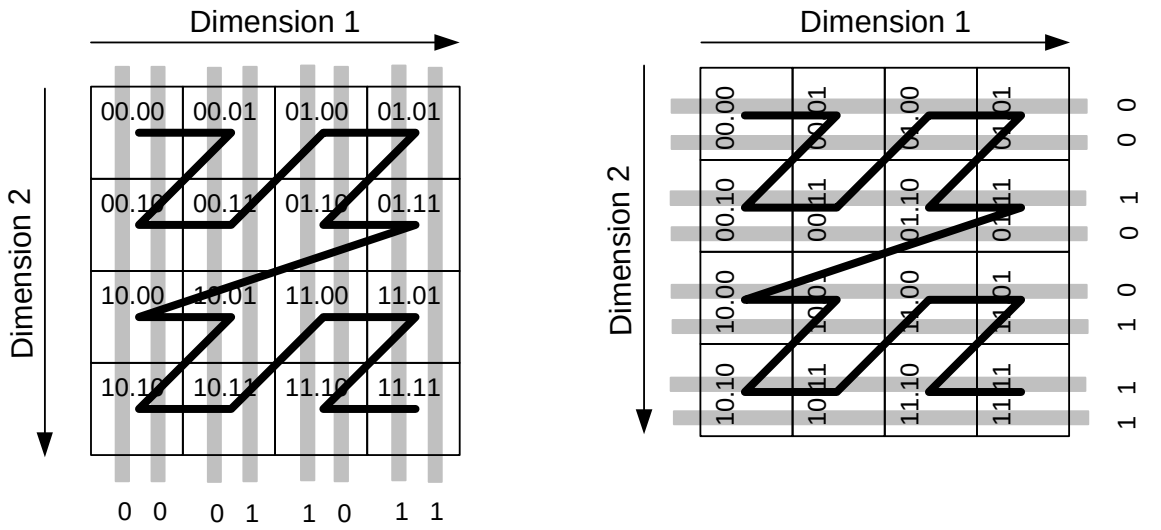


Abbildung 5-8: Beitrag der Sortierattribute zur Z-Adresse

Dies führt zu folgendem Satz:

Satz 5-5:

Das *k*-1-te Bit einer Stufe ist für alle Z-Adressen, die $\Psi_{k,c}$ schneiden, konstant. *k* bezeichnet die Sortierdimension.

Der Grund ist in der Abbildungsvorschrift von der Z-Adresse begründet. Hierbei handelt es sich um eine Permutation der Bits des Tupels, wobei die Position der Bits einer Dimension in jeder Stufe fest ist (siehe Definition 3-21).

Beweis:

Nach der Voraussetzung schneidet die Z-Adresse β die *Sweep-Ebene* $\Psi_{k,c}$ .Daraus folgt, dass $\beta = Z(x)$ ist und $x \in \Psi_{k,c}$. Demzufolge ist $x_k = c$. Nach Definition 3-21 ist die Z-Adresse wie folgt definiert:

$$Z(x) = \sum_{j=0}^{s-1} \sum_{i=1}^{d} x_{i,j} 2^{j \cdot d + i - 1}$$

und in der Dimension *k* gilt für die Z-Adresse:

$$Z(x_k) = \sum_{j=0}^{s-1} x_{k,j} 2^{j \cdot d + k - 1}$$

d.h., dass Bit k –1 auf Stufe (s-1) –j ist $x_{k,j}$ für alle *Z-Adressen* die $\Psi_{k,c}$ schneiden und ist somit konstant.

q.e.d.

5.7 Der minimale/maximale Tetris-Wert eines Z-Intervalls

Die Berechnung der nächsten Z-Region durch den Tetris-Algorithmus basiert im Wesentlichen auf der Berechnung der minimalen bzw. maximalen T-Adresse eines Z-Intervalls. Die minimale bzw. maximale T-Adresse eines Z-Intervalls bezüglich der Sortier-Dimension *k* ist der Punkt mit der kleinsten bzw. größten T-Adresse, die zu dem betrachteten Z-Intervall gehört. Es existiert jeweils ein eindeutiger Punkt, da es sich bei der T-Funktion um eine totale Ordnung auf Ω handelt (siehe Satz 5-3). Diese Eigenschaft kann wie folgt formalisiert werden:

Definition 5-4: Minimale und maximale T-Adresse

Die *minimale T-Adresse* β mit der Sortierdimension *k* eines Z-Intervalls $[\alpha,\chi]_z$ ist:

$$\beta := \min\{T_k(x) \mid Z(x) \in [\alpha,\chi]_z\}$$

Die *maximale T-Adresse* α mit mit der Sortierdimension *k* eines Z-Intervalls $[\alpha,\chi]_z$ ist:

$$\alpha := \max\{T_k(x) \mid Z(x) \in [\alpha,\chi]_z\}$$

Im folgenden Abschnitt wird der Fall der minimalen T-Adresse betrachtet, da der Fall der maximalen T-Adresse leicht abgeleitet werden kann. In 5.7.2.2 wird der Tetris-Algorithmus an Hand eines Beispiels vorgestellt. Dort wird die maximale T-Adresse benutzt. Grundsätzlich bestimmt die Sortierrichtung die Verwendung der maximalen bzw. minimalen T-Adresse. Wird das Ergebnis aufsteigend sortiert, kommt die minimale T-Adresse zum Einsatz, bei absteigender Sortierung entsprechend die maximale T-Adresse.

Zur Berechnung der *minimalen T-Adresse* werden nun zwei Algorithmen vorgestellt.

5.7.1 Z-Würfel

Die Z-Würfel-Methode basiert auf den folgenden zwei Sätzen:

Satz 5-6:

Jede Z-Region lässt sich als eine Menge von Z-Würfeln darstellen.

Beweis:

Ergibt sich sofort aus der Struktur der Z-Adresse.
q.e.d.

Satz 5-7:

Die minimale T-Adresse eines Z-Würfels $[\alpha| c]$ ist genau die T-Adresse der unteren Grenze des Z-Intervalls $[\text{pre}(\alpha,c), \text{pre}(\alpha,c)+ 2^{(s-c)d} -1]$ mit $c \in [0, s-1]$.

Beweis:

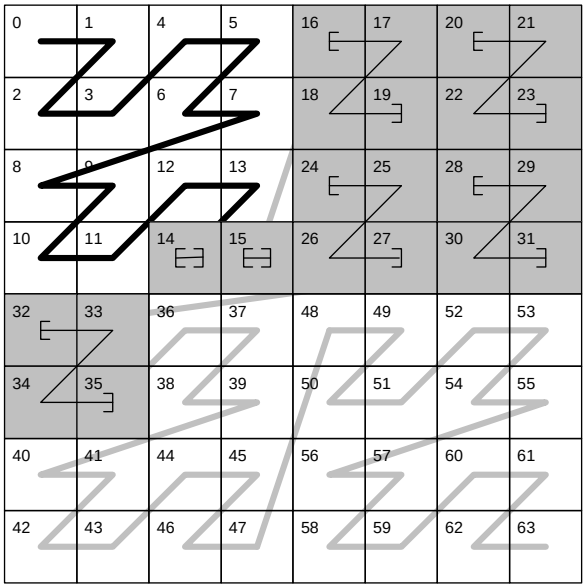
Ergibt sich sofort aus der Struktur des Z-Würfels (siehe Definition 4-6).
q.e.d.

Basierend auf diesen zwei Sätzen, kann man nun den Algorithmus der minimalen T-Adresse wie folgt angeben:

```
Input:   α  :untere Grenze des Z-Intervalls
         β  :obere Grenze des Z-Intervalls
Output:  τ  :minimale T-Adresse des Z-Intervalls

Begin:
      Zerlege die Z-Region[α,β] in eine Menge von Z-Würfeln;
      Berechne für die untere Grenze jedes Z-Würfels die T-Adresse;
      τ =  die kleinste T-Adresse der unteren Grenzen der Z-Würfel;
END
```

Die Zerlegung einer Z-Region in eine Menge von Z-Würfeln wird an Hand von Abbildung 5-9 erläutert.



Z-Würfel	T_{min}
[14,14]	19
[15,15]	27
[16,19]	32
[20,23]	48
[24,27]	34
[28,31]	50
[32,35]	4

Abbildung 5-9

Gegeben ist ein 2-dimensionaler Basisraum mit einer Auflösung von 8 in jeder Dimension. Die Z-Adresse besteht somit aus 3 Stufen. Der Ursprung ist links oben. Die Region [14:35] soll in Z-Würfel zerlegt werden. Der Algorithmus beginnt mit dem Z-Wert 14, die untere Grenze des Z-Intervalls. Der Z-Würfel [14|2] besteht aus dem Z-Intervall [14,14], da *suf*(*14*,2) = 0 ist (siehe Gl: 4-32). Nun wird getestet, ob die obere Grenze des Z-Würfels noch im Intervall der Region liegt. Da 14 kleiner als 35 ist, gehört [14|2] zur Z-Würfelzerlegung der Z-Region [14:35]. Nun geht man zum Bruder. Dazu addiert man auf der Z-Würfel-Ursprungsstufe ZU(14,2) den Wert 1. Man erhält nun den Wert 15. Der Z-Würfel besteht aus dem Intervall [15,15]. Addiert man auf der oberen Grenze des Z-Würfels, also auf den Wert 15, den Wert 1, kommt es auf Stufe 2 zu einem Überlauf, der bis auf Stufe 0 propagiert wird. Kommt es zu einem Überlauf, geht man auf eine niedrigere Stufe, in diesem Fall Stufe 1. Der aktuelle Würfel ist somit [16|1]. Er besteht aus dem Z-Intervall [16,19]. Die obere Grenze des Z-Würfels ist 19 und kleiner als 35. Somit gehört

dieser Würfel zur Ergebnismenge. Somit wird zum Bruder gegangen. Man gelangt zum Bruder, indem man auf Stufe 1 der Z-Adresse 16 den Wert 1 addiert. Betrachten wir hierzu die binäre Darstellung:

01.00.00 Gl: 5-17

Sie ist die binäre Darstellung für den Dezimalwert 16. Addiert man auf Stufe 1 den Wert 1, erhält man den Wert:

01.01.00 Gl: 5-18

Dies ist die untere Grenze des Z-Würfels. Die obere Grenze ergibt sich dadurch, dass alle Bits, die zum Suffix gehören, auf den Binärwert 1 gesetzt werden. Man erhält somit den Wert :

01.01.11 Gl: 5-19

Das entspricht dem Dezimalwert 23. Auch dieser Z-Würfel gehört zur Ergebnismenge.

Wir sind nun beim Z-Würfel [28|1], der durch das Z-Intervall [28,31] repräsentiert wird. Addiert man auf Stufe 1 den Wert 1 kommt es zu einem Überlauf. Nun gehen wir zum Z-Würfel [32|0]. Der Z-Würfel wird durch das Z-Intervall [32,47] repräsentiert. Da 47 größer als 35 ist, gehört dieser Z-Würfel nicht zur Ergebnismenge. Man geht dann wieder eine Stufe tiefer und betrachtet den Z-Würfel [32|1]. Der Z-Würfel wird durch das Z-Intervall [32,35] dargestellt. Da 35 gleich der oberen Grenze der Z-Region [14,35] ist, gehört dieser Z-Würfel zur Ergebnismenge und der Algorithmus terminiert.

Allgemein können bei einem Basisraum Ω mit einer Auflösung von 2^s

$$2(s-1)(2^d-1)+2^d-1 \qquad \text{Gl: 5-20}$$

Z-Würfel entstehen. Die Menge der Z-Würfel wird dann bezüglich der T-Adresse der unteren Grenze sortiert. Die CPU-Komplexität für das Sortieren beträgt O(n log n). *n* ist die Anzahl der Z-Würfel. Die Anzahl der Z-Würfel steigt exponentiell bezüglich der Anzahl der Dimensionen. Die kleinste T-Adresse ist die minimale T-Adresse der Region, in diesem Beispiel 4.

Messungen zeigten, dass der Algorithmus bereits bei 3 Dimensionen CPU-Bound ist.

5.7.2 Lineare Methode

In diesem Abschnitt stellen wir eine lineare Methode zur Berechnung der *minimalen T-Adresse* eines Z-Intervalls vor. Eine Z-Adresse besteht aus *s* Stufen. Somit hat eine *d*-dimensionale *Z*-Adresse folgende Form:

$$\underbrace{\alpha_{d(s-1)+(d-1)}\alpha_{d(s-1)+(d-2)}\ldots\alpha_{d(s-1)}}_{\text{step 0}}.\underbrace{\alpha_{d(s-2)+(d-1)}\alpha_{d(s-2)+(d-2)}\ldots\alpha_{d(s-2)}}_{\text{step 1}}\ldots\underbrace{\alpha_{d-1}\alpha_{d-2,}\ldots\alpha_{0}}_{\text{step s-1}} \qquad \text{Gl: 5-21}$$

Mit $\alpha_{[j]}$ wird die *j*-te Stufe einer Z-Adresse α bezeichnet. Das *i* Bit in der *j* Stufe einer Z-Adresse bezeichnen wir mit $\alpha_{[j],(i)}$. Somit kann eine Z-Adresse wie folgt dargestellt werden:

$$\underbrace{\alpha_{[0],(d-1)}\alpha_{[0],(d-2)}\ldots\alpha_{[0],(0)}}_{\text{step 0}}.\underbrace{\alpha_{[1],(d-1)}\alpha_{[1],(d-2)}\ldots\alpha_{[1],(0)}}_{\text{step 1}}\ldots\underbrace{\alpha_{[s-1],(d-1)}\alpha_{[s-1],(d-2),}\ldots\alpha_{[s-1],(0)}}_{\text{step s-1}} \qquad \text{Gl: 5-22}$$

Um den Dimensionswert x_i aus einer Z-Adresse α zu bestimmen, wird die Funktion *extract*(α,*i*) wie folgt definiert:

Definition 5-5: extract

Gegeben ist eine Z-Adresse α eines *d*-dimensionalen Basisraums Ω. Dann ist x_i = *extract*(α,*i*) der Wert der Dimension *i* mit i ∈ {1,..d}, die zur Z-Adresse beigetragen hat.

$$x_i = \sum_{j=0}^{s-1} \alpha_{[j],i}\, 2^{(s-1)-j}$$

Durch die Tetris-Ordnung wird die Stufe einer Z-Adresse in zwei Teile unterteilt: das extrahierte Bit (*ext*) der Sortierdimension *k* und die um die Sortierdimension reduzierten Bits (*re*), die dann eine reduzierte Z-Adresse bilden. Die reduzierten Bits einer Stufe werden noch weiter unterteilt, in den hochwertigen Teil *reH* und den niederwertigen Teil *reN*. Der Separator zwischen dem hochwertigen und niederwertigen Teil einer Stufe ist *ext*. Das Bit der Sortierdimension *k* ist nicht Bestandteil der reduzierten Teile.

$$.\underbrace{\alpha_{[1],(d-1)}\alpha_{[1],(d-2)}\ldots\alpha_{[1],(k)}}_{reH}\overbrace{\alpha_{[1],(k-1)}}^{ext}\underbrace{\alpha_{[1],(k-2)}\ldots\alpha_{[1],(1)}\alpha_{[1],(0)}}_{reN}$$

Abbildung 5-10: Unterteilung einer Z-Adressen-Stufe

Abbildung 5-10 zeigt die Aufteilung der Stufe 1 in *reH*, *ext* und *reN*. Um auf die einzelnen Bestandteile einer Z-Adresse α zuzugreifen werden folgende Funktionen auf einer *Stufe* einer Z-Adresse definiert:

Definition 5-6:

Gegeben sei die Stufe $u = \alpha_{[i]}$ einer Z-Adresse α eines d-dimensionalen Basisraums Ω. Die Sortierdimension sei k.

$$reH(u,k) = \begin{cases} \sum_{i=k}^{d-1} u_{(i)} 2^i & \text{für } 1 \le k \le d-1 \\ 0 & \text{für } k = d \end{cases}$$

$$ext(u,k) = u_{(k-1)} 2^{k-1} \quad \text{mit } 1 \le k \le d$$

$$reN(u,k) = \begin{cases} \sum_{i=0}^{k-2} u_{(i)} 2^i & \text{für } 2 \le k \le d \\ 0 & \text{für } k = 1 \end{cases}$$

Der Wert einer Stufe ergibt sich somit aus der Addition der drei Komponenten:

$$\alpha_{[i]} = reH(\alpha_{[i]},k) + ext(\alpha_{[i]},k) + reN(\alpha_{[i]},k) \qquad \text{Gl: 5-23}$$

5.7.2.1 Problembeschreibung und Randbedingungen

Nach Definition 5-4 ist die minimale T-Adresse τ eines Z-Intervalls $[\alpha,\beta]_z$, die T-Adresse, die in das Intervall $[\alpha,\beta]_z$ fällt, und aus allen T-Werten, die in $[\alpha,\beta]_z$ fallen, den kleinsten T-Wert besitzt:

$$\tau = \min\{T_k(x) \mid Z(x) \in [\alpha,\beta]_z\} \qquad \text{Gl: 5-24}$$

Für ein Z-Intervall $[\alpha,\beta]_z$ auf einem d-dimensionalen Basisraum Ω mit einer Auflösung r in jeder Dimension ergeben sich folgende Randbedingungen für die Berechnung der minimalen T-Adresse:

- $\alpha \le \beta$
- Die Anzahl der Stufen s ist beliebig, aber endlich
- $\tau = \min\{T_k(x) \mid Z(x) \in [\alpha,\beta]\}$

5.7.2.2 Algorithmus

Betrachtet man die Struktur einer T-Adresse τ, ist offensichtlich, dass die Sortierdimension den größten Einfluss auf den Wert der T-Adresse hat. Die reduzierten Bits werden bezüglich der Z-Ordnung verschränkt. Für die Berechnung der minimalen T-Adresse ist folgender Satz wesentlich:

Satz 5-8

Gegeben ist eine n-stellige Binärzahl α, dann gilt:

$$2^j + \sum_{i=j}^{n-1} \alpha_i \cdot 2^i > \sum_{i=0}^{n-1} \alpha_i \cdot 2^i$$

Beweis:

$$2^j = \sum_{i=0}^{j-1} 2^i + 1 > \sum_{i=0}^{j-1} 2^i \geq \sum_{i=0}^{j-1} \alpha_i 2^i$$

q.e.d

Um also die minimale T-Adresse eines Z-Intervalls zu bestimmen, muss zunächst der Wert der Sortierdimension verkleinert werden und dann der Wert der reduzierten Z-Adresse. Hierbei muss jedoch beachtet werden, dass die T-Adresse im Z-Intervall liegt. Ziel ist es somit eine Stufe zu finden, die nicht zum *Präfix* des Z-Intervalls gehört und in der das Bit der Sortierdimension *k* der unteren Grenze α gesetzt ist. Damit das Bit der Sortierdimension auf 0 gesetzt werden kann, muss man auf das nächst höherwertige Bit der Z-Adresse den Wert 1 addieren. Dadurch entsteht eine Z-Adresse, die größer als α ist (siehe Satz 5-8). Da die minimale T-Adresse gesucht wird, werden alle niederwertigen Bits als das Bit der Sortierdimension, das von 1 auf 0 gesetzt worden ist, auch auf 0 gesetzt. Um zu verhindern, dass die Z-Adresse der minimalen T-Adresse größer als β ist, muss gesichert werden, dass ein höherwertiges Bit als das der Sortierdimension *k*, der aktuellen Stufe, den Wert 0 besitzt. Damit das Bit der Sortierdimension auf 0 gesetzt werden kann, muss man auf das nächst höherwertige Bit der Z-Adresse den Wert 1 addieren. Die Addition wird aber nur auf die reduzierte Z-Adresse ausgeführt, damit der Wert der Sortierdimension nicht beeinflusst wird. Nun besteht jedoch die Möglichkeit, dass es ständig zu Überläufen auf die reduzierte Z-Adresse kommt, so dass der Wert 1 bis in den Teil, der zum Präfix gehört, propagiert wird. Dies wiederum würde dazu führen, das die neue Adresse größer als β wäre und somit nicht mehr Element des Z-Intervalls $[\alpha,\beta]$ wäre. Somit muss noch zugesichert sein, dass ein Bit der reduzierten Z-Adresse, das noch nicht zum Präfix gehört, gesetzt ist.

Dieser Algorithmus wird nun an Hand eines Beispiels verdeutlicht.

Gegeben sei das Z-Intervall = [21,30]. Die Sortierdimension sei $k = 1$. Die minimale T_1-Adresse des Z-Intervalls hat den Z-Adressen Wert 24. Um diesen Wert zu berechnen wird zunächst das Präfix eliminiert.

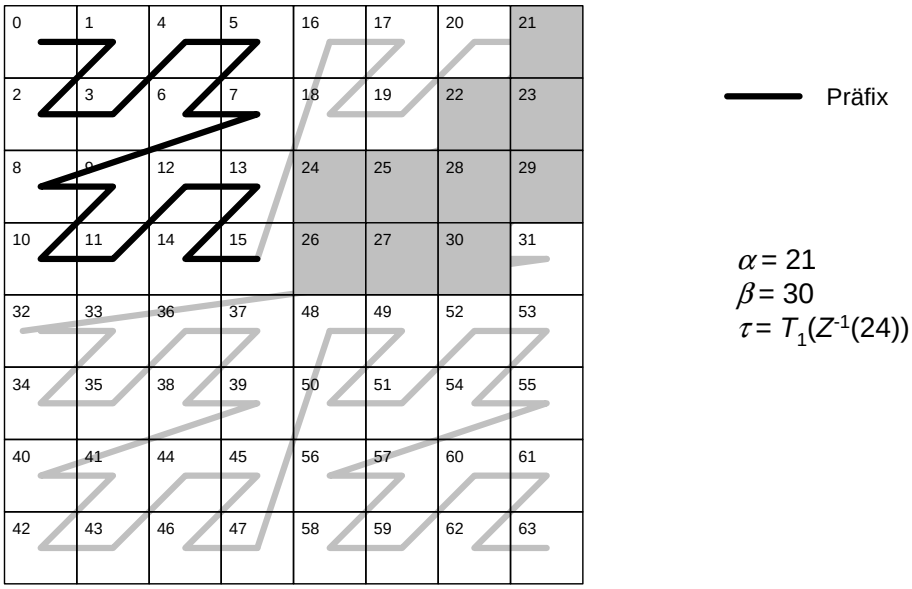


Abbildung 5-11

Zunächst betrachtet man dazu die binäre Darstellung von α und β.:

$$\alpha = 01.01.01$$
$$\beta = 01.11.10 \qquad \text{Gl: 5-25}$$

Das Präfix des Z-Intervalls sind die Stufen, beginnend von Stufe 0, in der α und β gleich sind. Somit ist das Präfix des Z-Intervalls [21,30] die Stufe 0.

$$\alpha = \underbrace{01}_{\text{Präfix}}.01.01$$
$$\beta = \underbrace{01}_{\text{Präfix}}.11.10 \qquad \text{Gl: 5-26}$$

Für den allgemeinen Fall ergibt sich somit:

$$\overbrace{\underbrace{\alpha_{[0],(d-1)}\alpha_{[0],(d-2)}\ldots\alpha_{[0],(0)}}_{\text{step 0}}\ldots\underbrace{\alpha_{[n],(d-1)}\alpha_{[n],(d-2)}\ldots\alpha_{[n],(0)}}_{\text{step n}}\cdots}^{\text{Präfix}} \qquad \text{Gl: 5-27}$$
$$\| \qquad\qquad \|$$
$$\underbrace{\underbrace{\beta_{[0],(d-1)}\beta_{[0],(d-2)}\ldots\beta_{[0],(0)}}_{\text{step 0}}\ldots\underbrace{\beta_{[n],(d-1)}\beta_{[n],(d-2)}\ldots\beta_{[n],(0)}}_{\text{step n}}\cdots}_{\text{Präfix}}$$

d.h, die Stufen 0 bis *n* der Z-Adresse α und β haben den gleichen Wert.

Der nächste Schritt des Algorithmus untersucht die erste Stufe, die nicht mehr zum Präfix des Z-Intervalls gehört. In unserem Beispiel Stufe 1. Man sucht nun den Bereich, in dem sich die Z-Adressen α und β unterscheiden. Es werden hier 3 Fälle unterschieden. **Fall 1** untersucht zunächst, ob

$$reH(\alpha_{[i]},k) < reH(\beta_{[i]},k) \qquad \text{Gl: 5-28}$$

gilt. Ist dies der Fall existiert mindestens ein Bit in $reH(\alpha_{[i]},k)$, das den Wert 0 hat. Somit wird ein Überlauf durch eine Addition auf ein niederwertiges Bit spätestens in diesem

Bereich gestoppt, d.h. das Präfix, kann nicht durch einen Überlauf betroffen werden. Somit ist die Z-Adresse der gesuchten T-Adresse kleiner als β. Nun sucht man die Stufe, in der die Sortierdimension den Wert 1 besitzt. Diese setzt man dann auf 0 und addiert auf das nächst höhere Bit den Wert 1. Die Addition wird jedoch nur auf die reduzierte Z-Adresse durchgeführt. Der Algorithmus terminiert und liefert die minimale T-Adresse.

Ist dies nicht der Fall wird **Fall 2** betrachtet. In Fall 2 wird die Sortierdimension betrachtet. Hier wird getestet ob

$$ext(\alpha_{[i]},k) < ext(\beta_{[i]},k) \qquad \text{Gl: 5-29}$$

gilt.

Ist dies der Fall, ist das Bit der Sortierdimension das erste Bit in dem sich α und β unterscheiden. In dieser Situation könnte es passieren, dass durch eine Addition von 1 auf die reduzierte Z-Adresse ein Überlauf bis in den Bereich des Präfix propagiert wird und somit die T-Adresse nicht mehr in der Region liegt. Somit wird von der aktuellen Stufe in die reduzierte Z-Adresse hinabgestiegen, bis man ein Bit findet, das den Wert 0 besitzt. Ab dieser Stufe wird nun nach einem Bit in der Sortierdimension gesucht, das den Wert 1 besitzt. Ist das Bit gefunden, wird das Bit der Sortierdimension von 1 auf 0 gesetzt und das nächst höherwertige Bit mit dem Wert 1 addiert. Die Addition wird nur auf die reduzierte Z-Adresse ausgeführt.

Ist $ext(\alpha_{[i]},k) < ext(\beta_{[i]},k)$ nicht wahr, muss

$$reN(\alpha_{[i]},k) < reN(\beta_{[i]},k) \qquad \text{Gl: 5-30}$$

gelten. Hierbei handelt es sich um **Fall 3.** In diesen Fall geht man zu Stufe i+1 und sucht ab Stufe i+1 in der Sortierdimension ein Bit mit dem Wert 1. Ist die Stufe gefunden, verfährt man wie in den anderen Fällen.

Betrachten wir wieder unser Beispiel. Es wurde festgestellt, dass das Präfix die Stufe 1 ist.

$$\alpha = \underbrace{01}_{\text{Präfix}} . \underbrace{0}_{reH} 1.01 \qquad \text{Gl: 5-31}$$
$$\beta = \underbrace{01}_{\text{Präfix}} . \underbrace{1}_{reH} 1.10$$

Da $reH(\alpha_{[1]},1) = 0$ und $reH(\beta_{[1]},1) = 1$ ist Fall 1 erfüllt. Es wird nun nach dem ersten Bit in der Sortierdimension k gesucht, das den Wert 1 hat, in unserem Beispiel auch die Stufe 1. Es wird nun das Bit der Sortierdimension auf 0 gesetzt und auf das nächst höherwertige Bit 1 addiert. Die Addition wir nur auf die reduzierte Z-Adresse ausgeführt.

Man erhält:

$$\alpha' = \underbrace{01}_{\text{Präfix}} . \underbrace{1}_{reH} 0.00 \qquad \text{Gl: 5-32}$$
$$\beta = \underbrace{01}_{\text{Präfix}} . \underbrace{1}_{reH} 1.10$$

α' ist somit die Z-Adresse, deren T_1-Adresse $f_1(\alpha')$ die minimale T-Adresse repräsentiert. Wie man sieht ist $\alpha < \beta$.

Abbildung 5-14 zeigt die Prototypimplementierung des Algorithmus zur Berechnung der minimalen T-Adresse.

Im Algorithmus werden folgende Variablen verwendet:

Bezeichnung	Beschreibung
dub	Metabeschreibung des UB-Baums
res	Z-Adresse, die als Ergebnis die minimale T-Adresse als Z-Adresse liefert
lr	Z-Adresse: Untere Grenze der Region
hr	Z-Adresse: Obere Grenze der Region
extrAttr	Sortierdimension
Stepp	aktuelle Stufe
extrStepNull	Stufe, in der die Sortierdimension von 1 auf 0 gesetzt werden kann

Abbildung 5-12: Verwendete Variablen im Algorithmus

Die verwendeten Funktionen im Algorithmus haben folgende Bedeutung:

Bezeichnung	Beschreibung
`DUB_adSteps(dub)`	Gibt die Anzahl der Stufen einer Z-Adresse zurück
`AD_copy(dub, res, lr)`	Kopiert die Z-Adresse lr in die Z-Adresse res
`AD_stepIsEq(dub, lr, hr, step)`	Testet, ob die Z-Adresse lr und hr auf der Stufe step denselben Wert besitzen
`AD_stepReduceHighIsLt(dub, lr, hr, step, extrAttr)`	Testet, ob reH($lr_{[step]}$,extrAttr) < reH($hr_{[step]}$,extrAttr) ist.
`AD_stepGetDim(dub, lr, step, extrAttr)`	Gibt den Bitwert der Z-Adresse lr auf Stufe step der Dimension *extrAttr* zurück
`AD_maxValLow(dub, lr, step, extrAttr)`	Testet, ob die Z-Adresse auf Stufe step reN den maximalen Wert hat
`AD_maxValLow(dub, lr, step, extrAttr)`	Testet, ob die Z-Adresse auf Stufe step reH den maximalen Wert hat
`AD_extractFirstStepNotNull(dub, &extrStepNull, lr, step, extrAttr)`	Gibt die Stufe der Z-Adresse lr, auf der die Sortierdimension den Wert 1 besitzt, zurück. Gestartet wird auf Stufe Step
`AD_stepSet0Dim(dub, res, extrStepNull, extrAttr);`	Setzt das Bit extrAttr der Stufe extrStepNull der Z-Adresse auf 0
`AD_reduceIncStartAtStepSw(dub, res, extrStepNull, extrAttr)`	Addiert auf die reduzierte Z-Adresse der Z-Adresse res auf Stufe extrStepNull den Wert 1 auf rehH
`AD_tailSet0Sw(dub, res, extrStepNull, extrAttr)`	Setzt den tail der Z-Adresse ab Stufe extrStepNull und der Sortierdimension extrAttr auf 0

Abbildung 5-13:Verwendete Funktionen des Algorithmus

```
Status AD_minTet(DUB * dub, AD * res, AD * lr, AD * hr, int extrAttr)
{
   int step = 0;            /*  current step that is processed */
   int extrStepNull = 0;    /*  first significant extract bit
                                that is not null   */

   AD_copy(dub, res, lr);
   /* find the prefix of the region */
   while (  (step < DUB_adSteps(dub)) &&
            (AD_stepIsEq(dub, lr, hr, step) == TRUE))
         step++;

   /*
      test if reduce high of current step hr>lr
    */
   if (AD_stepReduceHighIsLt(dub, lr, hr, step, extrAttr) != TRUE)
   {
      if (AD_stepGetDim(dub, lr, step, extrAttr) !=
      AD_stepGetDim(dub, hr, step, extrAttr))
      {
         if (AD_maxValLow(dub, lr, step, extrAttr) == TRUE)
         {
            step = step + 1;
            while (step < DUB_adSteps(dub))
            {
               if (AD_maxValHigh(dub, lr, step, extrAttr) != TRUE)
                  break;
               if (AD_maxValLow(dub, lr, step, extrAttr) != TRUE)
               {
                  step = step + 1;
                  break;
                  step = step + 1;

               }
            }
         }
         else
         {
         /* rl.l < rh.l */
            step = step + 1;
         }
      }
      if (step < DUB_adSteps(dub))
      {
         /* find the first extract bit that is not null */
         if (AD_extractFirstStepNotNull(dub, &extrStepNull, lr,
             step, extrAttr) == OK)
         {
            AD_stepSet0Dim(dub, res, extrStepNull, extrAttr);
            if (AD_reduceIncStartAtStepSw(dub, res, extrStepNull,
                  extrAttr) != OK)
               RETURNERROR;
            if (AD_tailSet0Sw(dub, res, extrStepNull, extrAttr) != OK)
                  RETURNERROR;
         }
      }
   return OK;
}
```

Abbildung 5-14: Algorithmus zur Berechnung der minimalen T-Adresse

Die Komplexität des Algorithmus ist O(n), wobei *n* die Anzahl der Bits bezeichnet, aus denen die Z-Adresse besteht. Somit ist die Berechnung unabhängig von der Anzahl der Dimensionen einer Z-Adresse.

5.8 Algorithmus

Um den Namen des Algorithmus zu verdeutlichen, werden die Daten in diesem Abschnitt in absteigender Reihenfolge nach Dimension 2 gelesen. Somit werden die Regionen von unten nach oben in T-Ordnung aus dem Sekundärspeicher geladen. Dies ähnelt dem Tetris-Spiel, das als Name Parte stand. Somit wird in diesem Beispiel die maximale T_2-Adresse jeweils berechnet.

Die folgenden Abbildungen zeigen die wesentlichen Schritte des Tetris-Algorithmus. Jede Abbildung besteht aus drei Bereichen. Links ist der Algorithmus, rechts oben die Regionszerlegung Θ des 2-dimensionalen UB-Baums, rechts unten der entsprechende UB-Baum als Baumstruktur abgebildet. Die Blätter der Baumstruktur enthalten die entsprechenden Regionen.

Im Algorithmus werden folgende Variablen verwendet:

Bezeichnung	Beschreibung
UB	UB-Baum
τ	Z-Adresse mit dem maximalen Tetris-Wert *phi*.
α,β	Z-Adressen der aktuellen Region ρ = [α:β]
pg	Seiten der aktuellen Region ρ
`phi[]`	Feld (array) von Z-Intervallen, die den noch nicht abgearbeiteten Anfragebereich Q abdecken.
`k`	Sortierdimension
`ql`	untere Grenze des Anfragebereichs
`qh`	obere Grenze des Anfragebereichs
`cache[]`	Feld von Tupeln, die noch kleiner oder gleich als die Sweepebene sind.
`sweep`	aktuelle Position der Sweepebene bezüglich der Dimension *k*.

Abbildung 5-15: Verwendete Variablen im Tetris-Algorithmus

Die verwendeten Funktionen im Algorithmus haben folgende Bedeutung:

Bezeichnung	Beschreibung
`calcZ-IntervalSet(ql,qh)`	Berechnet eine Menge von Z-Intervallen, deren Bild den Anfragebereich [[ql,qh]] vollständig abdeckt.
`getRegionSeperators(UB,τ,α,β)`	Gibt die Grenzen der Region [α:β] aus dem UB-Baum UB zurück, für die $\alpha \leq \tau \leq \beta$ gilt.
`GetPage(UB,`β`)`	Gibt die entsprechende Seite zurück, deren Regions-separator β ist
`PutMatchingTuplesToCache (cache,pg,ql,qh)`	Filterfunktion, die alle Tupel in den Cache einfügt, die im Anfragebereich [[ql,qh]] liegen.
`updatePhi(phi,α,β)`	Reduziert die Regionsmenge *phi* um die Region [α:β]
`nextEvent(phi,max,k)`	Gibt die Z-Adresse τ mit dem größten Tetris-Wert von *phi* zurück. Sie entspricht der T-Adresse τ_k (Eventpunkt)
`Extract(τ,k)`	Berechnet den Wert x_k aus der Z-Adresse τ
`isempty(phi)`	Testet, ob phi leer ist.
`outputSortedTuples(cache, k,sweep)`	Gibt alle Tupel sortiert nach Dimension k zurück, die größer als die Sweepline sind
`outputSorted(cache,k)`	Gibt alle Tupel, die im Cache enthalten sind, sortiert nach Dimension *k* zurück.

$z(x)$	Berechnet aus dem Tupel x die Z-Adresse

Abbildung 5-16:Verwendete Funktionen des Tetris-Algorithmus

```
TETRIS(UB-Tree UB,Tuple ql, Tuple qh, int k)
{
   Z-Address τ,α,β  = Z(qh); ;
   Page pg = {};
   Z-Interval phi [] = calcZ-IntervalSet(qh,ql);
   Tupel cache[] = {};
   Dim sweep = qh_k;

   while (1)
   {
      getRegionSeparators(UB, τ, α,β);
      pg = getPage(UB,β);
      putMatchingTuplesToCache(cache,pg,ql,qh);
      updatePhi(phi, α,β )
      if(isempty(phi))
            break;
     τ  = nextEvent(phi,max,k);
      if(extract( τ ,k) < sweep)
      {
          sweep = extract(τ,k);
          outputSortedTuples(cache, k,sweep);
      }
  } /*end while*/
  outputSorted(cache,k);
  return ;
}
```

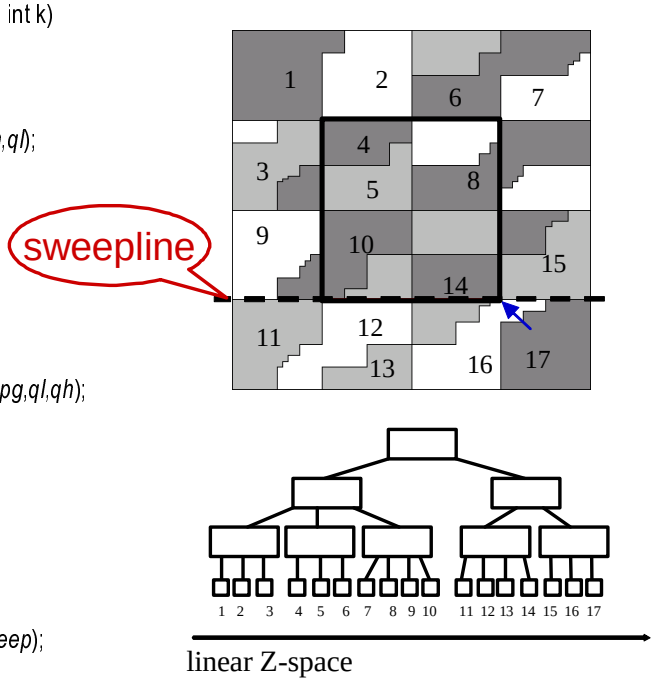


Abbildung 5-17: Tetris-Algorithmus

Die Eingabeparameter für den Tetris-Algorithmus sind der Anfragebereich [[*ql*,*qh*], die Sortierdimension *k* und der UB-Baum, der die Tabelle R speichert. Die Zerlegung Θ des Basisraums Ω für die 2-dimensionale Relation *R* ist in der rechten oberen sowie in der rechten unteren Hälfte als Baumstruktur dargestellt.

```
TETRIS(UB-Tree UB,Tuple ql, Tuple qh, int k)
{
   Z-Address τ,α,β  = Z(qh); ;
   Page pg = {};
   Z-Interval phi [] = calcZ-IntervalSet(qh,ql);
   Tupel cache[] = {};
   Dim sweep = qh_k;

   while (1)
   {
     getRegionSeparators(UB,τ, α,β);
     pg = getPage(UB,β);
     putMatchingTuplesToCache(cache,pg,ql,qh);
     updatePhi(phi, α,β )
     if(isempty(phi))
          break;
     τ = nextEvent(phi,max,k);
     if(extract(τ,k) < sweep)
     {
         sweep = extract(τ,k);
         outputSortedTuples(cache, k,sweep);
     }
  } /*end while*/
  outputSorted(cache,k);
  return ;
}
```

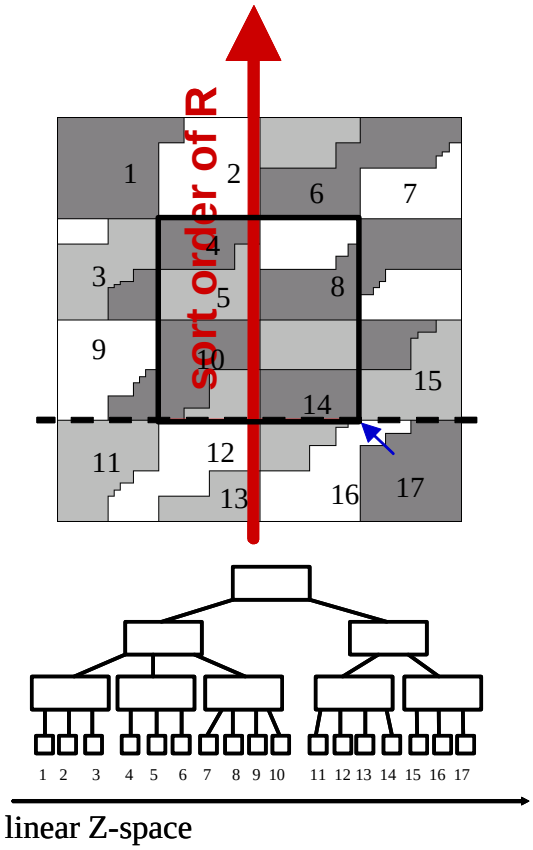


Abbildung 5-18

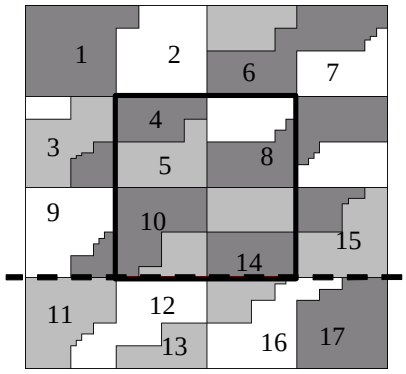
Die Daten der Relation werden bezüglich Dimension A2 sortiert. Die entsprechende SQL-Anfrage, die durch den Tetris-Algorithmus verarbeitet wird, lautet somit:

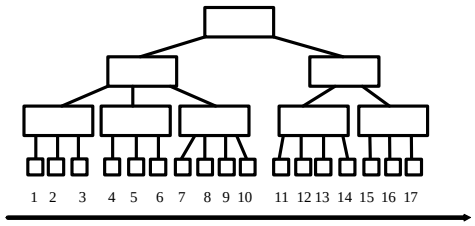
```
SELECT A2 , A1 FROM R
WHERE A1 BETWEEN ql1 AND qh1 AND
          A2 BETWEEN ql2 AND qh2
ORDER BY A2 DESCENDING
```

Abbildung 5-19: ORDER-BY-Klausel

```
TETRIS(UB-Tree UB,Tuple ql, Tuple qh, int k)
{
  Z-Address τ,α,β  = Z(qh); ;
  Page pg = {};
  Z-Interval phi [] = calcZ-IntervalSet(qh,ql);
  Tupel cache[] = {};
  Dim sweep = qh_k;

  while (1)
  {
    getRegionSeparators(UB, τ, α,β);
    pg = getPage(UB,β);
    putMatchingTuplesToCache(cache,pg,ql,qh);
    updatePhi(phi, α,β )
    if(isempty(phi))
        break;
    τ = nextEvent(phi,max,k);
    if(extract(τ,k) < sweep)
    {
      sweep = extract(τ,k);
      outputSortedTuples(cache, k,sweep);
    }
  } /*end while*/
  outputSorted(cache,k);
  return ;
```

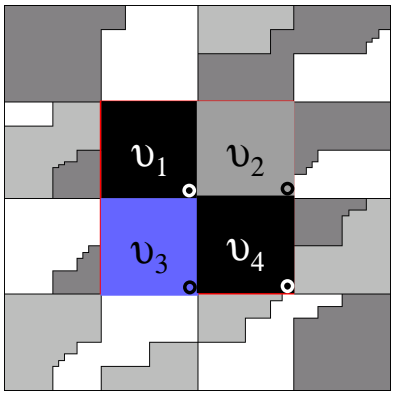


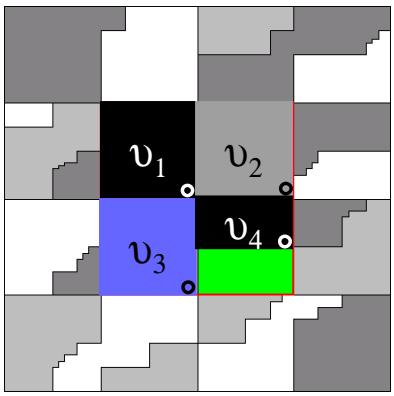


linear Z-space

Abbildung 5-20

Der Tetris-Algorithmus liest die Daten der Relation in diesem Beispiel von unten nach oben (siehe Abbildung 5-18, rechts). Die Z-Adresse τ, die die maximale Tetris-Adresse $f_2(\tau)$ repräsentiert, wird zunächst auf Z(*qh*) und die Sweepline *sweep* = qh_2 gesetzt. Der noch zu lesende Anfragebereich [[*ql*,*qh*]] wird in Z-Intervalle zerlegt und in *phi* gespeichert (siehe Abbildung 5-20). Es entstehen vier Z-Intervalle, die eindeutig auf die vier Z-Unterräume $\{\upsilon_1,\upsilon_2,\upsilon_3,\upsilon_4\}$ über Z^{-1} abgebildet werden und den Anfragebereich [[*ql*,*qh*]] vollständig abdecken (siehe Abbildung 5-21 links). Der Cache ist zunächst leer.





- Maximale Tetris-Adresse

Abbildung 5-21

Zunächst wird die Region [α:β], die Z(qh) = τ enthält ($\alpha \leq \tau \leq \beta$), bestimmt. Da in der Datenbank eine Region durch ihre obere Grenze eindeutig bestimmt wird (siehe Kapitel 3.6.4), lädt `getPage`(UB,β) die entsprechende Seite in den Arbeitsspeicher (siehe Abbildung 5-22 links). Hierbei handelt es sich um Region 14. Dabei wird die Baumstruktur des UB-Baums bis zur Blattebene durchlaufen (siehe Abbildung 5-22 rechts).

```
TETRIS(UB-Tree UB,Tuple ql, Tuple qh, int k)
{
   Z-Address τ,α,β  = Z(qh); ;
   Page pg = {};
   Z-Interval phi [] = calcZ-IntervalSet(qh,ql);
   Tupel cache[] = {};
   Dim sweep = qh_k;

   while (1)
   {
     getRegionSeparators(UB, τ, α,β);
     pg = getPage(UB,β);
     putMatchingTuplesToCache(cache,pg,ql,qh);
     updatePhi(phi, α,β )
     if(isempty(phi))
          break;
     τ  = nextEvent(phi,max,k);
     if(extract( τ ,k) < sweep)
     {
         sweep = extract(τ,k);
         outputSortedTuples(cache, k,sweep);
     }
  } /*end while*/
  outputSorted(cache,k);
  return ;
```

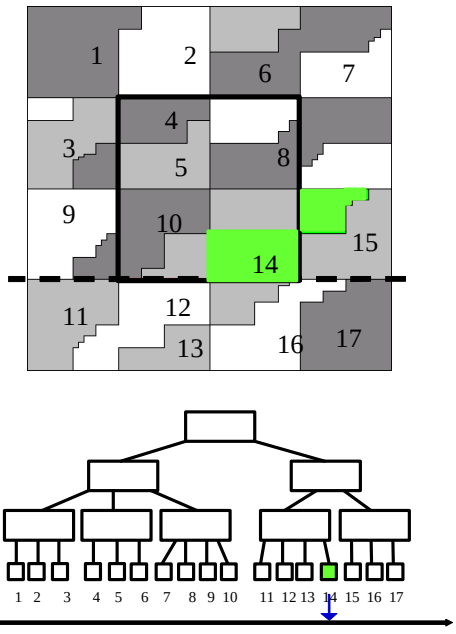


Abbildung 5-22

Danach werden die Tupel der Seite *pg*, die innerhalb des Anfragebereichs [[*ql*,*qh*]] liegen, in das Feld `cache` abgelegt (siehe Abbildung 5-23). Das Feld `cache` wird im Prototyp als Prioritäts-Heap nach Sortierdimension *k* mit dem größten Wert in Dimension *k* an der Wurzel [Knu98v3] realisiert, d.h. es kann das größte Element mit $O(1)$ Zeit entfernt werden. Jedoch muss danach die Heap-Eigenschaft wieder erzeugt werden. Dies schlägt mit $O(\log n)$ zu Buche. Das Einfügen eines neuen Datensatzes benötigt somit $O(\log n)$. n bezeichnet die Anzahl der Tupel, die in einer Scheibe enthalten sind.

```
TETRIS(UB-Tree UB,Tuple ql, Tuple qh, int k)
{
  Z-Address τ,α,β  = Z(qh); ;
  Page pg = {};
  Z-Interval phi [] = calcZ-IntervalSet(qh,ql);
  Tupel cache[] = {};
  Dim sweep = qh_k;

  while (1)
  {
    getRegionSeparators(UB, τ, α,β);
    pg = getPage(UB,β);
    putMatchingTuplesToCache(cache,pg,ql,qh);
    updatePhi(phi, α,β )
    if(isempty(phi))
        break;
    τ = nextEvent(phi,max,k);
    if(extract(τ,k) < sweep)
    {
      sweep = extract(τ,k);
      outputSortedTuples(cache, k,sweep);
    }
  } /*end while*/
  outputSorted(cache,k);
  return ;
```

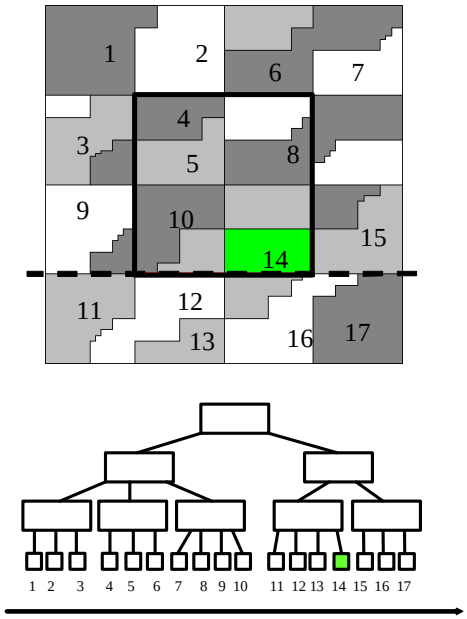


Abbildung 5-23

Die Regionen werden entsprechend der Tetris-Ordnung T_2 gelesen (siehe Abbildung 5-2, links), d.h. sortiert nach Dimension 2. Da es sich in diesem Beispiel um eine absteigende Sortierung handelt, werden die Regionen in absteigender T_2-Ordnung aus dem UB-Baum UB gelesen.

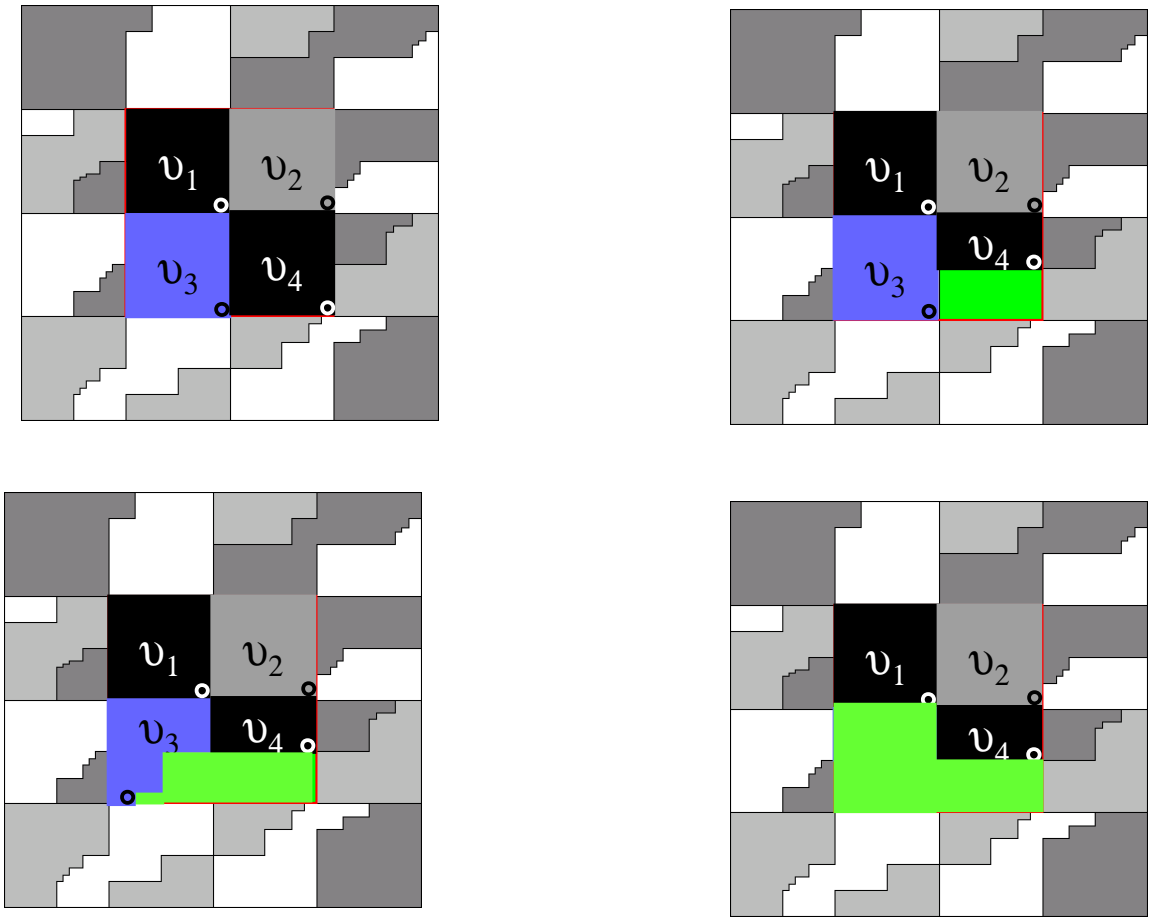


Abbildung 5-24

Um die nächste Region entsprechend der T_2-Ordnung in den Arbeitsspeicher zu laden, wird aus dem noch nicht verarbeiteten Anfragebereich $Z^{-1}(\Phi)$ (Kurzschreibweise für Gl: 5-1), der aus der Menge von Z-Unterräumen $\{\upsilon_1,\upsilon_2,\upsilon_3,\upsilon_4\}$ besteht, die maximale Tetris-Adresse bestimmt. Dies wird dadurch erreicht, dass für jeden Z-Unterraum υ die maximale Tetris-Adresse bestimmt wird (siehe Abbildung 5-24, rechts). Die Z-Adresse τ, die die größte Tetris-Adresse $\tau_2 = f_2(\tau)$ repäsentiert (der Eventpunkt), wird zurückgegeben. Ist also eine Seite *pg* aus dem UB-Baum gelesen worden die zur *Region* ρ gehört, wird die Region ρ aus den nicht verarbeiteten Anfragebereich $Z^{-1}(\Phi)$ geschnitten. Da die Region ρ nach Definition 3-23 ein Teilraum vom Baisraum Ω sowie $Z^{-1}(\Phi)$ ein Teilraum vom Basisraum Ω ist, kann dies formal durch die Differenzmenge zwischen $Z^{-1}(\Phi)$ und ρ wie folgt beschrieben werden:

$$Z^{-1}(\Phi') = Z^{-1}(\Phi) \setminus \rho \qquad \text{Gl: 5-33}$$

In diesem Beispiel wird also der Z-Unterraum υ_4 durch die Region ρ verkleinert, so dass eine neue maximale Tetris-Adresse für υ_4 berechnet werden muss. Dies wird durch die Prozedur `updatePhi(phi,α,β)`bewerkstelligt (siehe Abbildung 5-24). Die Prozedur

`updatePhi`(phi,α,β) modifiziert alle Z-Intervalle in Φ, die das Z-Intervall der eben gelesenen Region schneiden. Der Index wird entsprechend angepasst.

```
TETRIS(UB-Tree UB,Tuple ql, Tuple qh, int k)
{
  Z-Address τ,α,β  = Z(qh); ;
  Page pg = {};
  Z-Interval phi [] = calcZ-IntervalSet(qh,ql);
  Tupel cache[] = {};
  Dim sweep = qh_k;

  while (1)
  {
    getRegionSeparators(UB, τ, α,β);
    pg = getPage(UB,β);
    putMatchingTuplesToCache(cache,pg,ql,qh);
    updatePhi(phi, α,β )
    if(isempty(phi))
        break;
    τ = nextEvent(phi,max,k);
    if(extract( τ,k) < sweep)
    {
       sweep = extract(τ,k);
       outputSortedTuples(cache, k,sweep);
    }
  } /*end while*/
  outputSorted(cache,k);
  return ;
```

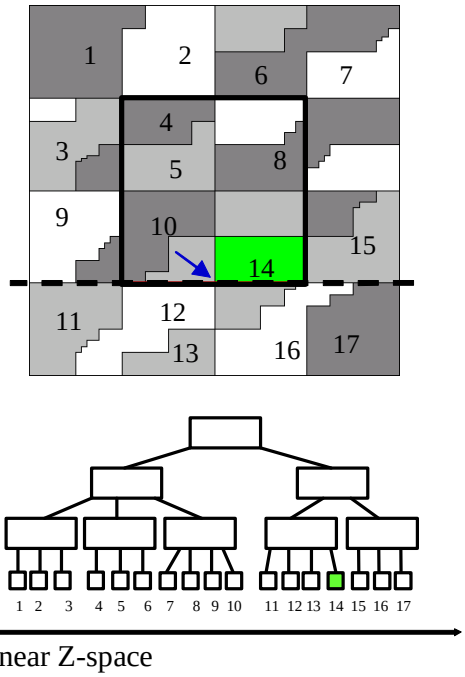


Abbildung 5-25

Mit der Funktion `isempty(phi)` wird getestet, ob noch ein Z-Intervall in `phi` enthalten ist (siehe Abbildung 5-25). Ist `phi` die leere Menge, ist der Anfragebereich vollständig verarbeitet worden und die while-Schleife wird abgebrochen. Die bereits im Cache enthaltenen Tupel werden sortiert nach der Sortier-Dimension ausgegeben, dann terminiert der Algorithmus. Da `phi` in diesem Beispiel noch vier Z-Intervalle enthält und somit noch nicht der ganze Anfrageraum verarbeitet worden ist, wird nun der nächste Eventpunkt berechnet. Hierzu wird der Z-Adresse τ, die Z-Adresse der maximalen T-Adresse $\tau_k = f_2(\tau)$ aus Φ (Eventpunkt), mit der Prozedur `nextEvent(phi,max,k)` zugewiesen. Hierbei handelt es sich um die T-Adresse von υ_3 (siehe Abbildung 5-24), die τ_κ entspricht. Allgemein wird somit für jeden Unterraum υ von Φ die maximale T-Adresse τ_κ berechnet. Die entsprechenden Z-Adressen $\tau = f^{-1}_k(\tau_k)$ stellen potentielle Eventpunkte [8]dar.

[8] Die Z-Adresse τ, die die maximale T-Adresse τ_k repräsentiert, muss nicht mit der oberen Grenze des Z-Intervalls $[\alpha,\beta]$ des Unterraums $\upsilon = [\alpha::\beta]$ übereinstimmen.

```
TETRIS(UB-Tree UB,Tuple ql, Tuple qh, int k)
{
  Z-Address τ,α,β  = Z(qh); ;
  Page pg = {};
  Z-Interval phi [] = calcZ-IntervalSet(qh,ql);
  Tupel cache[] = {};
  Dim sweep = qh_k;

  while (1)
  {
    getRegionSeparators(UB, τ, α,β);
    pg = getPage(UB,β);
    putMatchingTuplesToCache(cache,pg,ql,qh);
    updatePhi(phi, α,β )
    if(isempty(phi))
        break;
    τ = nextEvent(phi,max,k);
    if(extract( τ,k) < sweep)
    {
      sweep = extract(τ,k);
      outputSortedTuples(cache, k,sweep);
    }
  } /*end while*/
  outputSorted(cache,k);
  return ;
```

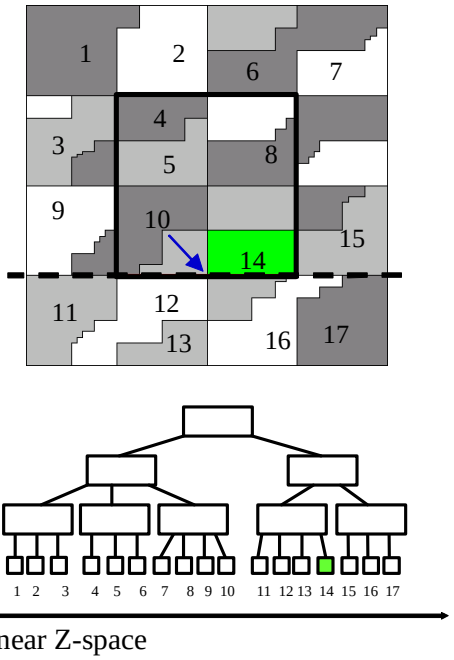


Abbildung 5-26

Als nächster Schritt wird getestet, ob eine Scheibe bereits vollständig abgearbeitet worden ist. Hierzu wird die aktuelle Position der Sweep-Hyperebene mit der aktuellen Position τ verglichen. Da die T-Adressen eine totale Ordnung auf dem Raum erzeugen, trennt τ den Raum in zwei Bereiche, das T-Intervall $[T_k(ql),T_k(Z^{-1}(\tau)]$ und $]T_k(Z^{-1}(\tau),T_k(qh)]$. Das offene Intervall $]T(Z^{-1}(\sigma)T_k(qh)]$ deckt den Bereich ab, der vollständig verarbeitet worden ist. Somit können keine neuen Daten mehr hinzukommen.

Das T-Intervall $[T_k(ql),T_k(Z^{-1}(\tau)]$ ist der Bereich, der teilweise noch nicht vollständig verarbeitet worden ist und somit noch nicht an den Verbraucher weitergereicht werden kann. Da die T-Ordnung parallel zur Sweep-Hyperebene verläuft, muss nur getestet werden, ob $Z^{-1}(\tau)$ in der Sweep-Hyperebene liegt. Gilt:

$$extract(\tau, k) < sweep \qquad \text{Gl: 5-34}$$

enthält die Sweep-Hyperebene $\Psi_{k,sweep}$ (siehe Definition 4-2) $Z^{-1}(\tau)$ nicht. Das bedeutet, die Sweep-Hyperebene $\Psi_{k,sweep}$ kann verschoben werden, d.h. eine neue Scheibe ist vollständig verarbeitet worden und kann sortiert an den Aufrufer weitergereicht werden.

```
TETRIS(UB-Tree UB,Tuple ql, Tuple qh, int k)
{
  Z-Address τ,α,β  = Z(qh); ;
  Page pg = {};
  Z-Interval phi [] = calcZ-IntervalSet(qh,ql);
  Tupel cache[] = {};
  Dim sweep = qh_k;

  while (1)
  {
    getRegionSeparators(UB, τ, α,β);
    pg = getPage(UB,β);
    putMatchingTuplesToCache(cache,pg,ql,qh);
    updatePhi(phi, α,β )
    if(isempty(phi))
        break;
    τ = nextEvent(phi,max,k);
    if(extract(τ,k) < sweep)
    {
      sweep = extract(τ,k);
      outputSortedTuples(cache, k,sweep);
    }
  } /*end while*/
  outputSorted(cache,k);
  return ;
}
```

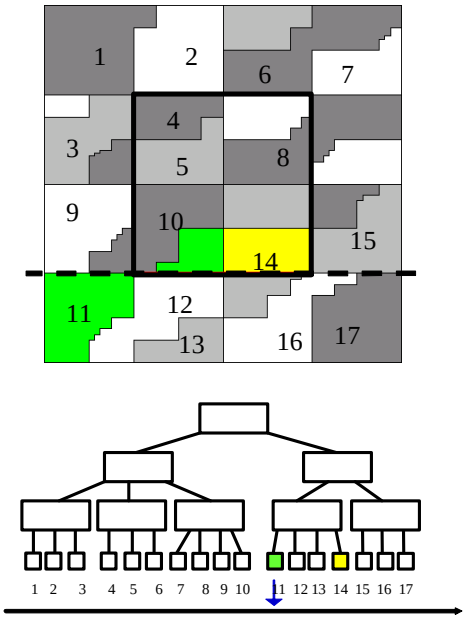


Abbildung 5-27

In dem Beispiel ist es noch nicht der Fall, so dass die nächste Region (Region 11), die den aktuellen Eventpunkt enthält, aus dem UB-Baum gelesen wird (siehe Abbildung 5-27). Um die Scheibe zu vervollständigen muss auch noch Region 10 geladen werden.

Man betrachtet nun die Situation, dass alle Regionen, die die erste Schicht bzw. Scheibe repräsentieren, in den Arbeitsspeicher geladen worden sind (siehe Abbildung 5-28). Da der nächste Eventpunkt nicht mehr auf der Sweep-Hyperebene liegt (siehe Abbildung 5-28, links Pfeil), ist die Bedingung $extract(\tau, k) < sweep$ erfüllt.

Die Funktion *extract*(τ,*k*) gibt den Dimensionswert der Dimension *k* der Z-Adresse τ zurück (siehe Definition 5-5). Die Hyperebene kann dann auf

$$sweep = extract(\tau, k) \qquad \text{Gl: 5-35}$$

verschoben werden.

```
TETRIS(UB-Tree UB,Tuple ql, Tuple qh, int k)
{
  Z-Address τ,α,β  = Z(qh); ;
  Page pg = {};
  Z-Interval phi [] = calcZ-IntervalSet(qh,ql);
  Tupel cache[] = {};
  Dim sweep = qh_k;

  while (1)
  {
    getRegionSeparators(UB, τ, α,β);
    pg = getPage(UB,β);
    putMatchingTuplesToCache(cache,pg,ql,qh);
    updatePhi(phi, α,β )
    if(isempty(phi))
        break;
    τ = nextEvent(phi,max,k);
    if(extract(τ,k) < sweep)
    {
      sweep = extract(τ,k);
      outputSortedTuples(cache, k,sweep);
    }
  } /*end while*/
  outputSorted(cache,k);
  return ;
```

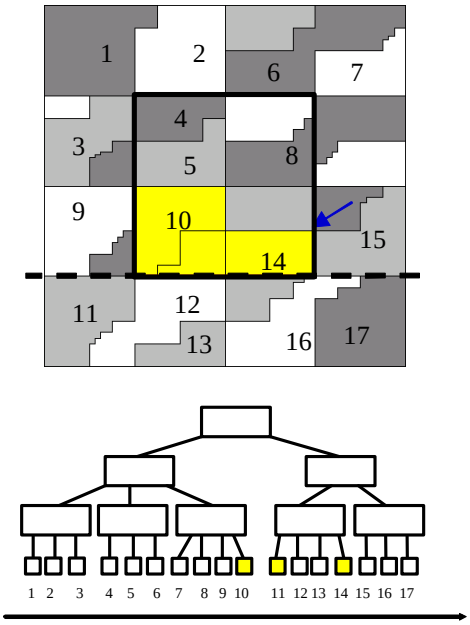


Abbildung 5-28

Die dadurch entstehende neue Schicht S^{-r}(siehe Definition 4-3) kann bereits als Teilergebnis weiterverarbeitet werden. Da im Cache nicht der Raum, sondern die Tupel gespeichert werden, wird die entsprechende Scheibe S^{-t} (siehe Definition 4-4) verarbeitet. Da der Cache als Heap organisiert ist, wird solange das Element an der Wurzel des Heap entfernt, bis der Wert der Sortierdimension des Tupels an der Wurzel des Heaps gleich dem der Sweep-Hyperebene ist (siehe Abbildung 5-29), da die Sweep-Hyperebene nicht zum statischen Bereich gehört.

```
TETRIS(UB-Tree UB,Tuple ql, Tuple qh, int k)
{
  Z-Address τ,α,β  = Z(qh); ;
  Page pg = {};
  Z-Interval phi [] = calcZ-IntervalSet(qh,ql);
  Tupel cache[] = {};
  Dim sweep = qh_k;

  while (1)
  {
    getRegionSeparators(UB, τ, α,β);
    pg = getPage(UB,β);
    putMatchingTuplesToCache(cache,pg,ql,qh);
    updatePhi(phi, α,β )
    if(isempty(phi))
        break;
    τ = nextEvent(phi,max,k);
    if(extract( τ,k) < sweep)
    {
      sweep = extract(τ,k);
      outputSortedTuples(cache, k,sweep);
    }
  } /*end while*/
  outputSorted(cache,k);
  return ;
```

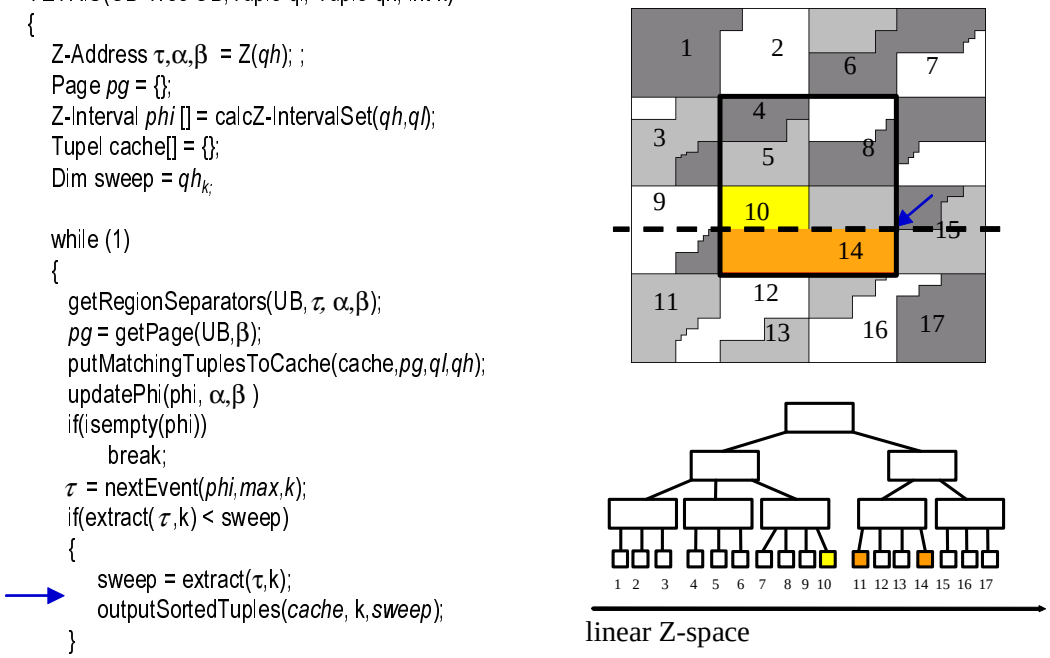


Abbildung 5-29

Ist die letzte Region [α:β] aus dem UB-Baum geladen worden, die den Anfragebereich schneidet, wird durch die Funktion `updatePhi(phi,α,β)` die Menge *phi* leer. Die WHILE-Schleife terminiert und die restlichen Tupel, die noch im Cache sind, werden sortiert ausgegeben.

5.9 Verwaltung von Phi

Ohne Beschränkung der Allgemeinheit wird in diesem Abschnitt wieder ***nur die minimale T-Adresse betrachtet***. Der Eventpunkt ist ***somit die minimale T-Adresse*** von Φ.

Die Datenstruktur, die Φ verwaltet, muss zwei Operationen effizient unterstützen:

- Finde den nächsten Eventpunkt, also die kleinste T-Adresse, die in Φ enthalten ist.
- Schneide das Z-Intervall, das einer Region entspricht, aus Φ aus.

Formal handelt es sich beim Herausschneiden um die Differenzmenge von Φ \ {[χ,δ]}. Hier muss jedoch betont werden, dass die Differenzbildung auf Z-Adressen basiert, die in die jeweiligen Z-Intervalle fallen. Abbildung 5-30 zeigt zunächst die Definition der Differenzbildung auf Z-Intervallen.

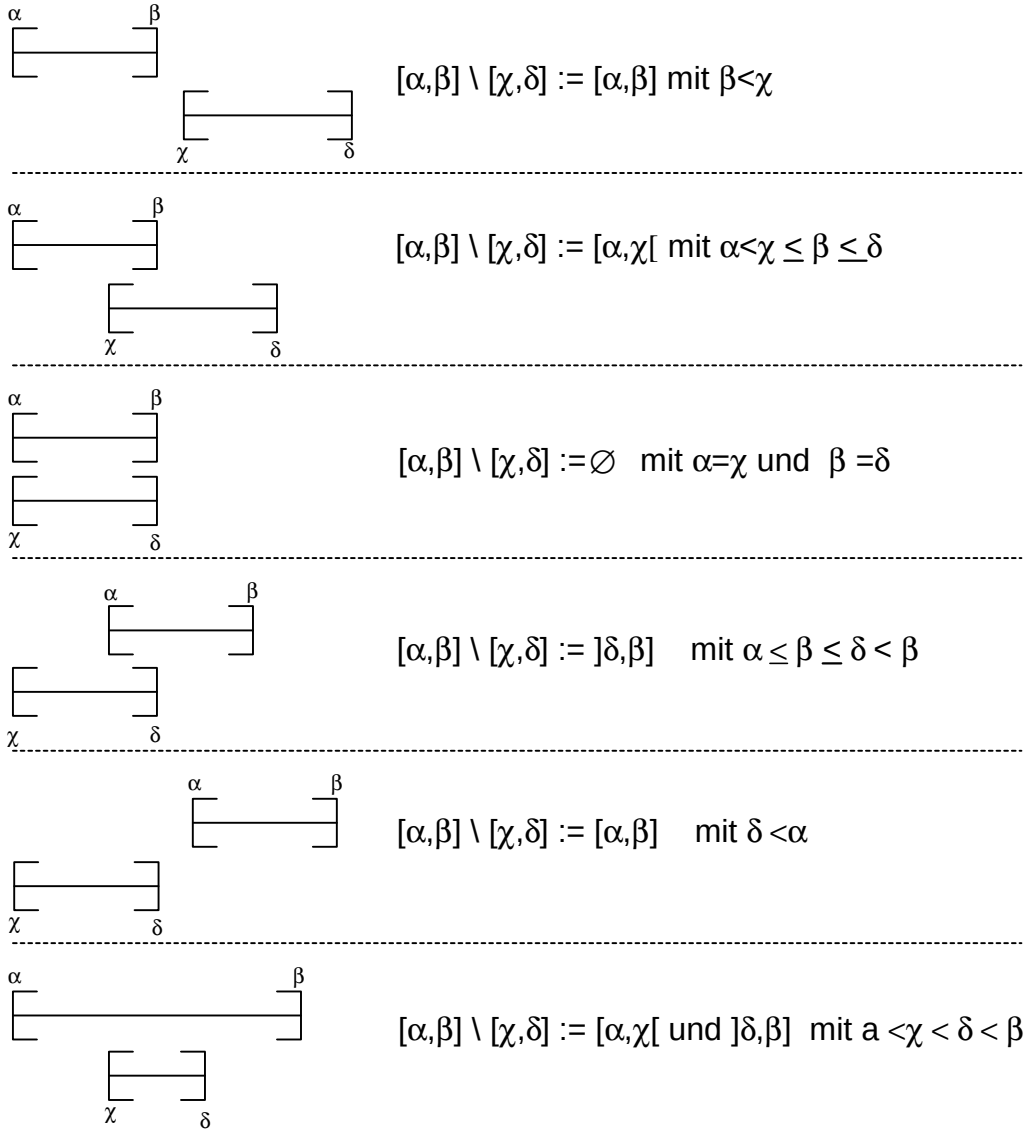


Abbildung 5-30: Differenz von Z-Intervalle

Formal kann nun die Differenzmenge $\Phi \setminus \{[\chi,\delta]\}$ wie folgt beschrieben werden:

$$\Phi \setminus \{[\chi,\delta]\} := \{[\alpha,\beta] \setminus [\chi,\delta] \mid [\alpha,\beta] \in \Phi\} \qquad \text{Gl: 5-36}$$

Ein Beispiel ist in Abbildung 5-31 zu finden.

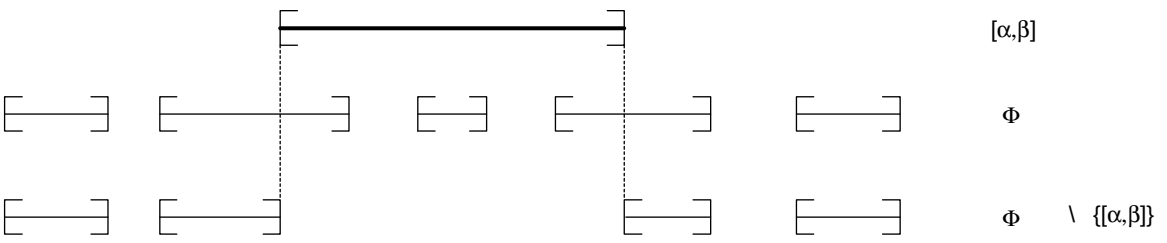


Abbildung 5-31: Differenzmenge auf Z-Intervalle

Die oben genannten Anforderungen führen zu folgender Datenstruktur für die Elemente in Φ. Der nicht verarbeitete Raum wird als Menge von Z-Intervallen abgebildet, da jedem Z-Intervall ein Z-Unterraum zugeordnet werden kann. Die Vereinigung der Z-Unterräume ergibt genau den nicht verarbeiteten Anfrageraum *Q*. Da die Z-Intervalle, die den Anfragebereich *Q* abdecken, nicht unbedingt auf der Z-Kurve zusammenhängen, müssen jeweils die obere und untere Grenze in die Index-Struktur abgelegt werden. Für den Zugriff auf die Z-Intervalle wird somit ein Index, den man als Z-Index bezeichnet, eingeführt. Für jedes Z-Intervall in Φ wird die minimale T-Adresse bestimmt, die auch in einem Index, den T-Index, aufgenommen wird. Der Z-Index ist nach der Z-Ordnung, der T-Index nach der Tetris-Ordnung sortiert. Ein Z-Intervall wird als PHI-Element bezeichnet.

```
struct PHIEL_S
{
    AD low;           /** Untere Grenze des Z-Intervalls*/
    AD high;          /** Obere Grenze des Z-Intervalls */
    AD tvalue;        /** minimale T-Adresse des Z-Intervalls */

    AD event;         /** Eventpunkt: entspricht der minimalen T-Adresse.
                           Ist jedoch als Z-Adresse gespeichert */

};
```

Abbildung 5-32: PHI-Element

Abbildung 5-32 zeigt die Struktur des PHI-Elements der Prototyp-Implementierung in C. Aus Gründen der Effizienz wurde die minimale T-Adresse sowohl als T-Adresse als auch als Z-Adresse abgelegt.

Das Entfernen eines Z-Intervalls $[\alpha,\beta]$ aus der Menge von Z-Intervallen Φ hat folgende Randbedingungen:

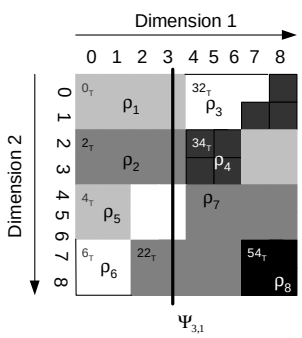
- Es handelt sich um ein 1-dimensionales Problem
- Das Z-Intervall kann *n* Z-Intervalle aus Φ schneiden (siehe Abbildung 5-31).

Um die Differenzmengen-Operation zu realisieren, muss die Such-, Einfügen- und Lösch-Operation auf Z-Intervalle effizient unterstützt werden. Es handelt sich somit um eine klassische Bereichsanfrage im 1-dimensionalen Raum.

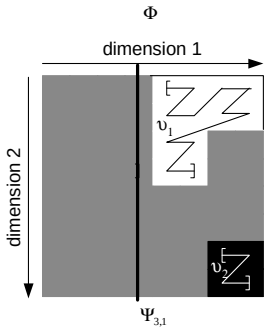
Hierfür ist eine baumstrukturierte Speicherorganisation besonders geeignet, da sowohl ein wahlfreier Zugriff als auch ein sequenzieller Zugriff in Sortierreihenfolge (Z-Ordnung) effizient möglich ist. Da eine Hashfunktion [OttW96] nicht ordnungserhaltend ist, können nur Punktanfragen effizient unterstützt werden. Sie scheidet somit aus. Wird als Datenstruktur ein balancierter Baum eingesetzt, können die Grundoperationen Suchen, Löschen, Einfügen und Aktualisieren in $O(\log_2 n)$ Operationen durchgeführt werden. Eine mögliche Lösung besteht im Einsatz einer 1-dimensionalen Datenstruktur, wie z.B. der AVL-Baum [AdeL62], ein balancierter binärer Baum. Die innovative Grundidee besteht darin, eine Degenerierung des Suchbaums durch eine Höhendifferenzforderung der Teilbäume eines jeden Knotens zu verhindern. Diese Methode benötigt nur zwei zusätzliche Bits pro Knoten. Die Komplexität für folgende Operationen beträgt $O(\log_2 n)$:

1. Finden des Elements mit dem Schlüssel *k*
2. Einfügen eines Elements bezüglich des Schlüssels *k*
3. Löschen eines Elements aus dem Index

Eine genaue Kostenanalyse ist in [Knu98v3] und [OttW96] zu finden. Daneben bietet der AVL-Baum einen sequenziellen Zugriff in Sortierreihenfolge bezüglich des Schlüssels in linearer Zeit [Knu98v1]. Alternativ können auch andere balancierte Datenstrukturen, wie z.B. der 2-3 Baum [AhoHU74] oder dessen binäre Repräsentation [Bay71] eingesetzt werden.



(a) Z-Partitionierung



(b) Z-Intervalle in Φ

UB-Baum Region	Z-Intervall	Φ Z-Intervall	Event
ρ1	[0,7]	[32,43]	32_T
ρ2	[8,15]	[60,63]	54_T
ρ3	[16,20]		
ρ4	[21,27]		
ρ5	[28,35]		
ρ6	[36,43]		
ρ7	[44,59]		
ρ8	[60,63]		

(c) Region, UB-Z-Intervall, Φ–Z-Intervalle, Eventpunkte

Abbildung 5-33: Φ für einen 2-dimensionalen Basisraum

Um das aktuelle Z-Intervall [α,β], z.B. [16,20], das die Region ρ_3 repräsentiert (siehe Abbildung 5-33), aus Φ zu entfernen, muss im AVL-Baum mit der Bedingung

$$\alpha \leq \gamma \leq \beta \quad \text{mit } \gamma \in [\eta,\lambda] \wedge [\eta,\lambda] \in \Phi \qquad \text{Gl: 5-37}$$

gesucht werden. Hierzu muss der normale Such-Algorithmus auf Binär-Bäumen [OttW96] leicht modifiziert werden. Der Grund liegt darin, dass es sich hier nicht um eine genaue Punktanfrage handelt, sondern um ein Intervall. Dazu wird die Vergleichsfunktion wie folgt auf dem Z-Intervall definiert:

$$\begin{aligned} \alpha < [\eta,\lambda] \quad &\Rightarrow \quad \alpha < \eta \\ \alpha \in [\eta,\lambda] \quad &\Rightarrow \quad \eta \leq \alpha \leq \lambda \\ \alpha > [\eta,\lambda] \quad &\Rightarrow \quad \alpha > \lambda \end{aligned}$$

Somit kann in O(log *n*) Vergleichen bestimmt werden, ob die untere oder obere Grenze des Z-Intervalls [a,b] in ein Z-Intervall von Φ fällt. *n* bezeichnet die Anzahl der Z-Intervalle in Φ.

Um nun alle Z-Intervalle in Φ zu finden, die das Z-Intervall [α,β] schneiden, wird mit α in den AVL-Baum eingestiegen. Es wird soweit in dem Baum herabgestiegen, bis man ein PHI-Element findet, für das $\alpha \in [\eta,\lambda]$ gilt, oder ein Blatt erreicht wird, ohne dass die Bedingung erfüllt worden ist. Dann wird der Baum „In-Order" traversiert, bis ein Knoten gefunden wird, für den $\eta \leq \beta \leq \lambda$ oder $\beta < \eta$ gilt, oder kein weiterer Knoten existiert. Jeder Knoten, der traversiert worden ist, gehört zur Menge der geschnittenen Z-Intervalle und

wird aus dem AVL-Baum vollständig entfernt, falls er durch das Z-Intervall [α,β] vollständig abgedeckt ist. Das Z-Intervall wird aktualisiert, falls nur ein Teil durch das Z-Intervall [α,β] abdeckt ist. Es können maximal zwei Updates entstehen. Abbildung 5-34 zeigt den Z-Index für den in Abbildung 5-33 dargestellten 2-dimesionalen Basisraum Ω.

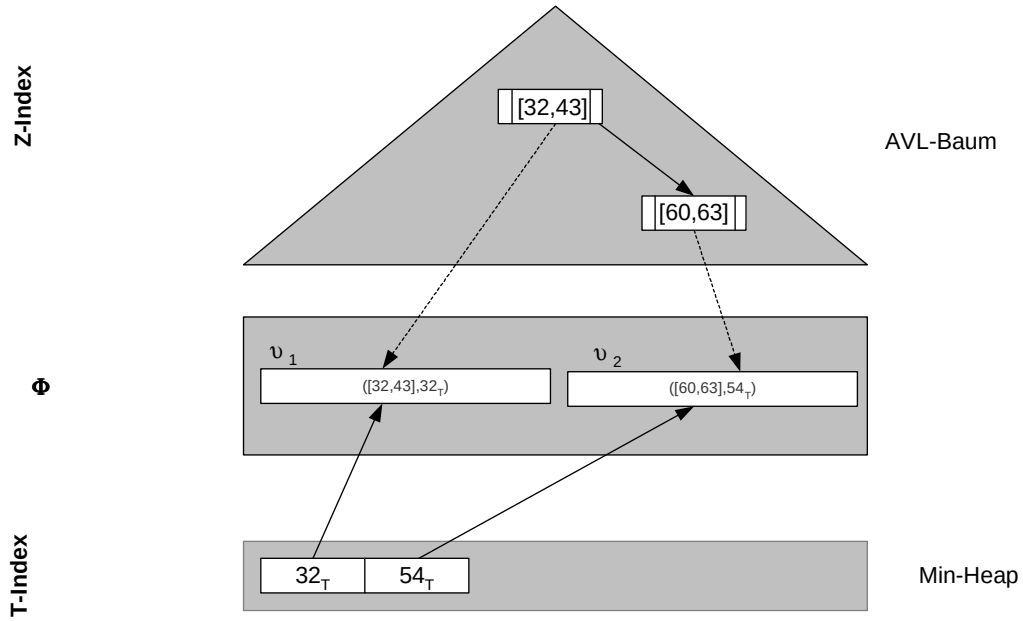


Abbildung 5-34: Φ-Verwaltung mit Indexen

Um den nächsten Eventpunkt zu finden, muss immer die kleinste[9] T-Adresse aus Φ zurückgegeben werden. Diese Operation entspricht der Vorrangswarteschlage (engl. priority Queues), die sehr effizient durch einen Heap realisiert werden kann [Wil64]. Deshalb wird die T-Adresse des jeweiligen Z-Intervalls in einem Heap gespeichert. Somit kann die minimale T-Adresse von Φ in O(1) zurückgegeben werden. Um die minimale T-Adresse aus dem Heap zu entfernen, benötigt man für das Einfügen $O(\log n)$ Vergleichs-Operationen. Abbildung 5-34 zeigt die vollständige Datenstruktur für die Verwaltung von Φ. Alle geforderten Operationen haben eine logarithmische CPU-Komplexität.

```
typedef struct PHI_S
{
    AD ql;            /* untere Grenze des Anfragebereichs    */
    AD qh;            /* obere Grenze des Anfragebereichs     */
    boolean min;      /* Flag ob absteigend oder aufsteigend  */
                      /*  der Tetris Algorithmus gelesen wird */
    int extAttr;      /* Sortierdimension                     */
    DUB *dub;         /* Pointer auf Metadaten des UB-Baums   */
    ACCPA ap;         /* treap */

}
PHI;
```

Abbildung 5-35: Φ

[9] Ohne Beschränkung der Allgemeinheit, wird hier nur der aufsteigend sortierte Fall betrachtet. Im Beispiel in Kapitel 5.8 wurde der absteigende Fall dargestellt, um den Namen Tetris zu veranschaulichen.

Abbildung 5-35 zeigt die C-Struktur der Prototypimplementierung für Φ. Für die Zugriffs-Struktur wurde nicht ein AVL-Baum und ein Heap, sondern ein Treap eingesetzt. Hierbei handelt es sich um eine Hybrid-Struktur, die einen Binärbaum und einen Heap miteinander vereint.

Die Datenstruktur *Treap* wurde von Aragon und Seidel [AraS89] entwickelt. Hierbei handelt es sich um einen binären Suchbaum für die Schlüsselkomponenten von *R* und einen Min-Heap [Wil64] für die Prioritäten der Elemente von *R*. Ein Treap ist somit eine Hybridstruktur, die die Eigenschaften von binären Suchbäumen (trees) und Prioritäts-warteschlangen (heap) miteinander vereint.

Das Problem des Binärbaums besteht jedoch darin, dass er zu einer Liste entarten kann, wenn die Daten sortiert eingefügt werden. Die Kosten für die Einfüge- und Such-Operation betragen in diesen Fall $O(n)$. Werden die Daten per Zufall in den Baum eingefügt, bleibt er jedoch balanciert und die Kosten sind dann $O(\ln n)$.

Um die Balancierung auch beim Einfügen von sortierten Folgen zu erhalten, wird zusätzlich eine neue Zufallsgröße eingeführt, die die Heap-Ordnung bestimmt. Diese Zufallsgröße ist unabhängig von der Ordnung der eingefügten Daten und erzeugt somit einen balancierten Binärbaum.

Da für die Verwaltung von PHI ein AVL-Baum für die Z-Intervalle und ein Heap für die T-Adresse eine Realisierungsmöglichkeit ist, liegt es nahe diese beiden Zugriffsstrukturen durch den Treap zu ersetzen. Hierbei wird die T-Ordnung, die den Bereich bestimmt, der als nächstes bearbeitet wird, als Priorität im Treap herangezogen.

Benutzt man die T-Ordnung für die Priorität im Treap, kann durch die Heap-Eigenschaft das Minimum $O(1)$ bestimmt werden, da es an der Wurzel liegt. Um die Heap-Eigenschaft wieder herzustellen, werden $O(\log n)$ Operationen benötigt, so dass $O(\log n)$ Kosten entstehen.

```
Status PHI_cut(PHI * phi, AD * rl, AD * rh)
{
   int num = 0;
   TreapNode *elPtr = NULL;
   long anzrot = 0;
   PHIEL source;
   PHIEL res1;
   PHIEL res2;
   DUB *dub;
   int extAttr;

   dub = PHI_getDubPtr(phi);
   extAttr = PHI_getExtAttr(phi);

  /**
     map the ADs to region to find it in phi
   */

   PHIEL_set(PHI_getMetaAccPaPtr(phi), (ADTELDATA *) & source, rl, rh);

  /**
    find the set of regions that an intersected by the
     region "source"
   */
    while (TR_intersect(PHI_getAccPaPtr(phi), &elPtr,
           (ADTELDATA *) & source) == OK)
    {
      PHIEL_copy(PHI_getMetaAccPaPtr(phi), (ADTELDATA *) & res1,
                       elPtr->data);
      TR_delete(PHI_getAccPaPtr(phi), (TreapNode *) elPtr, &anzrot);
      if (PHIEL_difference(PHI_getMetaAccPaPtr(phi),
                           (ADTELDATA *) & res1,
                           (ADTELDATA *) & res2,
                           (ADTELDATA *) & source,
                           &num) == OK)
      {
         if (num >= 1)
         {
            TR_insertData(PHI_getAccPaPtr(phi),
               (ADTELDATA *) & res1, &anzrot);
         }
         if (num == 2)
         {
            TR_insertData(PHI_getAccPaPtr(phi), (ADTELDATA *) & res2,
                          &anzrot);
         }
      }

   }
   return OK;
}
```

Abbildung 5-36: Differenzmenge-Operator

Die Z-Ordnung wird dementsprechend für die Suche im Binärbaum verwendet. Somit können durch einfaches Suchen im Binärbaum die entsprechenden Z-Intervalle bestimmt werden, die das Z-Intervall $[\alpha,\beta]$ schneiden. Abbildung 5-36 und Abbildung 5-37 zeigen die Prototypimplementierung.

```
Status TR_intersect(Treap * t, TreapNode ** res, ADTELDATA * d)
{
    TreapNode *pos = t->root;

    *res = NULL;

    if (t != NULL)
    {
        while (pos != NULL)
        {
            if (ADTELDATA_intersect(t->privMeta, d, pos->data) > 0)
            {
              pos = pos->Right;
            }
            else if (ADTELDATA_intersect(t->privMeta, d, pos->data) < 0)
            {
                pos = pos->Left;
            }
            else if (ADTELDATA_intersect(t->privMeta, d, pos->data) == 0)
            {
                *res = pos;
                break;
            }
       }

        if (*res == NULL)
        {
            return UB_setError(Treap_intersect__ERROR,
                  "No intersection found",ERROR);
        }
        else
        {
            return OK;
        }
    }
    return UB_setError(Treap_intersect__ERROR, "No TreapNodes", ERROR);
}
```

Abbildung 5-37: Suchen im Treap

Beim Einfügen eines Z-Intervalls werden somit seine T-Adresse und eine Z-Adresse berechnet und entsprechend in den Treap eingefügt.

Es stellt sich jedoch die Frage, ob die T-Ordnung und die Z-Ordnung einen balancierten Binärbaum erzeugen, da es zwischen beiden Ordnungen eine Korrelation gibt (siehe Kapitel 5.4), sie also nicht unabhängig voneinander sind.

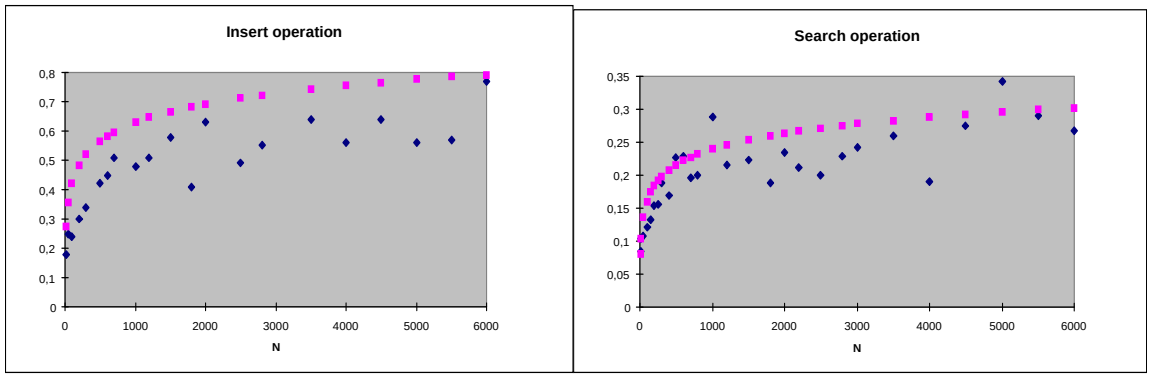


Abbildung 5-38

Abbildung 5-38 und Abbildung 5-39 zeigen das Verhältnis der Zeit zur Anzahl der Tupel für die drei Grundoperationen Suchen, Einfügen und Löschen. Die hellen Quadrate zeigen

den idealisierten logarithmischen Verlauf. Die dunklen Rauten sind die gemessenen Werte des Treaps für die unterschiedlichen Datenmengen. Für alle drei Grundoperationen kann ein logarithmischer Verlauf festgestellt werden. Ein wesentlicher Vorteil gegenüber einer 1-dimensionalen balancierten Datenstruktur, wie z. B. ein 2-3-Baum oder der AVL-Baum, besteht darin, dass für die Verwaltung von Φ nur ein Index benötigt wird und so der Speicherbedarf für den Index auf 50% reduziert wird.

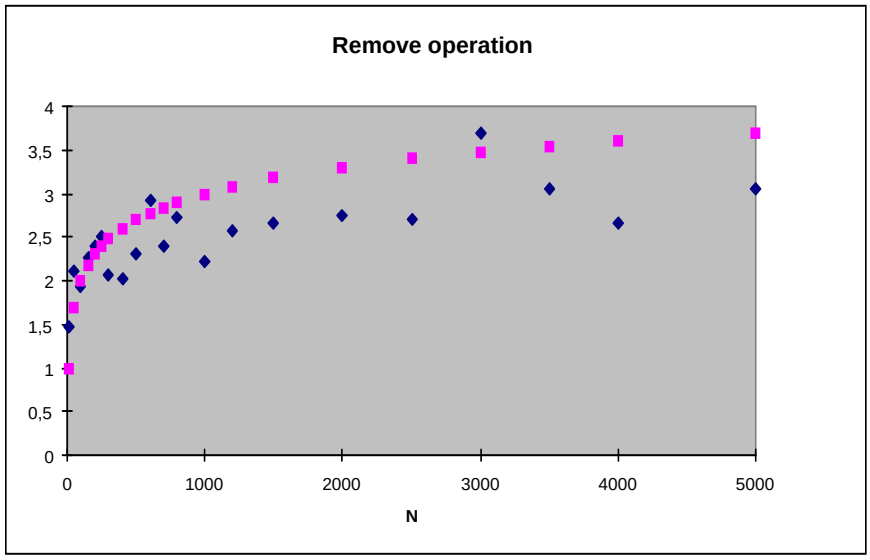


Abbildung 5-39

5.10 Cacheverwaltung

An die Cacheverwaltung des Tetris-Algorithmus werden folgende Anforderungen gestellt:

- Füge die Tupel der Seite, die den Anfragebereich Q schneiden, in den Cache ein.
- Gib alle Tupel sortiert nach der Sortierdimension aus, die kleiner als die Sweep-Hyperebene sind.

Die wesentliche Operation ist somit die Sortierung. Hier gibt es im Wesentlichen zwei Alternativen: Quicksort [Knu98v3] und Heapsort [Knu98v3].

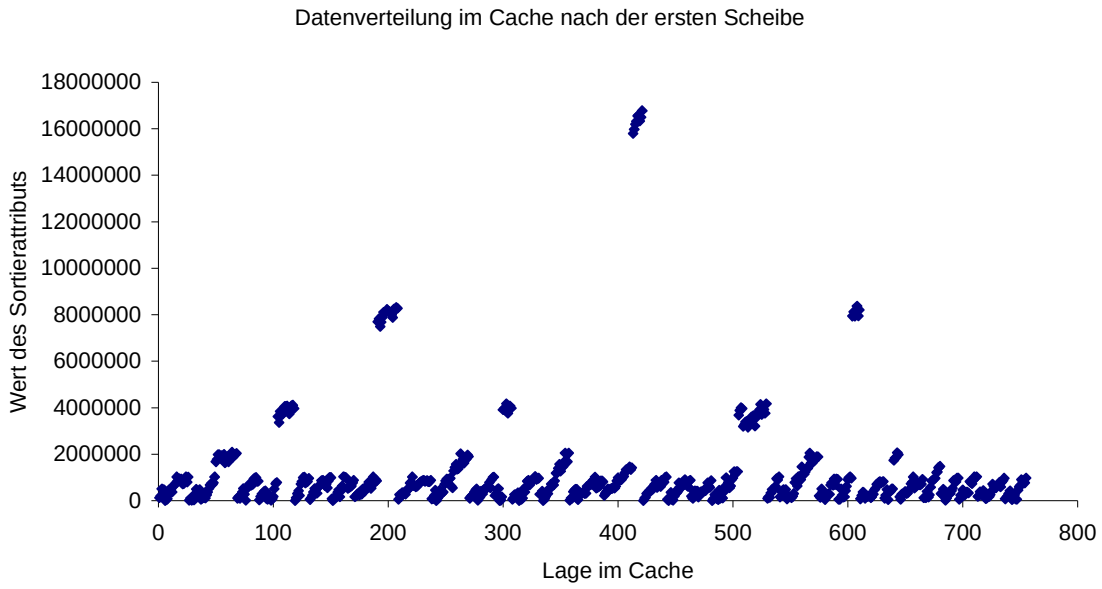


Abbildung 5-40: Datenverteilung des Caches nach der ersten Scheibe für den 2-dimensionalen Fall

Abbildung 5-40 zeigt die Datenverteilung des Caches nach dem Lesen der ersten Scheibe. Hierbei wurden die Tupel in eine Liste eingetragen. Der Graph zeigt eindeutig, dass die meisten Daten relativ nahe bei der Sweep-Hyperebene $\Psi_{1,0}$ liegen. Eine genaue Analyse der Datenverteilung im Cache sowie ein Vergleich von Quicksort und Heapsort für ihre Anwendung in der Cacheverwaltung ist im Anhang (Kapitel 11.1 und 11.2) zu finden. Die Analyse basiert auf [KleZ00], die vom Autor betreut worden ist.

Das Ergebnis dieser Analyse ist, dass der Z-Abstand wie eine Parabel in Sortierrichtung wächst. Da die Z-Kurve nicht stetig ist, kann es jedoch auch zu Sprüngen kommen. Dies wiederum führt dazu, dass zwei Punkte in Sortierrichtung den maximalen Abstand besitzen, jedoch auf der Z-Kurve Nachbarn sind. Diese Regionen sind somit Problemregionen, da die Daten sehr lange im Speicher gehalten werden müssen. Die Anzahl der Sprünge, die niedrige Stufen überspringen, nimmt exponentiell ab. Somit ist die Z-Kurve relativ lokal und es existieren nur wenige Regionen, deren Tupel lange im Arbeitsspeicher gehalten werden müssen. Ein weiteres Ergebnis ist, dass das Sortierattribut einen signifikanten Einfluss auf die Datenverteilung des Caches hat. Allgemein kann man sagen, dass weniger große als kleine Sprünge existieren. Das Modell wurde durch Messungen bestätigt.

Die internen Sortierverfahren Quicksort und Heapsort zeigen in Messungen ähnliche CPU-Kosten, obwohl der Quicksort-Algorithmus im Allgemeinen nur 56% der Vergleichsoperationen des Heap-Sort-Algorithmus benötigt. Bei der Cacheverwaltung des Tetris-Algorithmus ergibt sich jedoch ein völlig anderes Bild, das durch die Struktur der Z-Kurve, nach der der UB-Baum die Daten auf den Sekundärspeicher clustert, verursacht wird. Im Cache sind bereits n_{old} Tupel, die bereits durch die Verarbeitung vorangegangener Scheiben in den Cache geladen worden sind. Es werden pro Scheibe n_{new} Tupel hinzugefügt. Somit sortiert der Quicksort-Algorithmus n_{old} Tupel erneut. Ist das Verhältnis $n_{new}/n_{old} < 0{,}64$ benötigt der Heapsort-Algorithmus weniger Vergleiche als der Quicksort-Algorithmus und stellt somit die bessere Alternative für die Sortierung des Caches dar. Da der Heap-Sort-Algorithmus nicht ganz so effizient wie der Quicksort-Algorithmus ist,

kostet eine Vergleich-Operation in beiden Algorithmen unterschiedlich viel Zeit. Bei der hier verwendeten Prototypimplementierung benötigt der Heap-Sort-Algorithmus 2,4 mal mehr Zeit als der Quicksort-Algorithmus um eine Vergleichsoperation auszuführen. Des Weiteren wurde festgestellt, dass die Löschoperation des Heaps der Kostentreiber ist. Da der Heap-Sort-Algorithmus O(n log n) garantiert, stellt er die bevorzugte Methode für den Tetris-Algorithmus dar.

5.11 Leistungsmessung

In diesem Abschnitt werden einige Leistungsmessungen präsentiert, die den Tetris-Algorithmus dem SRQ-Algorithmus gegenüberstellen.

Die Messungen wurden auf einem AMD Thunderbird 900MHZ durchgeführt. Der Arbeitsspeicher M beträgt 768MB. Als Sekundärspeicher wurde eine Seagate ST38416 mit 18 GB eingesetzt. Die durchschnittliche Lesezeit ist mit 5,5 *ms* angegeben.

Für die Messung wurde eine 3-dimensionale Datenbank verwendet. Die technischen Daten sind Tabelle 5-3 zu entnehmen. Die Datenbank wurde durch unsortiertes Einfügen in den UB-Baum erzeugt, so dass der UB-Baum nicht seitengeclustert ist.

Name	Dimensionen	Wertebereich	Verteilung	Anzahl der Tupel	Anzahl der Seiten	Datenbankgröße [GB]
ub3d1G	3	[0, 6777215]	gleichverteilt	2400000	419958	0,83

Tabelle 5-3

```
SELECT x1,x2,x3 FROM table
WHERE x1 BETWEEN a AND b AND
      x2 BETWEEN c AND d AND
      x3 BETWEEN e AND f
ORDER BY x1
```

Die Selektivität der Sortier-Dimension x_1 wurde auf 25 %, 50%, 75% und 100% festgelegt, die Dimension x_2 und x_3 jeweils auf 50%. Für die jeweilige Selektivität wurde eine Messung mit dem SRQ-Algorithmus und mit dem Tetris-Algorithmus durchgeführt.

Die Ergebnismenge P_E umfasst bei der 25%-Messung 26.330 Seiten. Bei einer Seitengröße von 2 KB ergibt das eine Ergebnisgröße von ca. 51,4 MB. Für die 50%-Messung ergibt sich eine Ergebnismenge P_E von ca. 102,8 MB. Für die 75% und 100%-Messung erhält man eine Ergebnismenge P_E von ca. 154,1 MB und 205,5 MB (siehe Tabelle 5-4.). Dies bestätigt somit das Kostenmodell aus 4.4.4.1 Gleichung Gl: 4-60.

Operator	Selektivität x_1 [%]	E/A [s]	CPU [s]	Scheiben	gelesene Seiten	mehrmals gelesene Seiten [%]	Antwort zeit [s]	Erstes Teilergebnis[s]
SRQ	25	199,6	0,8	16	29.876	13	200,4	16,5
Tetris	25	212,1	13,4	55	26.330	0	225,5	16,7
SRQ	50	399,1	1,9	32	59.732	13	401,0	16,5
Tetris	50	424,1	26,6	117	52.639	0	450,7	16,7
SRQ	75	589,9	2,5	38	89.628	14	601,4	16,5
Tetris	75	636,1	39,9	178	78.939	0	676,0	16,7
SRQ	100	798,5	3,3	64	119.494	14	801,8	16,5
Tetris	100	848,1	53,2	226	105.253	0	901,3	16,7

Tabelle 5-4

Die E/A-Kosten $c_{E/A\text{-}SRQ}$ und $c_{E/A\text{-}tet}$ (siehe Gl: 4-72) für den SRQ-Algorithmus und den Tetris-Algorithmus (siehe Kostenmodell 4.4.4.1) ergeben sich aus den Lesekosten des UB-Baums und aus dem Schreiben der Ergebnismenge P_E.

Eine Scheibe beim SRQ-Algorithmus besteht aus ca. 1645 Seiten. Bei einer Seitengröße von 2 KB ergibt sich somit eine Cachebedarf von 3,2 MB. Somit kann die Scheibe vollständig im Arbeitsspeicher gehalten und durch einen Quicksort-Algorithmus in die Zielordnung überführt werden. Da der Tetris-Algorithmus Tupel im Cache hält, bis sie im statischen Bereich liegen, ist der Cachebedarf größer als im SRQ-Algorithmus. Für diese Messung beträgt der maximale Cachebedarf 2439 Seiten, das entspricht 4,7 MB. Der Tetris-Algorithmus hat somit 46% mehr Cachebedarf. Im Durchschnitt benötigt der Tetris-Algorithmus jedoch nur 1527 Seiten das entspricht ca. 2,9 MB. Dies bestätigt das Kostenmodell aus 4.4.4.3. Nachdem ist der Cachebedarf $cache_{sortierte\_Lesen}(P,d) = \sqrt[d]{P^{d-1}}$.

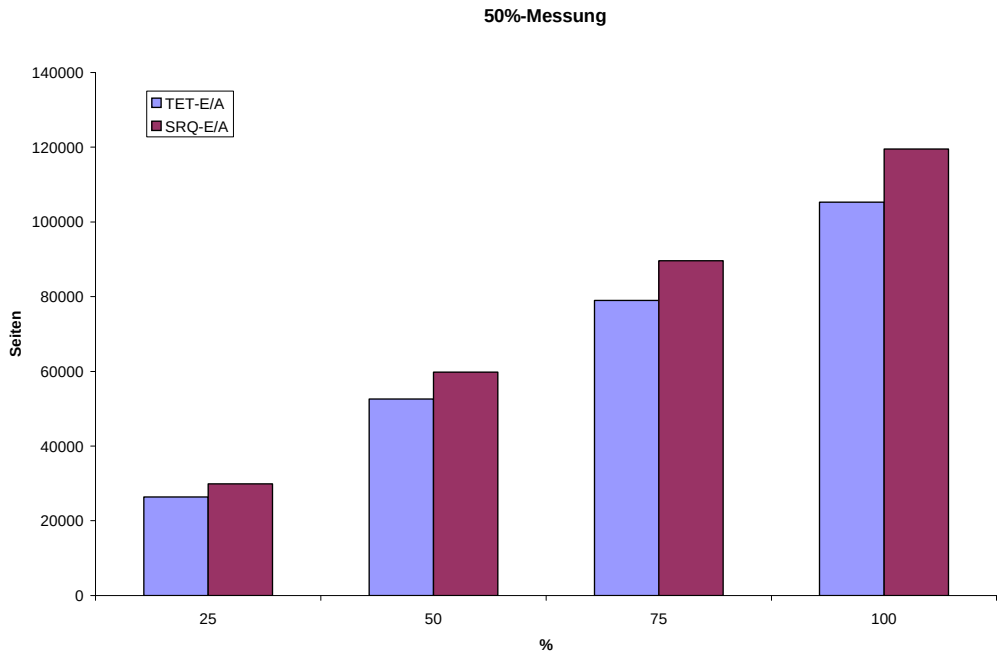


Abbildung 5-41: Anzahl der gelesenen Seiten

Da der Tetris-Algorithmus Tupel, die bereits im Arbeitsspeicher gelesen worden sind, jedoch noch im dynamischen Bereich liegen, im Cache hält, liest der Tetris-Algorithmus jede Seite nur einmal. Da jedoch die Region wegen der Fransen [FriM97] und Sprünge nicht unbedingt in eine statische Scheibe des SRQ-Algorithmus fallen, werden manche Regionen mehrmals gelesen. Für die 50%-Messung werden 13-14% mehr Seiten als vom Tetris-Algorithmus gelesen. (siehe Abbildung 5-41, Tabelle 5-4).

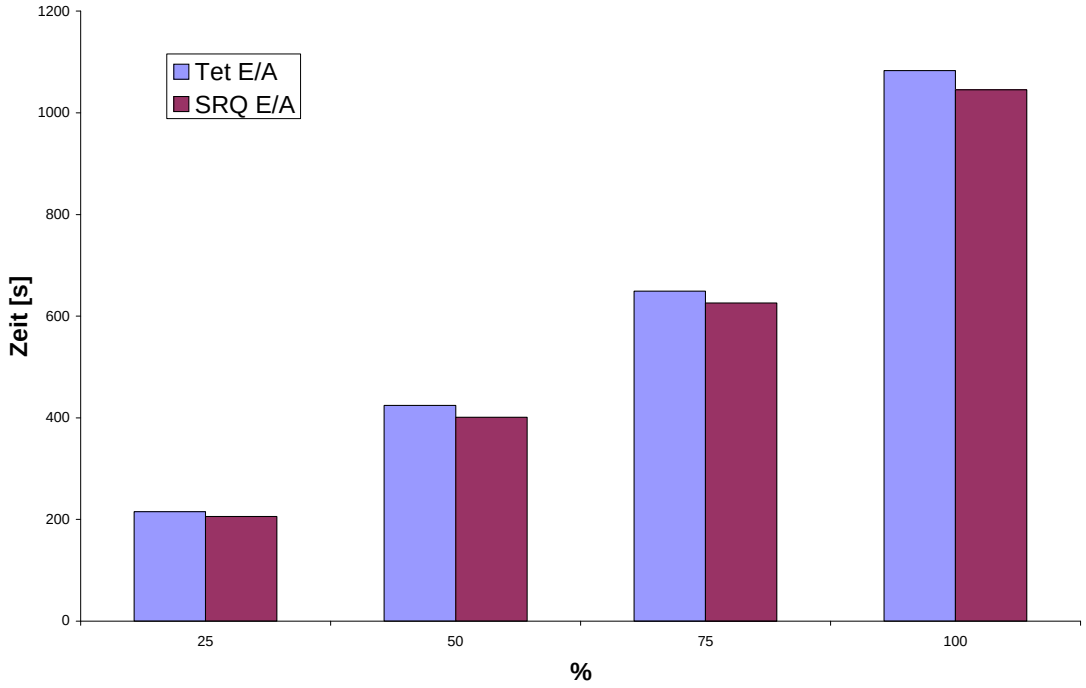


Abbildung 5-42: E/A-Kosten

Obwohl der SRQ-Algorithmus mehr Seiten aus dem Sekundärspeicher als der Tetris-Algorithmus lesen muss, sind die E/A-Kosten geringer (siehe Abbildung 5-42). Die E/A-Kosten beim Tetris-Algorithmus liegen um 6% über den E/A-Kosten des SRQ-Algorithmus (siehe Tabelle 5-4). Der Grund liegt in den Kosten pro Seite. Der SRQ-Algorithmus benötigt pro Seite 6,6 *ms*, der Tetris-Algorithmus hingegen 8,0 *ms*. Somit sind die E/A-Kosten pro Seite 20% höher als beim SRQ-Algorithmus (siehe Abbildung 5-42). Da der Tetris-Algorithmus streng nach der Tetris-Ordnung die Seiten aus dem Sekundärspeicher liest und der SRQ-Algorithmus im Prinzip eine Menge von Bereichsanfragen, die nach der Z-Ordnung verarbeitet werden, abarbeitet, ist das E/A-Verhalten des SRQ-Algorithmus besser als das des Tetris-Algorithmus, da der Prefetchfaktor C besser ausgenutzt werden kann.

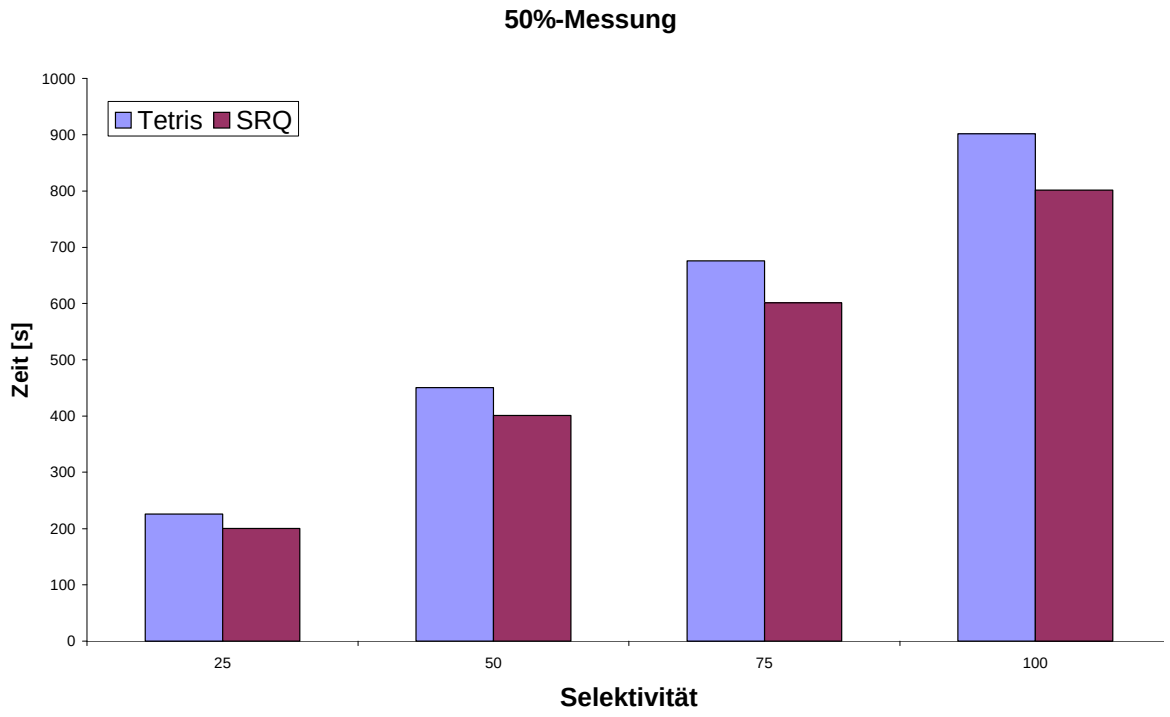


Abbildung 5-43: Antwortzeit

Abbildung 5-43 zeigt die Antwortzeit für den Tetris- und SRQ-Algorithmus. Der SRQ-Algorithmus ist dem Tetris-Algorithmus überlegen. Der Tetris-Algorithmus benötigt ca. 12% mehr Zeit als der SRQ-Algorithmus, d.h. dass die CPU-Kosten für den Tetris-Algorithmus höher sind als die des SRQ-Algorithmus. Dies entspricht den Erwartungen, da der Tetris-Algorithmus zum einen die komplexe Raumverwaltung Φ^{-1} verwendet, um die nächste T-Adresse zu berechnen und zum anderen noch nicht verarbeitete Datensätze im Cache hält. Für den Tetris-Algorithmus wird ein Heap eingesetzt, da Tupel im Cache verbleiben (siehe Anhang Kapitel 11.2). Der SRQ-Algorithmus verwendet den Quick-Sort Algorithmus, der effizienter als *Heapsort* ist [Knu98v3], wenn keine Tupel im Cache vorhanden sind. Der SRQ-Algorithmus fügt nur Tupel in den Cache ein, die zur aktuellen Scheibe gehören, so dass jedes Tupel genau einmal sortiert wird.

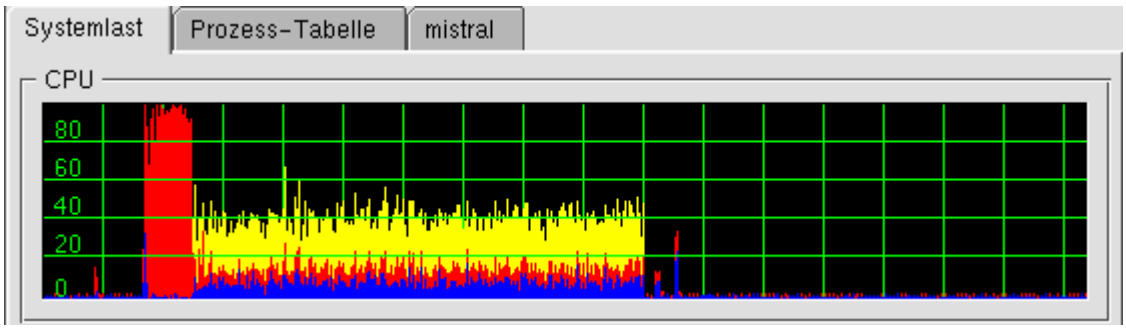


Abbildung 5-44: CPU Tetris-Algorithmus bei einer 3 dimensionalen DB

Beide Algorithmen sind bei dieser Konfiguration nicht *CPU-Bound* (siehe Abbildung 5-44 und Abbildung 5-45). Der Tetris-Algorithmus zeigt eine kontinuierliche CPU-Belastung wo hingegen der SRQ-Algorithmus große Schwankungen in der CPU-Belastung aufweist.

Da der SRQ-Algorithmus eindeutig zwischen Sortieren und Lesen trennt, erkennt man schön, zu welchem Zeitpunkt die Daten vom Sekundärspeicher gelesen werden und wann nicht. Der Tetris-Algorithmus benutzt zum Sortieren den Heap, so dass bei der Lesephase die Daten in den Heap eingefügt werden. Der Sortiervorgang wird somit bereits in der Lesephase begonnen.

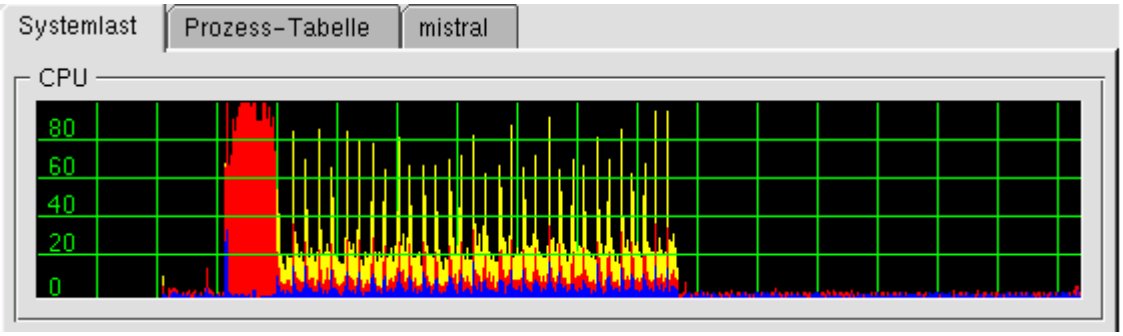


Abbildung 5-45: CPU SRQ-Algorithmus bei einer 3-dimensionalen DB

Um den Algorithmus auch unter Realbedingungen zu testen wurde das Data-Warehouse für die GFK aufgebaut. Hierbei werden die Daten auf 2-Monatsperioden voraggregiert abgespeichert. Die Daten werden in einem 3-dimensionalen Cube mit den Dimensionen *Produkt*, *Segment* und *Periode* verwaltet.

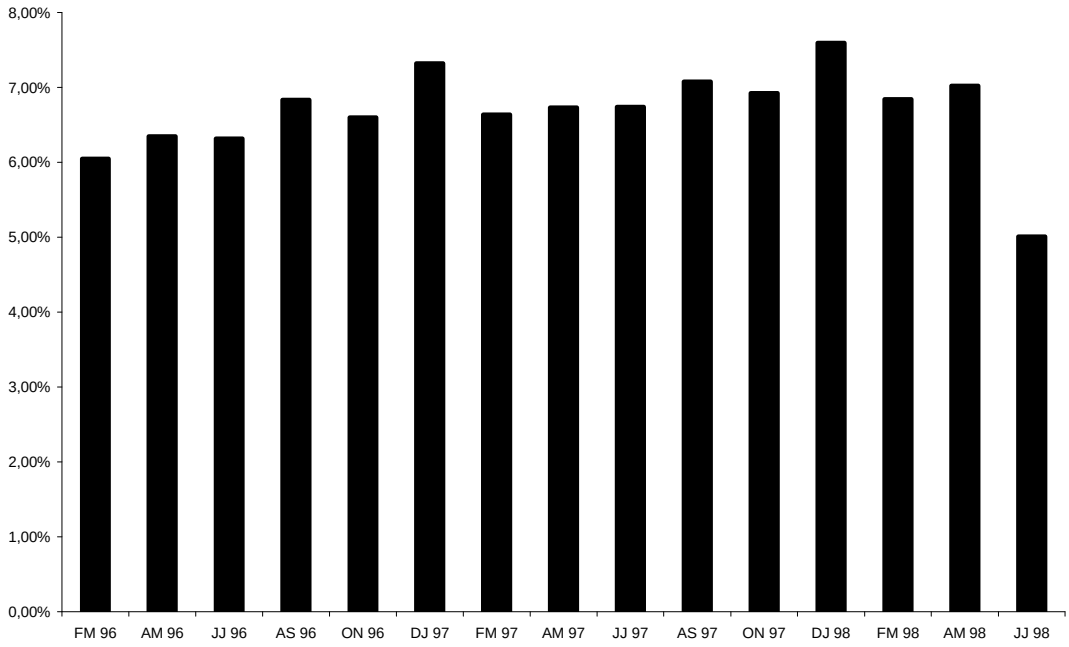
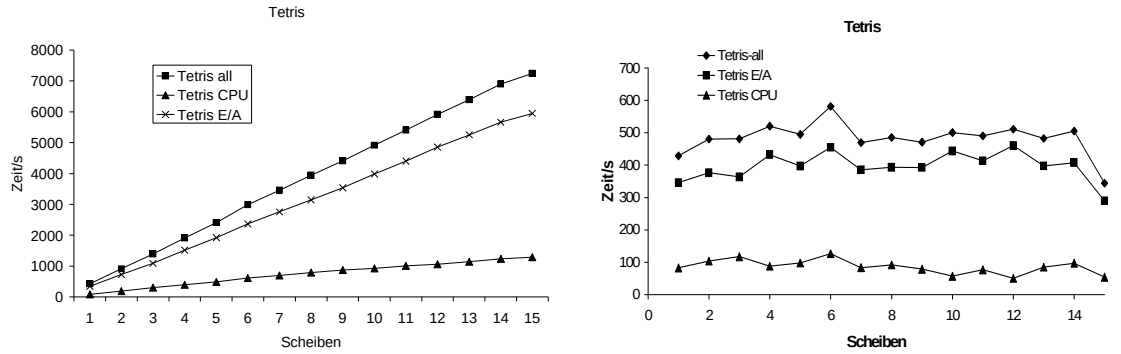


Abbildung 5-46: Datenverteilung Data-Warehouse GfK

Die Faktentabelle des Data-Warehouse der GfK besteht aus 43 Millionen Tupeln. Die Dimension *Periode* besteht auf der Konsolidierungsstufe 0 (siehe Kapitel 1.2) aus 15 Zwei-Monatsperioden. Die Datenverteilung bezüglich der Periodendimension ist in Abbildung 7-27 aufgetragen. Die Größe eines Tupels beträgt 56 Byte. Für die Messung wurde nur die Faktentabelle berücksichtigt, da es sich hier um die größte Tabelle handelt und auf diese Weise die Leistungssteigerung am eindrucksvollsten dargestellt werden kann. Die Seitengröße beträgt 2KB. Wegen der zusätzlichen internen Seitenverwaltung des Datenbanksystems können jedoch nur 31 statt 36 Tupel auf einer Seite gespeichert werden. Die Faktentabelle ist durch den TempTris-Algorithmus (siehe Kapitel 6.6) aufgebaut worden. Der TempTris-Algorithmus erzeugt einen UB-Baum, der seitengeclustert ist. Es handelt sich hierbei um einen Massenlade-Algorithmus.

Es wurde eine 100%-Messung durchgeführt. Hierzu wurde die Faktentabelle sortiert nach Dimension *Period* gelesen. Der SRQ-Algorithmus liest die Faktentabelle in 15 Scheiben. Der Tetris-Algorithmus liest die Faktentabelle in 17 Scheiben. Die ersten drei Scheiben des Tetris-Algorithmus wurden zur Scheibe 1 zusammengefasst, da erst nach der 3. Scheibe Datensätze zurückgegeben werden.



(a) (b)

Abbildung 5-47: Kosten des Tetris-Algorithmus

Die Ergebnismenge P_E umfasst bei einer Selektivität von 100% 1863822 Seiten. Bei einer Seitengröße von 2 KB ergibt sich somit eine Ergebnisgröße von 3,6 GB. Abbildung 5-47 zeigt die Kosten für den Tetris-Algorithmus. Abbildung 5-47.(a) zeigt die Kosten kumuliert in Abhängigkeit zu den verarbeiteten Scheiben. Abbildung 5-47.(b) zeigt die Kosten der jeweiligen Scheibe. Da die Daten bezüglich der Dimension *Period* relativ gleichmäßig verteilt sind, zeigt der Tetris-Algorithmus das erwartete lineare Verhalten. Das Verhältnis zwischen E/A-Kosten und CPU-Kosten beträgt 6/1. Der große CPU-Anteil liegt darin begründet, dass der gesamte Anfragebereich Q nur in 15 Scheiben verarbeitet wird und somit eine Scheibe im Durchschnitt 124254 Seiten enthält, die auf einmal sortiert werden. Somit benötigt man im optimalen Fall einen Cache von 242 MB. Dies entspricht dem Kostenmodell aus 4.4.4.3. Die Anzahl der Tupel, die der Tetris-Algorithmus im Cache zwischenspeichert, beträgt bei dieser Messung nur 546 Datensätze. Das entspricht 30 KB und kann somit vernachlässigt werden.

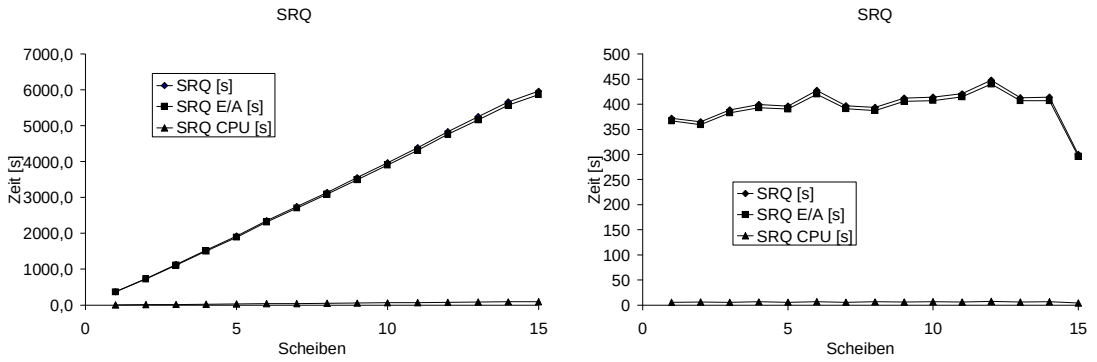


Abbildung 5-48: Kosten des SRQ-Algorithmus

Abbildung 5-48 zeigt die Kosten für den SRQ-Algorithmus. Abbildung 5-48(a) zeigt die Kosten kumuliert in Abhängigkeit zu den verarbeiteten Scheiben. Abbildung 5-48.(b) zeigt die Kosten der jeweiligen Scheibe. Der SRQ-Algorithmus zeigt das erwartete lineare Verhalten. Das Verhältnis zwischen E/A-Kosten und CPU-Kosten beträgt jedoch für den SRQ-Algorithmus 63/1. Der geringe CPU-Anteil gegenüber dem Tetris-Algorithmus liegt darin begründet, dass durch die Datenverteilung kaum Daten mehrfach geladen werden und somit die Anzahl der zu sortierenden Datensätze mit dem Idealfall beinnahe übereinstimmt und zum anderen der Quicksort-Algorithmus für diese Messung nur die Komplexität von O (n log n) zeigt. Eine Scheibe beim SRQ-Algorithmus enthält im Durchschnitt 124.310 Seiten, das entspricht 248 MB Cache. Der SRQ-Algorithmus benötigt somit 2% mehr Cache als der Tetris-Algorithmus.

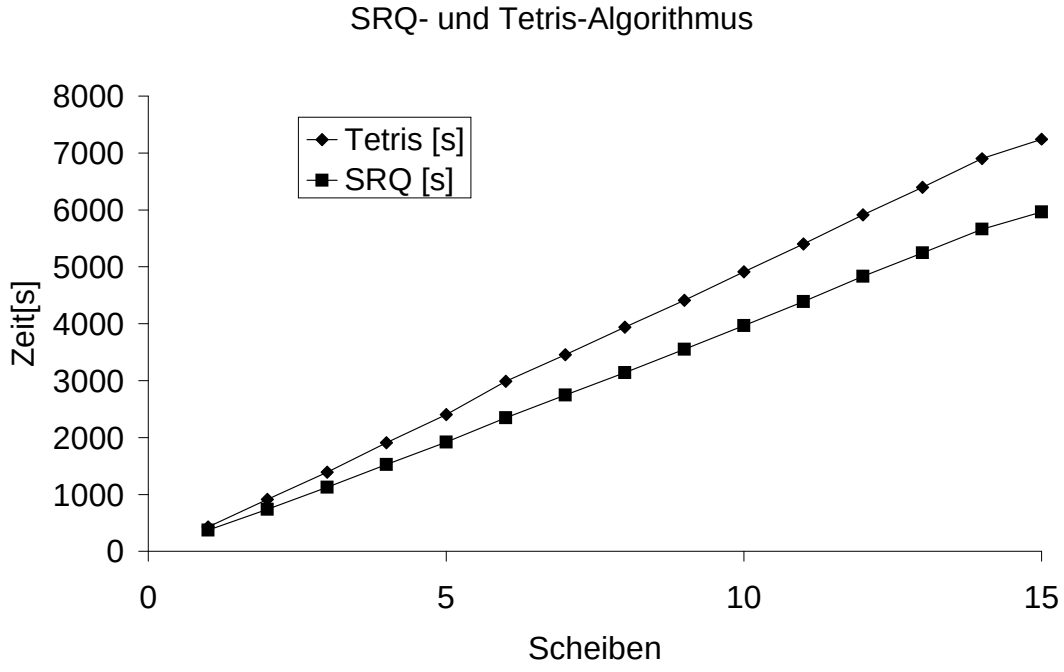


Abbildung 5-49

Abbildung 5-49 zeigt die Kosten des Tetris- und SRQ-Algorithmus für die 100%-Messung. Die Kosten sind kumuliert in Abhängigkeit zu den verarbeiteten Scheiben dargestellt.

Obwohl der Tetris-Algorithmus 830 Seiten weniger als der SRQ-Algorithmus liest, liegen die Gesamtkosten des Tetris-Algorithmus um 21 % über den Gesamtkosten des SRQ-Algorithmus.

Scheiben	Tetris [s]	SRQ [s]	Tetris E/A [s]	SRQ E/A [s]	Tetris CPU [s]	SRQ CPU [s]	Tetris Zeit/Seiten [ms]	SRQ Zeit/Seiten [ms]	Tetris Seiten	SRQ Seiten	Tetris Seiten	SRQ Seiten
1	430,3	372,2	347,2	366,7	83,1	5,5	3,8	3,3	112.664	112.832	112.664	112.832
2	910,4	737,3	723,4	725,8	187	11,6	4,1	3,0	230.913	230.947	118.249	118.115
3	1.391,2	1.126,1	1.086,6	1.108,9	304,6	17,3	4,1	3,3	348.545	348.821	117.632	117.874
4	1.911,2	1.525,8	1.518,7	1.501,9	392,5	23,9	4,1	3,1	475.890	476.014	127.345	127.193
5	2406	1.921,9	1.915,5	1.892,2	490,4	29,8	4,0	3,2	598.771	599.134	122.881	123.120
6	2987	2.349,2	2.370,1	2.312,3	616,9	36,9	4,3	3,1	735.251	735.531	136.480	136.397
7	3.456,2	2.746,2	2.755,6	2.703,4	700,6	42,8	3,8	3,2	858.854	859.294	123.603	123.763
8	3.941,3	3.140,1	3.148,5	3.090,7	792,8	49,4	3,9	3,1	984.244	984.636	125.390	125.342
9	4.412,2	3.552,3	3.540,7	3.496,7	871,5	55,5	3,8	3,2	1109.793	1.110.347	125.549	125.711
10	4.912,6	3.966,6	3.984,0	3.904,0	928,5	62,6	3,8	3,1	1241.664	1.242.151	131.871	131.804
11	5.402,7	4.387,4	4.397,3	4.318,6	1.005,4	68,8	3,8	3,2	1370.669	1.371.315	129.005	129.164
12	5.913,6	4.835,2	4.857,6	4.758,8	1.056,0	76,4	3,6	3,1	1512.189	1.512.849	141.520	141.534
13	6.395,9	5.248,1	5.254,7	5.165,6	1.141,2	82,6	3,8	3,2	1639.639	1.640.367	127.450	127.518
14	6.900,6	5.662,2	5.662,1	5.572,7	1.238,5	89,5	3,9	3,1	1770.510	1.771.279	130.871	130.912
15	7.244,2	5.962,5	5.951,7	5.868,7	1.292,5	93,8	3,7	3,2	1863.822	1.864.652	93.312	93.373

Tabelle 5-5

Dies entspricht den Erwartungen, da die gleichmäßige Datenverteilung und die geringe Anzahl der Scheiben das E/A-Verhalten des SRQ-Algorithmus begünstigt. Die Kosten pro Seite liegen beim SRQ-Algorithmus bei 3,2 ms und beim Tetris-Algorithmus bei 3,9 ms. Somit liegen die E/A-Kosten des Tetris-Algorithmus pro Seite 23% über denen des SRQ-Algorithmus. Tabelle 5-5 zeigt die Messergebnisse der 100%-Messung nochmals zusammengefasst.

5.12 Zusammenfassung

Der Tetris-Algorithmus liest die Regionen aus einem UB-Baum sortiert nach dem Sortierattribut, um eine Bereichsanfrage mit ORDER-Klausel effizient abzuarbeiten. Hierbei wird eine Art Sweepline-Algorithmus verwendet. Die Grundidee besteht darin, die Sweepline nur soweit in Sortierrichtung zu verschieben, bis wieder eine vollständige Scheibe im Arbeitsspeicher enthalten ist. Durch die Einführung einer Raum füllenden Kurve, der Tetris-Kurve, die auf der Tetris-Ordnung basiert, wird das *n*-dimensionale Schnittproblem auf ein 1-dimensionales Schnittproblem reduziert. Der Raum wird entsprechend der Tetris-Ordnung in den Arbeitspeicher gelesen. Somit wird jede Region nur einmal gelesen.

Nach der formalen Einführung der Tetris-Ordnung wurden die Eigenschaften der T-Kurve untersucht. Es konnte bewiesen werden, dass die T-Kurve eine raumfüllende Funktion auf einem diskreten Basisraum Ω ist. Die wesentliche Erkenntnis dieser Untersuchung besteht darin, dass die Sweep-Hyperebene, die den statischen Bereich vom dynamischen Bereich trennt, durch ein zusammenhängendes Intervall dargestellt werden kann und dass jedem Z-Intervall genau eine minimale und maximale T-Adresse zugeordnet werden kann. Somit kann die Hyperebene durch einen Punkt, der als Separator zwischen dynamischen und statischen Bereich fungiert, ersetzt werden. Anhand dieses Separators kann in Kombination mit der Minimalen oder Maximalen T-Adresse entschieden werden, ob ein Z-Intervall den

dynamischen Bereich schneidet. Es wurde ein Algorithmus vorgestellt, der die minimale bzw. maximale T-Adresse in O(*n*) berechnet, wobei *n* die Anzahl der Bits bezeichnet, die zur Repräsentation der T-Adresse benötigt werden.

Der Tetris-Algorithmus wurde dann präsentiert. Der Tetris-Algorithmus beruht im Gegensatz zum SRQ-Algorithmus weder auf statischen Scheiben, die jeweils mit dem Bereichsanfrage-Algorithmus abgearbeitet werden, noch auf Statistiken über die Datenbankgröße und deren Datenverteilung. Der Tetris-Algorithmus lädt jeweils die minimale Scheibe aus dem UB-Baum, indem der Anfragebereich nach der T- Ordnung abgearbeitet wird. Das Kostenmodell aus Kapitel 4 für den SRQ-Algorithmus basiert auf gleichverteilten Daten und kann deshalb auf den Tetris-Algorithmus angewandt werden. Die Korrektheit des Kostenmodells wurde durch Messungen belegt, die sowohl auf künstlichen als auch auf realen Daten (GfK) durchgeführt worden sind. Bei den Vergleichsmessungen zeigten beide Algorithmen ähnliche Gesamtkosten. Der Tetris-Algorithmus liest immer die minimale Anzahl der Seiten aus dem Sekundärspeicher und entspricht somit dem RQ-Algorithmus. Der SRQ-Algorithmus, der die Scheiben jeweils durch einen RQ-Algorithmus abarbeitet, liest durch die Struktur der Region mehrmals und ist somit dem Tetris-Algorithmus unterlegen. Bei den Messungen für gleichverteilte Daten wurden durch den SRQ-Algorithmus 14% mehr Seiten gelesen als durch den Tetris-Algorithmus. Der SRQ-Algorithmus hat jedoch ein besseres E/A-Verhalten, da der SRQ-Algorithmus die Daten nach der Z-Kurve aus dem Sekundärspeicher liest und somit den Prefetchfaktor *C* besser ausnutzt. Betrachtet man die Zeit, die benötigt wird, um die Seiten zu laden, war der SRQ-Algorithmus dem Tetris-Algorithmus um 6% überlegen. Die Verwaltung des Cache beim Tetris-Algorithmus erzeugt zusätzliche CPU-Kosten. Bei dieser Prototyp-Implementierung erhielt man im Mittel für das Verhältnis SRQ-Algorithmus zu Tetris-Algorithmus den Wert 1:15 an CPU-Kosten. Beide Algorithmen sind nicht CPU-Bound. Somit können die Ergebnisse aus Kapitel 4 übernommen werden.

Daneben wurde die Datenverteilung und Verwaltung des Caches für den Tetris-Algorithmus untersucht. Zum einen konnte gezeigt werden, dass sie Sortierrichtung einen signifikanten Einfluss auf die Datenverteilung des Caches hat, zum anderen nimmt die Anzahl an Sprüngen exponentiell zur Entfernung ab.

Instead of this absurd division into sexes they ought to class people as static and dynamic.

Evelyn Waugh (1903–1966), British novelist.
Decline and Fall (1928)

6 Gruppierung und Duplikateliminierung

In diesem Kapitel, dem zweiten Themenbereich dieser Arbeit (siehe Abschnitt), beschäftigen wir uns mit der Berechnung von Gruppierungen. Das Problem kann wie folgt beschrieben werden.

Gegeben ist eine Folge von Tupeln:

$$\langle a_{(0)}, a_{(1)},a_{(2)},a_{(3)},\ldots,a_{(n-1)} \rangle \text{ mit } \prec a_{(0)} \prec a_{(1)} \prec a_{(2)} \prec a_{(3)} \prec \ldots \prec a_{(n-1)} \qquad \text{Gl: 6-1}$$

Ohne Beschränkung der Allgemeinheit bestehen die Tupel nur aus Schlüsselattributen und der Menge der Gruppierungsattribute aus einem Attribut. Der Gruppierungsalgorithmus soll nun die Folge in eine Menge von Partitionen $\{B_0,B_1,...,B_{k-1}\}$ bezüglich des Gruppierungsattributs *j* zerlegen:

$$\{B_0,B_1,...,B_{k-1}\} \text{ mit } \forall B_i: a,b \in B_i \Leftrightarrow a_j = b_j \qquad \text{Gl: 6-2}$$
$$\text{und } \bigcup_{i=0}^{k-1} B_i = \{a_{(0)},a_{(1)},a_{(2)},...,a_{(n-1)}\}$$
$$\text{und } (B_i \in a) \wedge (B_i \in b) \wedge B_i \neq B_k \Rightarrow a_j \neq b_j$$

Ziel ist es nun, die Daten, die das Selektionsprädikat erfüllen, vom Sekundärspeicher zu laden und nach einem Attribut zu gruppieren. Hierbei soll das Attribut Element des Schlüssels sein. Im Allgemeinen werden dann auf den einzelnen Mengen (Gruppen) Mengenfunktionen (Aggregatfunktionen, Spaltenfunktionen) angewendet, wie z.B. SUM. Diesen Sachverhalt kann man in SQL wie folgt dartellen.

```
SELECT Ki, SUM(sales), … FROM Fakten_Tabelle_UB-Baum
WHERE  K1 BETWEEN r1 AND r2,
       K2 BETWEEN r3 AND r4,
       K3 BETWEEN r5 AND r6,
       …
GROUP BY Ki
```

Abbildung 6-1: Bereichsanfrage mit Gruppierung

Die Ergebnismenge wird bezüglich *Ki* des Schlüssels gruppiert.

6.1 Anwendungsgebiet

Das primäre Anwendungsgebiet für die Gruppierung ist die Aggregation. Die Aggregatsbildung ist ein sehr wichtiges statisches Konzept im Datenbank- und im Besonderen im DW-Bereich. Durch die Aggregatsbildung können dem Anwender verschiedene Konsolidierungsebenen auf den Daten bereitgestellt werden. Die Idee der Aggregatsbildung ist somit eine Menge von Datensätzen durch eine Kenngröße zu repräsentieren oder die Menge in Gruppen zu klassifizieren und der jeweiligen Gruppe eine Kenngröße zuzuweisen.

6.2 Eigenschaften der Gruppierung und Duplikateliminierung

Gruppierung und Duplikateliminierung können durch dieselben physischen Operatoren implementiert werden. Während die Gruppierung Tupeln zu einer Äquivalenzklasse zusammenfasst, bei denen ein oder mehrere Attribute der Relation *R* im Wert übereinstimmen, fasst die Duplikateliminierung diejenigen Tupel zusammen, die vollständig übereinstimmen. Somit ist die Duplikateliminierung ein Spezialfall der Gruppierung.

Jede Äquivalenzklasse besteht also aus einer Menge von Tupeln der Relation *R*, wobei die Äquivalenzklasse eine disjunkte Zerlegung der Relation *R* ist. Die Gruppierung und Duplikationseliminierung definiert somit eine Äquivalenzrelation auf *R*. Allgemein definiert man eine Äquivalenzrelation wie folgt:

Definition 6-1: Äquivalenzrelation, Äquivalenzklasse

Eine Relation ~ auf der Menge *R* heißt *Äquivalenzrelation* wenn sie:

- *symmetrisch*,
- *reflexiv*
- *transitiv*

ist.

$[x] := \{y \mid y \in R \wedge y \sim x\}$ heißt die *Äquivalenzklasse* von *x*. *x* ist der Repräsentant von [*x*].

Die wesentliche Eigenschaft der Gleichheit einer *Äquivalenzklasse* $[x] = [y]$ ergibt sich aus der Transitivität und Symmetrie der *Äquivalenzrelation*. Dies gilt genau dann, wenn $x \sim y$ erfüllt ist[10]. Desweiteren bildet jede *Äquivalenzrelation* auf R eine disjunkte Zerlegung von R.

Satz 6-1: *Äquivalenzrelation ⇔ Zerlegung*

Eine Äquivalenzrelation ~ über einer Menge R erzeugt eine disjunkte Zerlegung Θ von R und jede Zerlegung Θ von R definiert eine Äquivalenzrelation ~ auf R, wobei die einzelnen Teilmengen die Äquivalenzklassen darstellen.

Der Beweis ist z. B. in [CorLR90] auf Seite 82 zu finden. Betrachtet man die Gruppierung und Duplikateliminierung aus dem Blickwinkel der Äquivalenzrelation, stellen die einzelnen Gruppen $B_0, B_1, \ldots, B_{k-1}$ der Gleichung Gl: 6-2 Äquivalenzklassen dar und somit eine Zerlegung der Folge von Gl: 6-1. Da es sich bei Äquivalenzklassen um Mengen handelt, ist eine Ordnung innerhalb einer Äquivalenzklasse keine notwendige Voraussetzung. Dies ist eine wesentliche Erkenntnis, da viele Algorithmen, die auf Sortierung der Äquivalenzklassen beruhen, im Prinzip zuviel Arbeit leisten. Die einzelnen herkömmlichen Verfahren werden im Folgenden kurz vorgestellt.

6.3 Herkömmliche Gruppierungs- und Duplikateliminierungs-Algorithmen

Konventionelle Gruppierungs- und Duplikateliminierungs-Algorithmen basieren im Wesentlichen auf drei Methoden: die Schleifeniteration (Nested Loop), die Mischmethode (Merge) und die Hash-Methode. Für die Kosten werden nur die E/A-Kosten betrachtet. Es wird wieder davon ausgegangen, dass die Daten nicht im Arbeitsspeicher gehalten werden können.

Variable	Beschreibung	Definition
M	Arbeitsspeichergröße in Seiten	
P	Größe der Eingaberelation R in Seiten	
R	Relation	
P_E	Größe der Eingaberelation nach der Selektion in Seiten	
P_A	Größe der Ausgaberelation in Seiten	
$P_{ub\text{-}rq}$	Anzahl der Seiten, die durch den Bereichsanfrage-Algorithmus bewegt werden	Gl: 4-55
$P_{ext\text{-}sort}$	Anzahl der Seiten, die durch das externe Sortieren bewegt werden	Gl: 6-5
C	Anzahl der Seiten pro E/A-Einheit	Definition 3-10
v	Anzahl der Verschmelzungsebenen	Gl: 4-4
w	Anzahl der Initialläufe	Gl: 4-5
m	Verschmelzungsgrad (Fan-in)	Gl: 4-10
G	Durchschnittliche Gruppengröße	Gl: 6-7

Tabelle 6-1: Variablen, Beschreibungen und Einheiten

[10] Beweis, siehe [Stu95] S. 17

Schleifeniteration

Bei der Schleifeniteration wird eine temporäre Tabelle mit P_A Seiten erzeugt, in der für jede Äquivalenzklasse ein Ergebnistupel iterativ angelegt wird. Entsprechend zum Nested Loop Join wird für jedes Tupel x der Eingabetabelle P_E, die der äußeren Tabelle entspricht, die innere Tabelle P_A durchlaufen. Wird kein entsprechender Eintrag für $[x]$ in P_A gefunden, wird x in P_A hinzugefügt. Offensichtlich ergibt sich im ***schlechtesten Fall folgendes*** Kostenmaß[11]:

$$c_{GR\_si}(P_E) = \underbrace{2}_{lesen/schreiben} \cdot \sum_{i=1}^{P_E}(i-1) = 2\frac{(P_E-1)((P_E-1)+1)}{2} = P_E(P_E-1) = O(P_E{}^2) \qquad \text{Gl: 6-3}$$

Mischmethode

Die Mischmethode basiert auf Sortierung. Ist die Eingaberelation R nach den Gruppierungsattributen sortiert, kann die Aggregationsberechnung durch einfaches sequenzielles Lesen abgearbeitet werden, wobei für jedes Tupel einer Gruppe die Aggregationsfunktion aufgerufen wird. Dabei wird die Ergebnisrelation erzeugt. Für die E/A-Kosten ergibt sich somit:

$$c_{GR\_merge}(P_E) = \underbrace{P_E}_{lesen} + \underbrace{P_A}_{schreiben} = O(P_E) \qquad \text{Gl: 6-4}$$

Liegt die Eingaberelation nicht sortiert nach den Gruppierungsattributen vor, kommen noch die Kosten für das Sortieren hinzu. Grundsätzlich ist das Sortieren der Flaschenhals für die Anfrageverarbeitung, da das Ergebnis erst weitergereicht werden kann, wenn die Quellrelation vollständig verarbeitet worden ist. Es handelt sich also um eine blockierende Operation, d.h. es kann erst das erste Teilergebnis zurückgegeben werden, wenn die Ergebnismenge vollständig sortiert ist. Kann das Ergebnis trotz *früher Aggregation* [BitW83] nicht im Arbeitsspeicher gehalten werden, kommt es zum *externen Sortieren*. Dies stellt dann das dominierende Kostenmaß dar. Nach Gl: 4-12 ergibt sich für den ungünstigsten Fall[12]:

$$P_{ext-sort} = \underbrace{2P}_{\text{Initiallauf}} + \underbrace{2P\left\lceil \log_m \frac{P}{M} \right\rceil}_{\text{Mischphase}} = 2P\left(1+\left\lceil \log_m \frac{P}{M} \right\rceil\right) \qquad \texttt{Gl: 6-5}$$

Die E/A-Kosten eines auf Sortieren basierenden Aggregationsalgorithmus werden durch die Anzahl der Verschmelzungsebenen v sowie durch die Reduzierung der Äquivalenzklassen durch *frühe Aggregation* in den einzelnen Läufen auf den verschiedenen Verschmelzungsebenen bestimmt. Die absolute Zahl der Verschmelzungsebenen v wird durch die Aggregation nicht beeinflusst und kann somit vom externen Sortieren übernommen werden. Generell gilt für die Ausgaberelation P_A :

[11] Die Formel geht davon aus, dass P_A probes gemacht werden
[12] Es wird davon ausgegangen das $P > M$ ist.

$$P_E \geq P_A \qquad \text{Gl: 6-6}$$

Die E/A-Kosten sind somit auch abhängig von der Anzahl der Äquivalenzklassen. Die Anzahl der Äquivalenzklassen, die durch die Gruppierung gebildet werden, bestimmt die Größe der Ausgaberelation und somit die Anzahl der Seiten P_A. Gemäß [Gra93] bezeichnen wir die *durchschnittliche Gruppengröße* der Äquivalenzklasse mit:

$$G = \frac{P_E}{P_A} \qquad \text{Gl: 6-7}$$

Geht man davon aus, dass in den ersten Verschmelzungsebenen die Elemente einer Äquivalenzklasse nicht in denselben Lauf fallen, wird die Größe eines Laufs nicht beeinflusst, bis sie die Größe von P_A erreicht. P_A ist somit die maximale Größe eines Laufes. Die Größe eines Laufes auf der Verschmelzungsebene *i* ist

$$P_i = M \cdot m^i \qquad \text{Gl: 6-8}$$

Ist $P_i = P_A$ vergrößern sich die einzelnen Läufe durch die Verschmelzung nicht mehr und die *frühere Duplikateliminierung* zeigt ihre Wirkung. Wenn *v* alle Verschmelzungsebenen sind, dann bezeichnet *k* die erste Ebene, für die $P_k = P_A$ gilt.

$$\begin{aligned}
& P_k = M \cdot m^k \\
\Leftrightarrow\ & P_A = M \cdot m^k \\
\Leftrightarrow\ & \frac{P_E}{G} = M \cdot m^k \\
\Leftrightarrow\ & m^k = \frac{P_E}{M \cdot G} \\
\Leftrightarrow\ & \log_m m^k = \log_m \frac{P_E}{M \cdot G} \\
\Leftrightarrow\ & k = \log_m w - \log_m G \\
\Leftrightarrow\ & k = v - \log_m G
\end{aligned} \qquad \text{Gl: 6-9}$$

Ist *G* nicht eine Potenz von *m*, gilt $\lceil \log_m G \rceil$. Allgemein sind somit nur die letzten $\lceil \log_m G \rceil$ Verschmelzungsebenen durch die Aggregation beeinflusst, da auf den früheren Verschmelzungsebenen mehr als *G* Läufe existieren und nach Annahme die einzelnen Elemente der Äquivalenzklassen über die Läufe gleichmäßig verteilt sind. Somit sind die Kosten für die ersten Verschmelzungsebenen äquivalent zum externen Sortieren. Für die letzten $\lceil \log_m G \rceil$ Verschmelzungsebenen bleibt die Größe der Läufe konstant und entspricht P_A. Die Anzahl der betroffenen Ebenen ist:

$$v_2 = \lceil \log_m G \rceil - 1 \qquad \text{Gl: 6-10}$$

Die Reduzierung um eins ergibt sich aus der Tatsache, dass bei dem letzten Mischvorgang kein Lauf mehr erzeugt wird, sondern der Ergebnisstrom P_A. Die Anzahl der normalen Verschmelzungsebenen bezeichnen wir mit v_1 und es gilt:

$$v_1 = v - v_2 \qquad \text{Gl: 6-11}$$

Mit

$$\frac{w}{m^i} \qquad \text{Gl: 6-12}$$

werden die Anzahl der Läufe, die auf der Ebene i verschmolzen werden, berechnet. Für die E/A-Kosten c_{GR_merge} ergeben sich mit dem obigen Formelwerk nach [Gra93] dann:

$$c_{E/A-GR\_merge}(P_E, P_A) = 2 \cdot P_E \cdot v_1 + 2 \cdot P_A \cdot \sum_{i=v_1}^{v-1} \frac{w}{m^i} = 2 \cdot P_E \cdot v_1 + 2 \cdot P_A \cdot w \frac{\left(\frac{1}{m^{v_1}} - \frac{1}{m^v}\right)}{1 - \frac{1}{m}} \qquad \text{Gl: 6-13}$$

Hashing

Die dritte Methode stellt das Hashing dar. Die Grundidee besteht darin, die einzelnen Tupel der Eingaberelation R bezüglich der Gruppierungsattribute über eine Hash-Funktion in einer Hashtabelle zu speichern. Tupel derselben Äquivalenzklasse werden beim Einfügen gefunden und entsprechend der Aggregationsfunktion aggregiert. Da nur Ergebnistupel in der Hashtabelle gehalten werden, entsteht nur dann ein Überlauf, falls $P_A > M$ ist.

Die E/A-Kosten einer hashbasierten Aggregation sind wie beim Mischverfahren grundsätzlich abhängig von der Anzahl der Verschmelzungsebenen. Liegt nur noch eine Partitionierung vor, ist die Datenmenge sortiert. Genau dies tritt ein, wenn eine Partitionierung eine Größe von

$$G \cdot M \qquad \text{Gl: 6-14}$$

hat. Da sich eine Partitionierungsmenge pro Durchgang um den Faktor m verkleinert, ergibt sich für die Anzahl der Durchgänge:

$$\left\lceil \log_m \frac{P_E}{G \cdot M} \right\rceil = \left\lceil \log_m \frac{P_A}{M} \right\rceil \qquad \text{Gl: 6-15}$$

Die Kosten bei jedem Durchgang sind proportional zur Ausgaberelation P_A. Für die gesamten E/A-Kosten eines hashbasierten Aggregationsalgorithmus unter Berücksichtigung eines Faktors 2 für den Lese- und Schreibe-Puffer ergibt sich nach [Gra93]:

$$c_{E/A-GR\_hash}(P_E, P_A) = c_{E/A} \cdot 2 \cdot \left\lceil \frac{P_E}{C} \right\rceil \cdot \left\lceil \log_m \frac{P_A}{M} \right\rceil \qquad \text{Gl: 6-16}$$

Die E/A-Komplexität für Hash-Algorithmus und der Mischmethoden beträgt $O(P\log_m P)$ und wird durch die durchschnittliche Gruppengröße G stark beeinflusst. Nach [Gra93] sind die E/A-Kosten für hash- und sortier-basierte Algorithmen gleich, so dass für die theoretische Analyse für beide Algorithmen die Kostenfunktion Gl: 6-16 verwendet wird.

Antwortzeit

Neben den E/A-Kosten für das vollständige Ergebnis muss noch der Aufwand E/A-Kosten betrachtet werden, bis das erste Teilergebnis bereits vorliegt. Da es sich bei beiden Methoden um eine lokal gültige Gruppierung bzw. Duplikateliminierung handelt, kann erst in der letzen Verschmelzungsphase das Ergebnis an den Aufrufer gegeben werden. Allgemein kann man also sagen, dass ein auf Sortieren, Schleifeniteration und Hashen basierender Gruppierungsalgorithmus ein blockierender Operator ist, der erst vollständig abgearbeitet werden muss. Die Kostenfunktion lautet:

$$c_{E/A-GR\_merge\_ant}(P_E, P_A) = c_{E/A} \cdot 2 \cdot \left\lceil \frac{P_E}{C} \right\rceil \cdot \left(\left\lceil \log_m \frac{P_A}{M} \right\rceil - 1 \right) + 1 \quad \text{für } \log_m \frac{P_A}{M} \geq 2 \qquad \text{Gl: 6-17}$$

6.4 Gruppierung und Duplikateliminierung für Bereichsfragen mit dem UB-Baum

In diesem Abschnitt betrachten wir die Einsatzmöglichkeiten des UB-Baums bei der Gruppierung und Duplikatseliminierung. Hierzu werden zwei Algorithmen vorgestellt.

6.4.1 Range-Query und Sortieren oder Hashen

Die in Abbildung 6-1dargestellte SQL-Anweisung zerfällt in eine Selektionsphase, in der die Daten vom Sekundärspeicher geholt werden, und einer Gruppierungsphase. Setzt man für die Selektionsphase den Bereichsanfrage-Algorithmus des UB-Baums ein, kann entsprechend die Selektivität aus allen Dimensionen ausgenutzt werden (siehe Gl: 3-28). Dieses Zwischenergebnis kann dann mit auf Sortieren oder Hashen basierenden Verfahren weiterverarbeitet werden. Um die Kostenfunktion übersichtlich zu halten, wurde P_E nicht durch P ersetzt. Der Zusammenhang lautet:

$$P_E = P \prod_{j=1}^{d} S_j \qquad \text{Gl: 6-18}$$

Für die Kostenfunktionen ergibt sich somit:

UB-Baum

$$c_{E/A-GR-ub-sort}(P) = c_{E/A-ub}(P_E) + c_{E/A-GR-hash}(P_E) = c_{E/A}\left(P_E + 2 \cdot \left\lceil \frac{P_E}{C} \right\rceil \cdot \left\lceil \log_m \frac{P_A}{M} \right\rceil \right) \qquad \text{Gl: 6-19}$$

IOT

Wird ein IOT für den Zugriffspfad eingesetzt, kann nur die Selektivität S_k in der Sortierdimension k ausgenutzt werden. Somit erhält man folgende Kostenfunktion:

$$c_{E/A-GR-iot-sort} = c_{E/A-iot} + c_{E/A-GR-hash} = c_{E/A}\left(S_k \cdot P + 2 \cdot \left\lceil \frac{P_E}{C} \right\rceil \cdot \left\lceil \log_m \frac{P_A}{M} \right\rceil\right) \qquad \text{Gl: 6-20}$$

FTS

Wird in der Selektionsphase ein FTS eingesetzt, kann der Prefetchingfaktor C voll ausgenutzt werden. Für die E/A-Kosten ergibt sich dann:

$$c_{E/A-GR-fts-sort} = c_{E/A-fts} + c_{E/A-GR-hash} = c_{E/A}\left(\left\lceil \frac{P}{C} \right\rceil + 2 \cdot \left\lceil \frac{P_E}{C} \right\rceil \cdot \left\lceil \log_m \frac{P_A}{M} \right\rceil\right) \qquad \text{Gl: 6-21}$$

Antwortzeit

Die Kostenfunktionen entsprechen den oben genanten E/A-Kosten, wobei der logarithmische Faktor um 1 reduziert wird (siehe Gl: 6-17).

Bezeichner	Beschreibung	Definition
$c_{E/A-GR-si}$	E/A-Kosten für Gruppierung durch Schleifeniteration	Gl: 6-3
$c_{E/A-GR-merge}$	E/A-Kosten für Gruppierung durch Mischmethode	Gl: 6-4
$c_{E/A-GR-hash}$	E/A-Kosten für Gruppierung mit Hash-Methode	Gl: 6-16
$c_{E/A-GR-merge-ant}$	E/A-Kosten für das erste Teilergebnis	Gl: 6-17
$c_{E/A-GR-iot-sort}$	E/A-Kosten für Gruppierung mit Zugriffspfad IOT	Gl: 6-20
$c_{E/A-GR-ub-sort}$	E/A-Kosten für Gruppierung mit Zugriffspfad UB	Gl: 6-19
$c_{E/A-GR-fts-sort}$	E/A-Kosten für Gruppierung mit Zugriffspfad FTS	Gl: 6-21

Tabelle 6-2: Kostenmodell Gruppierung

Bewertung

Das Kostenmodell in Tabelle 6-2 zeigt, dass der Leistungsunterschied in der Selektionsphase entsteht, da die E/A-Kosten für die Gruppierung in Gleichung Gl: 6-19, Gl: 6-20 und Gl: 6-21 aus dem Faktor

$$c_{E/A-GR\_hash}(P_E, P_A) = c_{E/A} \cdot 2 \cdot \left\lceil \frac{P_E}{C} \right\rceil \cdot \left\lceil \log_m \frac{P_A}{M} \right\rceil \qquad \text{Gl: 6-22}$$

besteht. Der Zugriffspfad entscheidet somit über die Leistungssteigerung. Für eine 10%-Messung[13] auf einer 3-Dimensionlen Datenbank ergibt sich das bekannte Bild.

[13] zwei Attribute werden je auf 10% eingeschränkt, ein Attribut variiert von 0 bis 100%

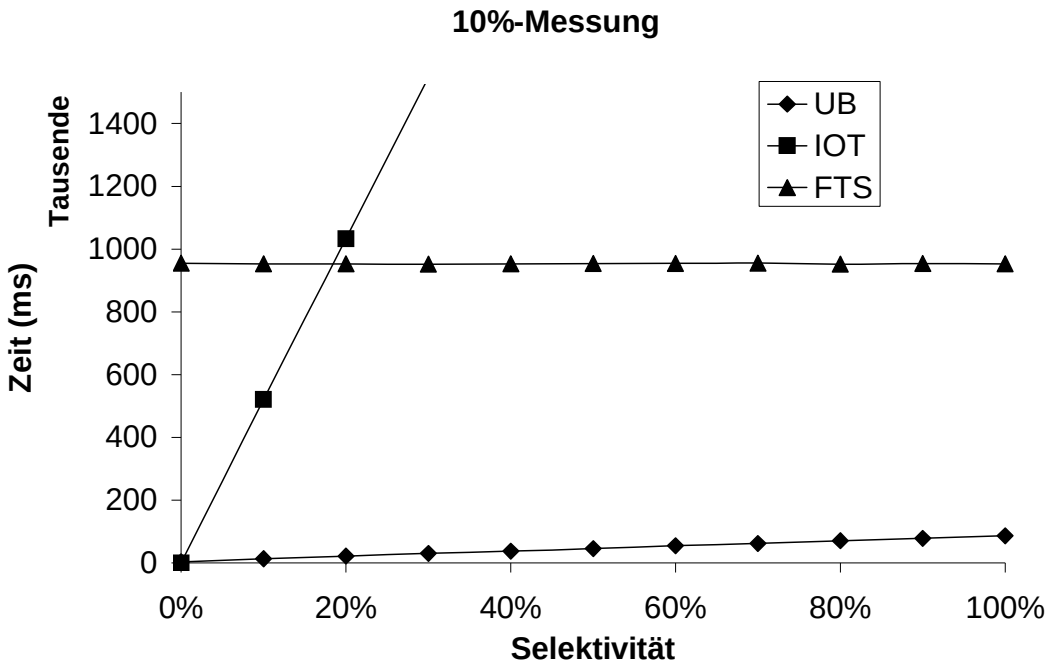


Abbildung 6-2: 10%-Messung auf einer 3-dimensionalen Datenbank

Die Datenbank enthält 2.400.000 Tupel und hat eine Größe von ca. 0,82 GB. Der UB-Baum enthält 419.891 Datenseiten.

6.4.2 UBG-Algorithmus

Der UB-Gruppierungs-Algorithmus (UBG-Algorithmus) entspricht im Wesentlichen dem Tetris-Algorithmus. Der Unterschied liegt darin, dass er die einzelnen Scheiben nicht sortiert sondern nach den Gruppierungsattributen gruppiert und auf die Äquivalenzklassen den entsprechenden Aggregationsoperator anwendet. Die Sortierdimension ist somit die Gruppierdimension. Hierbei wird die Bereichsanfrage, die durch die WHERE-Klausel spezifiziert ist, entsprechend dem Tetris-Algorithmus abgearbeitet. Dabei werden also die Regionen entsprechend der Tetris-Ordnung in den Arbeitsspeicher geladen. Die Tupel der geladenen Region werden dann in einer temporären Tabelle nach dem Gruppierungsattribut abgelegt. Ist für ein Tupel noch kein Eintrag zu finden, wird das Tupel eingefügt. Jedes Tupel in der Tabelle repräsentiert somit eine Äquivalenzklasse. Da sich durch das Lesen der Regionen bezüglich der Tetris-Ordnung in absteigender Richtung die Sweepline über den Anfragebereich in absteigender Richtung bewegt, können alle Tupel, die größer als die Sweepline sind, aus der temporären Tabelle entfernt und an den Aufrufer weiter gereicht werden. Somit können die wesentlichen Vorteile des Tetris-Algorithmus, keine blockender Operator, Reduzierung der E/A-Kosten auch bei der Aggregation genutzt werden.

Der wesentliche Vorteil des UBG-Algorithmus gegenüber dem Tetris-Algorithmus besteht darin, dass die Tupel einer Scheibe auf die Menge der Äquivalenzklassen, die in dieser Scheibe enthalten sind, reduziert werden. Somit reduziert sich noch ein Mal der Cacheverbrauch gegenüber dem Tetris-Algorithmus. Besteht die Menge der

Gruppierungsattribut nur aus einem Attribut und der UB-Baum wird bezgl. diesem Gruppierungsattribut gelesen, wird für den Cache nur ein 1 Seite benötigt, da jede Scheibe genau eine Äquivalenzklasse repräsentiert. Dies ist unabhängig von der Dimensionalität des UB-Baums. Die Größe wird jedoch noch durch die Anzahl der Sprünge in Gruppierungsrichtung beeinflusst. Da hier nur die Äquivalenzklassen gespeichert werden, ist die Anzahl gering. Eine Scheibe fügt in schlechtesten Fall s (s bezeichnet die Anzahl der Stufe einer Z-Adresse) Äquivalenzklassen ein. Die Folge ist somit, dass noch größere Datenmengen durch den UBG-Algorithmus verarbeitet werden können. Die Beschränkung liegt in diesem Fall nur in der Verwaltung von Φ und nicht in der Cachegröße.

Der UBG-Algorithmus entspricht bis auf eine Funktion dem Tetris-Algorithmus. Daneben werden im Cache nur die Äquivalenzklassen $[x_k]$, wobei k die Gruppierungsdimension ist, zwischengespeichert. Eine genaue Beschreibung des Tetris-Algorithmus ist in Kapitel 5.8 zu finden. Es wird deshalb auf eine nähere Beschreibung verzichtet. Der UBG-Algorithmus verwendet statt der Funktion *outputSortedTuples(cache, k, sweep)* die Funktion outPutGroupedTuples(cache, *k,* sweep). Diese Funktion gibt alle Gruppierungen zurück, die größer als die Sweepline sind und berechnet die entsprechenden Aggregationen.

```
UBG(UB-Tree UB,Tuple ql, Tuple qh, int k)
{
    Z-Address τ,α,β  = Z(qh); ;
    Page pg = {};
    Z-Interval phi [] = calcZ-IntervalSet(qh,ql);
    Tupel cache[] = {};
    Dim sweep = qh_k;
    while (1)
    {
       getRegionSeparators(UB,s, α,β);
       pg = getPage(UB,β);
       putMatchingTuplesToCache(cache,pg,ql,qh);
       updatePhi(phi, α,β )
       if(isempty(phi))
              break;
      τ  = nextEvent(phi,max,k);
       if(extract(τ ,k) < sweep)
       {
            sweep = extract(τ,k);
            outputGroupedTuples(cache, k,sweep);
       }
  } /*end while*/
  outputSorted(cache,k);
  return ;
}
```

Abbildung 6-3: UBG-Algorithmus

Abbildung 6-3 zeigt den UBG-Algorithmus. Die hervorgehobene Code-Zeile zeigt die Änderung gegenüber dem Tetris-Algorithmus.

6.5 Kostenanalyse

In diesem Kapitel werden die einzelnen Kosten betrachtet. Da der UBG-Algorithmus abgesehen von der Cacheverwaltung mit dem Tetris-Algorithmus übereinstimmt, können die Kostenfunktionen aus 4 übernommen werden.

6.5.1 E/A-Kosten

Die Kosten basieren auf dem Kostenmodell aus Kapitel 3.2.1.2. Die E/A-Kosten des UBG-Algorithmus gliedern sich in eine Lese- und Schreib-Phase. Die E/A-Kosten für die Lese-Phase entsprechen exakt den Kosten für die Lese-Phase des Tetris-Algorithmus aus Kapitel 4.4.4.1. Nach Gleichung Gl: 4-73 sind die E/A-Kosten für eine große Ergebnismenge

$$c_{E/A-ub} = c_{E/A} \prod_{j=1}^{d} \underbrace{n_j(d, P, y_j, z_j)}_{\text{Anzahl der schneidenden Z-Regionen in } A_j} \approx c_{E/A} P_E \qquad \text{Gl: 6-23}$$

P_E bezeichnet hierbei die Ergebnismenge der Bereichsanfrage (siehe Gleichung Gl: 4-60) und somit auch die Größe der Eingaberelation für die Gruppierung (siehe Tabelle 6-1). Die Schreibphase beim UBG-Algorithmus kann die Größe einer E/A-Operation, die C Seiten beträgt, vollständig ausnutzen. Beim Lesen wird dieser Faktor als *Prefetching* bezeichnet (siehe Definition 3-10). Somit ergibt sich für die E/A-Kosten:

$$c_{GR\_UBG} = c_{E/A} P_E + c_{E/A} \left\lceil \frac{P_A}{C} \right\rceil = c_{E/A} \left(P_E + \left\lceil \frac{P_A}{C} \right\rceil \right) \qquad \text{Gl: 6-24}$$

Die Vorteile der E/A-Kosten des UBG-Algorithmus gegenüber dem IOT und FTS entsprechen dem des Tetris-Algorithmus (siehe Kapitel 4.4.4.2). Da für die Ergebnismenge der Selektions-Klausel $P_E \geq P_A$ gilt und die Ergebnismenge der Äquivalenzklassen P_A den Faktor C voll ausnutzen kann, entstehen die Hauptkosten in der Selektionsphase.

6.5.2 Speicherplatzkosten

Der UBG-Algorithmus berechnet die Gruppierung eines Attributes A_i für eine d-dimensionale Relation R. Somit wird die Anzahl der Ergebnistupel nur durch die Ausprägung der Relation in Attribut A_i bestimmt. Dies wiederum bedeutet, dass die Größe der Ergebnismenge von der Dimensionalität der Relation R unabhängig ist.

Da der UBG-Algorithmus eine Scheibe nach der anderen verarbeitet und nur jeweils die Äquivalenzklassen zwischenspeichert, also die Tupel der aktuellen Region ρ gleich in den Cache verdichtet, ergibt sich bei einer idealisierten gleichverteilten Zerlegung Θ (siehe Definition 4-8) für die Speicherplatzkosten:

$$n_{UBG}(R) = \left\lceil \frac{\text{Anzahl der Äquivalenzklassen in Gruppierungsdimension } i}{\text{Anzahl der Scheiben in Gruppierungsdimension i}} \right\rceil \qquad \text{Gl: 6-25}$$

Die maximale Anzahl von Ausprägungen von Äquivalenzklassen ist durch den Wertebereich D_i definiert. Dieser kann bei gleichverteilten Daten dadurch unterschritten werden, falls die Kardinalität der Eingaberelation R kleiner als die Kardinalität des betroffenen Wertebereichs D_i ist. Die Anzahl der möglichen Äquivalenzklassen $|A_k|$ in der Sortierdimension k ergibt sich wie folgt:

$$|A| = \min\left(\left\{|R|, |D_k|\right\}\right) \quad \text{mit } k = \text{Sortierdimension} \qquad \text{Gl: 6-26}$$

Basierend auf dem Kostenmodell aus Kapitel 4.4.1.2 und Gl: 4-50 ist $l_k(P,d)$ die vollständige Teilungstiefe in Dimension *k*. $2^{l_k(P,d)}$ entspricht somit der Anzahl der Scheiben n_k, die in Dimension *k* existieren. Für die Anzahl der Tupel t_{UBG}, die im Arbeitsspeicher gehalten werden müssen gilt:

$$t_{UBG} = \frac{|\mathrm{A}_k|}{n_k} \qquad \text{Gl: 6-27}$$

In Gleichung Gl: 6-27 muss noch der Parameter $|A_k|$ näher bestimmt werden. Die Anzahl der Äquivalenzklassen repräsentiert die Anzahl der Ergebnistupel bei einer Selektivität von 100%. Ist P_A die Anzahl der Ergebnisseiten und κ die Kapazität einer Seite in Tupel und S_k die Selektivität der Sortierdimension, ergibt sich für die gleichverteilten Äquivalenzklassen:

$$|\mathrm{A}| = \frac{P_A \cdot \kappa}{S_k} = \frac{P_E}{S_k \cdot G} \cdot \kappa = \frac{P \cdot S_k \cdot \kappa}{S_k \cdot G} = \frac{P \cdot \kappa}{G} \qquad \text{Gl: 6-28}$$

Bei Gleichung Gl: 6-28 wurde Gleichung Gl: 6-7, die die durchschnittliche Gruppierungsgröße definiert, verwendet. Da P_E die Größe der Eingaberelation in die Gruppierung und *P* die Größe der Tabelle bezeichnet, muss noch die Selektivität in der Sortierdimension berücksichtigt werden. Da *G* das Verhältnis der Eingabe P_E zur Ausgabage P_A ist und P_E das Produkt der Scheiben in jeder Dimension ist, gilt:

$$G = \frac{\prod_{i=0}^{d} n_i}{n_k} \qquad \text{Gl: 6-29}$$

Die Anzahl der Tupel t_{UBG} im Arbeitsspeicher ist somit:

$$t_{UBG} = \frac{P \cdot \kappa}{G \cdot n_k} = \frac{\prod_{i=0}^{d} n_i \kappa}{\frac{\prod_{i=0}^{d} n_i}{n_k} n_k} = \frac{\prod_{i=0}^{d} n_i \kappa}{\prod_{i=0}^{d} n_i} = \kappa \qquad \text{Gl: 6-30}$$

Das entspricht somit genau der Kapazität einer Seite. Für eine idealisierte gleichverteilte Zerlegung Θ (siehe Definition 4-8) sind die Speicherplatzkosten:

$$cache_{UBG}(R) = 1 \qquad \text{Gl: 6-31}$$

Hierbei ist jedoch zu beachten, dass nur nach der Sortierdimension *k* gruppiert wird. Für den allgemeinen Fall führt jedes weitere Gruppierungsattribut zu einer Cachevergrößerung. Die Vergrößerung pro zusätzlichen Gruppierungsattribut wird durch die jeweilige Teilungstiefe $l_{\downarrow/d}$ (siehe Gl: 4-49) des Gruppierungsattributes bestimmt. Sei nun *n* die Anzahl der Gruppierungsattribute. Die Speicherplatzkosten für den allgemeinen Fall sind somit:

$$cache_{UBG}(P) = 2^{\frac{\log_2 P}{d}} \cdot (n-1) \qquad \text{Gl: 6-32}$$

6.5.3 Antwortzeit

Da der UBG-Algorithmus bei der Verarbeitung dem Tetris-Algorithmus entspricht, kann das erste Ergebnis erst geliefert werden, wenn die erste Scheibe S^{-t} vollständig abgearbeitet worden ist. Somit können die Ergebnisse aus Kapitel 4 und Kapitel 5 übernommen werden.

6.6 Zusammenfassung

In diesem Kapitel wurde der UBG-Algorithmus vorgestellt. Er entspricht bis auf eine Funktion dem Tetris-Algorithmus. Der Unterschied liegt in der Cachverwaltung. Im Cache werden nur Äquivalenzklassen $[x_k]$, wobei *k* die Gruppierungsdimension ist, zwischengespeichert. Die Kostenanalyse zeigt, dass die Vorteile der E/A-Kosten des UBG-Algorithmus gegenüber dem IOT und FTS dem des Tetris-Algorithmus entsprechen. Die wesentlichen Kosten entstehen in der Selektionsphase. Durch die Aggregation kann es zu einer Verdichtung der Daten kommen, so dass $P_E \geq P_A$ gilt. Dies führt zu einer Reduzierung der E/A-Kosten gegenüber dem Tetris-Algorithmus. Ein wesentlicher Vorteil liegt in der Reduzierung der Speicherkosten. Für gleichverteilte Daten benötigt der UBG-Algorithmus einen Cache von einer Seite, wenn nur nach der Sortierdimension gruppiert wird. Allgemein kann dieses Prinzip auch für den SRQ-Algorithmus eingesetzt werden.

There is nothing more difficult to take in hand more perilous to conduct, or more uncertainin its success than to take the lead in the introduction of a new order or things

Niccolò Machiavelli, Teh Prince (1513)

7 Ausnutzung von Vorsortierung beim Sortieren großer Datenmengen

Im Bereich der Anfrageverarbeitung werden häufig Zwischenergebnisse erstellt, auf denen eine Ordnung existiert, wie z.B. bei einem *Sort-Merge-Verbund* [Gra93], bei dem das (Zwischen-)Ergebnis sortiert nach den Verbundattributen vorliegt. Ein weiteres Beispiel ist das sequenzielle Lesen eines Indexes. Die Praxis selbst zeigt, dass z.B. Massendaten, die in ein DW durch Massenladen eingebracht werden sollen, üblicherweise nach der Dimension *Zeit* vorsortiert sind. Beim Massenladen werden die benötigten Indexe oder Zugriffsmethoden, die zuvor vom Datenbankadministrator in Form eines Datenbankschemas festgelegt wurden, erzeugt. Die Vorsortiertheit in Daten kann in der Anfrageverarbeitung herangezogen werden, um die Ergebnisse weiterer Operatoren, wie Verbund, Duplikateliminierung, Projektion oder Gruppierung effizient zu berechnen. Dies gilt besonders beim Massenladen zur Erzeugung von Zugriffsmethoden.

Wie in den vorangegangenen Kapiteln gezeigt wurde, werden Bereichsanfragen und Bereichsanfragen in Kombination mit Sortierung besonders gut durch den UB-Baum unterstützt. So stellt sich die Frage, wie man die Indexstruktur UB-Baum effizient erzeugen kann. Es wurden bereits einige Arbeiten im Bereich Massenladen für mehrdimensionale Datenstrukturen, wie z.B. R-Bäume [BöhK99, KamF93, LeuEL97, LoR95, RouL85], Grid-Files [LeuN97] und Quad-Trees [HjaSS97], veröffentlicht. Diese Algorithmen haben eine E/A-Komplexität von $O(P \cdot \log P)$. Das ist dadurch bedingt, dass die Algorithmen die Vorsortiertheit nicht ausnutzen und somit externes Sortieren für die Eingabedaten benötigen.

Der UB-Baum kann die für den B-Baum bekannten Massenlade-Algorithmen (packing algorithm) ohne große Änderungen benutzen, da er als Speicherstruktur einen B^*-Baum verwendet. Ein übliches Verfahren zur Erzeugung eines Speicherplatz optimalen B-Baums besteht darin, die Daten bezüglich der Schlüsselattribute extern zu sortieren und sie dann in die Sekundärspeicherseiten zu schreiben. Hierbei wird jede Seite bis zu dem benötigten Füllungsgrad mit Daten bestückt.

Dieses Kapitel beschäftigt sich mit der dritten Fragestellung dieser Arbeit, d.h. wie man aus einer gegebenen *Quellordnung* eine Z-Ordnung als *Zielordnung* effizient erzeugt und auf den Sekundärspeicher schreibt. Formal können wir das Problem also wie folgt beschreiben:

Gegeben ist eine Folge von Tupeln:

$$\langle x_{(0)}, x_{(1)}, x_{(2)}, \ldots, x_{(n-1)} \rangle \text{ mit } x_{(0)j} \leq x_{(1)j} \leq x_{(2)j} \leq \ldots \leq x_{(n-1)j} \qquad \text{Gl: 7-1}$$

, d.h. die Folge ist nach Attribut A_j sortiert. Der Sortieralgorithmus soll nun eine Permutation der Folge erzeugen, für die gilt:

$$\langle x_{\pi(0)}, x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n-1)} \rangle \text{ mit } x_{\pi(0)} \prec x_{\pi(1)} \prec x_{\pi(2)} \prec \ldots \prec x_{\pi(n-1)} \qquad \text{Gl: 7-2}$$

Da Symbol „$\prec$“ bezeichnet die Z-Ordnung.

In diesem Teil der Arbeit wird der TempTris-Algorithmus eingeführt, der das beschriebene Problem effizient löst. Es handelt sich hierbei um eine Technik, die eine mehrdimensionale, disjunkte Partitionierung des UB-Baums[14] ohne externes Sortieren in linearer Zeit erstellt. Hierzu wird ein genügend großer, aber dennoch moderater Zwischenspeicher (Cache) benötigt. Für gleichverteilte Daten benötigt der TempTris-Algorithmus nur $\sqrt[d]{P^{d-1}}$ Speicher, somit wächst der temporäre Speicherbedarf nur sublinear. Der wesentliche Vorteil des TempTris-Algorithmus gegenüber Sortierverfahren, wie dem externen Sortieren durch Verschmelzen [Knu98v3], besteht in der Ausnutzung der Vorsortiertheit der Quelldaten. Der TempTris-Algorithmus verallgemeinert somit den Massenlade-Algorithmus (*bulk loading*) für UB-Bäume [FenKM00], indem er die Vorsortiertheit der Eingabedaten berücksichtigt. Der TempTris-Algorithmus gehört deshalb zur Klasse der *adaptiven* Sortieralgorithmen [EstW92], d.h. die Kosten, die entstehen, wenn eine Folge von Tupel sortiert wird, hängen nicht nur von der Größe *n* der Folge ab, sondern auch vom Grad der Vorsortiertheit. Demzufolge sind z.B. die *E/A*-Kosten geringer als $O(P \log P)$, die für den allgemeinen Fall entstehen [Meh84]. Der nach einer Dimension sortierte Tupelstrom stellt eine partielle Ordnung dar, die ausgenutzt wird, um z.B. die mehrdimensionale Z-Ordnung zu erzeugen.

Eine typische Anwendung für den TempTris-Algorithmus ist die Erzeugung von clusternden (bündelnden) Indexen beim Massenladen. Daneben ist der TempTris-Algorithmus sowohl für die Erzeugung von permanenten Tabellen als auch für die Materialisierung von temporären Zwischenergebnissen in Anfrageverarbeitungsplänen besonders gut geeignet. Im Gegensatz zu allen früheren Ansätzen nützt die hier vorgestellte Methode die Tatsache aus, dass die Eingabetupel mindestens bezüglich eines Attributs sortiert sind, um die Z-Partitionierung zu erzeugen. Im Bereich der Anfrageverarbeitung können sortierte Tupelströme mit Hilfe des TempTris-Algorithmus ohne externes Sortieren in materialisierte UB-Bäume überführt werden. Diese können z.B. dann mit Hilfe des Tetris-Algorithmus nach jedem beliebigen Attribut sortiert ausgegeben werden (siehe Kapitel 5, Kapitel 4).

Als Anwendungsbeispiel für den TempTris-Algorithmus wird in dieser Arbeit eine neue Verarbeitungstechnik zur Berechnung von Aggregationsgittern (aggregation lattice). vorgestellt, mit denen der CUBE-Operator effizient berechnet werden kann (siehe Kapitel 9.2).

[14] Der Algorithmus ist so generisch, dass auch andere Partitionierungen, wie z. B. die Hilbert-Kurve oder der Gray-Code, verwendet werden können.

Zur Bewertung des *TempTris*-Algorithmus geben wir ein Kostenmodell an. Daneben wird ein theoretischer Vergleich mit dem traditionellen externen Sortier-Verfahren *Sort-Merge*, dam Standard-Sortierverfahren im Datenbankbereich, angegeben.

Der TempTris-Algorithmus ist der inverse Operator zum Tetris-Algorithmus. Diese Tatsache ermöglicht es, viele analytische und experimentelle Ergebnisse aus Kapitel 4 und Kapitel 5 zu übernehmen. Hier sind besonders die Größe des Caches sowie die Tetris-Ordnung zu nennen.

Der Rest dieses Kapitels ist folgendermaßen gegliedert: Abschnitt 7.1 führt zunächst in die Grundidee des TempTris-Algorithmus ein. Nach der Einführung des formalen Modells, das eine Erweiterung von Abschnitt 4.3.1 ist, wird der generische Algorithmus vorgestellt. Dieser Abschnitt schließt mit einem Korrektheitsbeweis und grundlegenden Leistungsmerkmalen. Abschnitt 7.1.6 zeigt eine mögliche Implementierung des TempTris-Algorithmus, basierend auf der Zugriffsmethode UB-Bäume. In diesem Abschnitt wird speziell auf Implementierungsaspekte eingegangen. Abschnitt 0 führt ein Kostenmodell ein und schließt mit einer Kostenanalyse. In Abschnitt 7.4 werden Messungen dargestellt.

7.1 TempTris mit Regionen

7.1.1 Grundidee

Bei der Beschreibung des Algorithmus gehen wir ohne Beschränkung der Allgemeinheit davon aus, dass die Daten in aufsteigender Reihenfolge bezüglich der vorsortierten Dimension *k* eingefügt werden.

Die grundlegende Idee des TempTris-Algorithmus ist die Erzeugung einer neuen Zielordnung auf den Eingabedaten *R* unter Ausnutzung der Quellordnung. Handelt es sich hierbei um eine mehrdimensionale Zielordnung, wie z.B. die Z-Ordnung, kann damit der mehrdimensionale Zugriffspfad UB-Baum effizient erstellt werden. Basierend auf dem bekannten *Segmentschnitt-Problem* verwenden wir eine Art *Plane-Sweep*-Algorithmus [PreS85], um zwischen dem *stabilen Bereich* der Partitionierung (der bereits auf den Sekundärspeicher geschrieben werden kann) und dem *dynamischen Bereich* zu unterscheiden. Da der dynamische Bereich noch Daten aufnehmen kann, entstehen zusätzliche E/A-Kosten, falls dieser bereits auf dem Sekundärspeicher ausgelagert wird. Der *dynamische Bereich* wird deshalb im Cache gehalten. Die Richtung der Trennebene (*Plane*) wird durch das Attribut, nachdem der Eingabestrom $\langle R \rangle$ sortiert ist, bestimmt. Ist z.B. ein 3-dimensionaler Tupelstrom mit den Dimensionsachsen A_1 ,A_2 ,A_3. nach Attribut A_1 sortiert, bewegt sich die Trennebene orthogonal zu A_2, A_3 entlang der Dimension A_1. Grundsätzlich führt der TempTris-Algorithmus folgende Schritte durch:

- Füge das Tupel bezüglich der Zielordnung in die entsprechende dynamische Region ein und teile sie, falls notwendig.

- Führe eine dynamische Region zu einer stabilen Region über, sobald sie den dynamischen Bereich nicht mehr schneidet.

7.1.2 Formales Modell

In diesem Abschnitt wird das formale Modell, basierend auf dem formalen Modell von Abschnitt 4.3.1, eingeführt.

Betrachten wir nun einen 16x16 Basisraum Ω mit A_1 als horizontaler Dimension, A_2 als vertikaler Dimension und dem Ursprung (0,0) in der linken oberen Ecke. Abbildung 7-1.a zeigt eine Regionszerlegung Θ von 7 Regionen $\{\rho_1, \rho_2, \rho_3 \rho_4, \rho_5, \rho_7, \rho_8\}$ und Abbildung 7-1.b eine zusätzliche Region ρ_6, die durch die Teilung der Region ρ_7 entstanden ist. Für die Regionen ρ_7 und ρ_8 sind die jeweils gespeicherten Tupel eingetragen.

Die Sweep-Hyperebene $\Psi_{k,c}$ (Definition 4-2) ist eine Trennebene, die in der Sortierdimension auf einen Punkt eingeschränkt ist. Abbildung 7-1.a zeigt die Trennebene $\Psi_{A1,10}$ mit dem aktuell eingefügten Tupel $t_a = (t_1, t_2) = (10,12)$, wodurch die Position der Trennebene bestimmt wird.

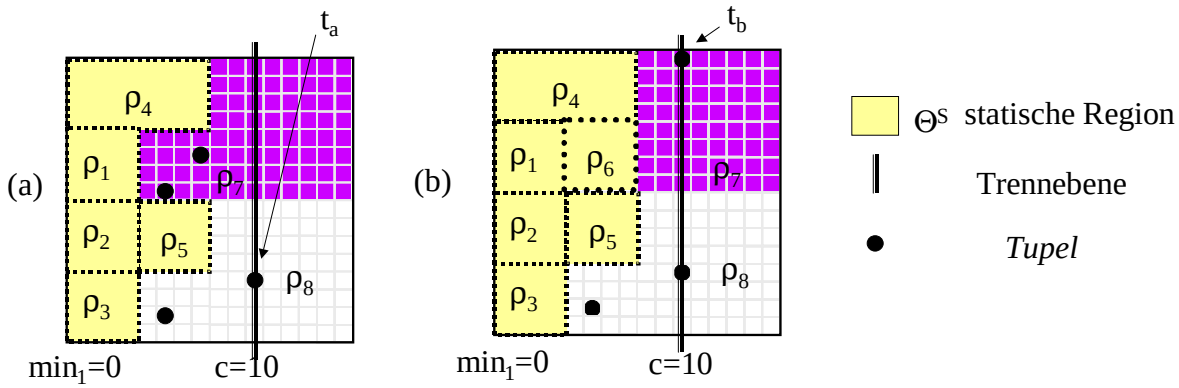


Abbildung 7-1: Terminologie

Für die folgende Erklärung nehmen wir an, dass A_k die Dimension ist, nach der die Eingabedaten vorsortiert sind. Gegeben ist somit eine sortierte Folge $(\langle R\rangle,\leq,k)$. t_k ist die Position der Sweep-Hyperebene $\Psi_{k,t}$, d.h. der Wert des Attributs A_k des i-ten Tupels $t = get(\langle R\rangle,i)$ des Eingabestroms.

Wie bereits oben beschrieben, unterteilt der TempTris-Algorithmus den Basisraum in zwei disjunkte Bereiche, den *stabilen Bereich* B^S und den *dynamischen Bereich* B^D, die durch die Trennebene $\Psi_{k,c}$ bestimmt werden. Hierbei ist der *stabile Bereich* B^S der Teil des Basisraums Ω, der von der Trennebene $\Psi_{k,c}$ durch Einfügen der Tupel der sortierten Folge $(\langle R\rangle,\leq, k)$ überstrichen worden ist und somit im Bereich $[min, c\text{ -}1]$ der Sortierdimension liegt. Formal ergibt sich folgende Definition:

Definition 7-1: stabiler Bereich B^S

Der *stabile Bereich* $B^S_{k,c}$ ist ein Teilraum von Ω in Abhängigkeit von $\Psi_{k,c}$, für den gilt:

$$B^S_{k,c} = \{x \mid x \in \Omega \wedge x_k < c\}$$

Da der dynamische Bereich B^D das Komplement zum stabilen Bereich ist, definieren wir diesen wie folgt:

Definition 7-2: dynamischer Bereich B^D

Der *dynamische Bereich* $B^D_{k,c}$ ist ein Teilraum von Ω in Abhängigkeit von $\Psi_{k,c}$ für den gilt:

$$B^D_{k,c} = \{x | x \in \Omega \wedge x_k \geq c\}$$

Basierend auf unserem Regionskonzept führen wir nun die Begriffe *stabile Regionsmenge, stabile Seitenmenge, dynamische Regionsmenge* sowie *dynamische Seitenmenge* ein. Die Regionen, die vollständig im stabilen Bereich liegen, werden *stabile Regionen,* die anderen als *dynamische Regionen* bezeichnet. Die Menge der stabilen Regionen bezeichnen wir als *stabile Regionsmenge* Θ^S, die der dynamischen Regionen *dynamische Regionsmenge* Θ^D. Formal definieren wir die Begriffe wie folgt:

Definition 7-3: stabile Regionsmenge Θ^S

Eine *stabile Regionsmenge* Θ^S ist eine Teilmenge einer Regionszerlegung Θ des Basisraums Ω bezüglich der Trennebne $\Psi_{k,c}$ mit der Eigenschaft:

$$\Theta^S = \{\rho | (\rho \in \Theta) \wedge (\rho \cap B^S_{k,c}) = \rho\}$$

Die *stabile Seitenmenge* $P_R{}^S(\Theta^S)$ ist die Menge an Seiten, die mit den Regionen von Θ^S korrespondieren.

Definition 7-4: stabile Seitenmenge $P^S{}_R$

Eine *stabile Seitenmenge* P^S_R ist eine Teilmenge der Seitenmenge P_R mit der Eigenschaft:

$$P^S_R(\Theta^S)\ \{p \mid p \leftrightarrow \rho \wedge \rho \in \Theta^S\}$$

Die dynamische Regionsmenge Θ^D und Seitenmenge P^D_R definieren wir wie folgt:

Definition 7-5: dynamische Regionsmenge Θ^D

Eine *dynamische Regionsmenge* Θ^D ist eine Teilmenge einer Regionszerlegung Θ des Basisraums Ω bezüglich der Trennebne $\Psi_{k,c}$ mit der Eigenschaft:

$$\Theta^D = \{\rho | (\rho \in \Theta) \wedge (\rho \cap B^D_{k,c}) \neq \varnothing\}$$

Die *dynamische Seitenmenge* $P_R{}^D$ ist die Seitenmenge, die mit der dynamischen Regionsmenge korrespondiert.

Definition 7-6: dynamische Seitenmenge $P_R{}^D$

Eine *dynamische Seitenmenge* P^D_R ist eine Teilmenge der Seitenmenge P_R mit der Eigenschaft:

$$P^S_R(\Theta^D) = \{p \mid p \leftrightarrow \rho \wedge \rho \in \Theta^D\}$$

Die Regionen $\{\rho_1, \rho_2, \rho_3 \rho_4, \rho_5\}$in Abbildung 7-1.a stellen die *stabile Regionsmenge* Θ^S und $\{\rho_7, \rho_8\}$ dementsprechend die *dynamische Regionsmenge* Θ^D mit $c = 10$ dar.

Während eines Durchgangs des TempTris-Algorithmus wird die stabile Regionsmenge allmählich auf den Sekundärspeicher geschrieben. Hierzu werden die dynamischen Regionen Θ^D im Arbeitsspeicher gehalten. Beim Übergang einer Region vom dynamischen zum statischen Zustand wird sie aus dem Cache entfernt und auf den Sekundärspeicher geschrieben. Dieser Vorgang ist der Übergang von Abbildung 7-1.a nach Abbildung 7-1.b. Die Region ρ_6, die durch eine Teilung der Region ρ_7 entstanden ist, bezeichnen wir als Δ-Menge bzw. Δ-Seitenmenge. Sie ist genau die Differenzregionsmenge, in der sich zwei *stabile Regionsmengen* Θ^S unterscheiden. Da eine Änderung der Mengen nur durch das Einfügen des nächsten Tupels aus der sortieren Folge $(\langle R\rangle,\leq,k)$ verursacht wird, spezifizieren wir einen speziellen Zustand der *stabilen* und *dynamischen Regionsmenge* durch das *i-te* Tupel $t_{(i)}$ der sortierten Folge $(\langle R\rangle,\leq,k)$ und schreiben Θ_i^S, Θ_i^D.

Definition 7-7: Δ-Menge, Δ_i-Menge

Geben ist eine sortierte Folge $(\langle R\rangle,\leq,k)$. Die Δ-Menge ist die Menge der Regionen, für die gilt:

$$\Delta = \Theta_i^S / \Theta_j^S \quad \text{mit } j < \text{i und } i,j \in \{0,...,|R|\text{-}1\}$$

wobei das i,j das *i-te* bzw. das *j-te* Tupel der sortierten Folge bezeichnet.

Die Δ_i-Menge ist die Menge der Regionen, für die gilt:

$$\Delta_i = \begin{cases} \Theta_i^S / \Theta_{i-1}^S & , i \in \{1,...,|R|-1\} \\ \emptyset & , i = 0 \end{cases}$$

Definition 7-8: Δ-Seitenmenge

$$P_\Delta(\Delta) = \{p \mid p \leftrightarrow \rho \wedge \rho \in \Delta\}$$

Während Basisraum, Region, Regionszerlegung, Sweep-Hyperebene, dynamischer Bereich, stabiler Bereich und Delta-Menge Teilmengen des *d*-dimensionalen Basisraums Ω sind, stellen Relation, Seitenmenge, dynamische Seitenmenge, stabile Seitenmenge, Δ-Seitenmenge und Seite Mengen von Tupeln dar. Die Abbildung des Raumes auf Tupelmengen ist durch die Funktion *seite*(ρ) definiert, da für jede *Region* $\rho \leftrightarrow p$ gilt (siehe Definition 3-23). Der Zusammenhang der einzelnen Teilräume und Teilmengen ist in Tabelle 7-1 zusammengefasst.

Raum		Relation	
Ω	Basisraum	R	Relation
Θ	Regionszerlegung	P_R	Seitenmenge
$\Psi_{k,c}$	Sweep-Hyperebene		
B^D	dynamischer Bereich		
B^S	stabiler Bereich		

Θ^S	stabile Regionsmenge	P_R^S	stabile Seitenmenge
Θ^D	dynamische Regionsmenge	P_R^D	dynamische Seitenmenge
Δ	Delta-Menge	P_Δ	Δ-Seitenmenge
ρ	Region	p	Seite

Tabelle 7-1: Raum-Tupel-Modell

7.1.3 Algorithmus

Gegeben sei eine *sortierte Folge* ($\langle R\rangle,\leq,k$), die nach Attribut A_k sortiert ist. Zu Beginn besteht die *dynamische Regionsmenge* Θ^D aus einer Region, die den gesamten Basisraum Ω abdeckt. Die *stabile Regionsmenge* Θ^S ist leer. Jedes Tupel $t_{(i)}$ der sortierten Folge ($\langle R\rangle,\leq, k$) wird iterativ in eine Region ρ der dynamischen Regionsmenge bezüglich der Zielordnung (Z-Ordnung) eingefügt. Das zuletzt eingefügte Tupel $t_{(i)}$ bestimmt die aktuelle Trennebene Ψ, die die stabile Regionsmenge Θ^S von der dynamischen Regionsmenge Θ^D trennt. $t_{(i),k}$ bezeichnet den Wert des Attributs A_k von Tupel $t_{(i)}$. Bei fortwährendem Einfügen von Tupeln aus der sortierten Folge ($\langle R\rangle,\leq,k$) in den Basisraum Ω bewegt sich die Sweep-Hyperebene vorwärts. Hierbei wächst die dynamische Regionsmenge Θ^D genau dann, wenn eine Region ρ in zwei Regionen ρ_1 und ρ_2 wegen eines Überlaufs geteilt wird und beide Regionen den dynamischen Bereich schneiden. Liegt eine dynamische Region vollständig im stabilen Bereich wird sie stabil und aus dem Cache entfernt (*cache flushing*). Wenn das letzte Tupel t der sortierten Folge ($\langle R\rangle, \leq k$) eingefügt wurde, wird der Cache vollständig stabil und demzufolge auf den Sekundärspeicher geschrieben. Zu diesem Zeitpunkt ist die mehrdimensionale Partitionierung der Relation beendet. Eine Pseudo-Implementierung, die die wesentlichen Aspekte veranschaulicht, ist in Abbildung 7-2 zu finden.

Marke	TempTris-Algorithmus

```
                Eingabe:  (⟨R⟩, ≤ ,k)   : sortierte Folge
                Ausgabe:  Θ^S           : stabile Regionsmenge
                {
[init]              tuple t;                    // das aktuelle Tupel, das in den dynamischen
                                                // Bereich eingefügt wird.
                    region ρ, ρ1,ρ2;
                    region-set Θ^D = {Ω};       //Zu Beginn besteht die dynamische Regionszerlegung
                                                //aus einer Region, die ganz Ω abdeckt
                    Region-set Δ = ∅;
                    Θ^S = ∅;                    // Die stabile Regionszerlegung ist leer
                    Int i = 0;
                    // Durchlaufe die Schleife, bis die sortierte Folge (⟨R⟩, ≤ ,k) vollständig
                    //in den dynamischen Bereich eingefügt worden ist. |R| bezeichnet die Anzahl
                    // der Tupel in der Relation R
[loop]              while(i < |R|)
                    {
[insert]                t =  get(⟨R⟩,i);
                        ι = ι + 1;
                        ρ  = {ρ | ρ∈ Θ^D  and t ∈ ρ }      // Suche die Region, die t enthält

                        page(ρ) =  page(ρ) ∪{t};          // Füge t in die entsprechende
                                                          // Seite ein

                        // Teste, ob durch das Einfügen des Tupels t  in die Seite page(ρ)
                        //ein Überlauf entstanden ist. Falls ja, teile ρ  in ρ1 und ρ2
                        If(|page(ρ)| > max_number_of_elements)
                        {
                            split(ρ ,ρ1,ρ2);
                            Θ^D = (Θ^D \ {ρ }) ∪ {ρ1 } ∪ {ρ2 }
                        }
                        // Berechne die aktuelle Delta-Menge. Hierbei handelt es sich um
                        // alle diejenigen Regionen, die sowohl in der dynamischen
                        // Regionsmenge und vollständig im stabilen Bereich liegen
                        // Eine Region wird stabil durch eine Teilung in zwei Regionen
                        // oder durch die Bewegung der Trennebene, die durch den Wert
                        // des Attributs A_k  des aktuellen Tupels t bestimmt wird.
[delta-set]             Δ = { ρ | ρ ∈ Θ^D and  ρ ∩ B^S_{k,t_k}  = ρ }

                        // Schreibe die neuen stabilen Regionen auf den Sekundärspeicher
[write]                 if(Δ ≠ ∅)
                        {

                            Θ^S = Θ^S ∪Δ      // Schreibe die Seiten, die zur Δ-Menge
                                              // gehören, auf den Sekundärspeicher

                            Θ^D = Θ^D \ Δ      // Entferne die Seiten der Δ-Menge
                                               //aus dem Arbeitsspeicher
                        }
[end-loop]          }
[flush]             Θ^S = Θ^S ∪ Θ^D   // Schreibe alle Seiten, die zur dynamischen
                                      //Seitenmenge gehören, auf den Sekundärspeicher
                }
```

Abbildung 7-2 TempTris-Algorithmus

7.1.4 Korrektheit des Algorithmus

In diesem Abschnitt wird ein formaler Korrektheitsbeweis für den Algorithmus, der in Abschnitt 7.1.3 vorgestellt wurde, angegeben. Die formale Spezifikation des Massenlade-Algorithmus lautet:

Gegeben ist eine Folge von Tupeln:

$$\langle x_{(0)},x_{(1)},x_{(2)},\ldots,x_{(n-1)}\rangle \text{ mit } x_{(0)j} \le x_{(1)k} \le x_{(2)k} \le \ldots \le x_{(n-1)k} \qquad \text{Gl: 7-3}$$

d.h. die Folge der Tupel wächst monoton bezüglich der Dimension *k.* Der TempTris-Algorithmus soll nun eine Permutation der Folge erzeugen, für die gilt:

$$\langle x_{\pi(0)},x_{\pi(1)},x_{\pi(2)},\ldots,x_{\pi(n-1)}\rangle \text{ mit } x_{\pi(0)} \prec x_{\pi(1)} \prec x_{\pi(2)} \prec \ldots \prec x_{\pi(n-1)} \qquad \text{Gl: 7-4}$$

Des Weiteren gilt die Randbedingung, dass die Tupel in einem UB-Baum gespeichert werden und somit eine stabile Seitenmenge P_R^S bzw. eine stabile Regionsmenge Θ^S bezüglich der Z-Kurve entsteht.

Satz 7-1: Korrektheit des sortierten Schreibens

Gegeben ist eine sortierte Folge ($\langle R\rangle$,≤,*k*) mit der Sortierdimension *k*. Dann erzeugt der Algorithmus in Abbildung 7-2 eine stabile Regionsmenge Θ^S bzw. eine stabile Seitenmenge P_R^S bezüglich der Z-Kurve, wobei jede Region genau einmal auf den Sekundärspeicher geschrieben wird.

Beweis:

Betrachten wir zunächst die WHILE-Schleife. Vor der Ausführung der Schleife ist die stabile Regionsmenge Θ^S_{init} leer, die dynamische Regionsmenge Θ^D_{init} besteht aus einer Region, die den ganzen Basisraum abdeckt. Die Iterationsvariable *i* hat den Wert 0. Hierzu wird folgende Invariante, die bei jedem Iterationsschritt erfüllt sein muss, definiert.

Invariante:

Die *i-te* stabile Regionsmenge Θ_i^S ist die Vereinigung der paarweise disjunkten Δ-Mengen:

$$\Theta_i^S = \Delta_0 \cup \Delta_1 \cup \Delta_2 \cup \Delta_3 \ldots \cup \Delta_i = \bigcup_{j=0}^{i} \Delta_j$$

1.Terminierung
Zunächst wird die Terminierung betrachtet. Da bei jedem Durchgang der WHILE-Schleife *i* um 1 erhöht wird und die sortierte Folge beliebig groß, aber endlich ist, terminiert der Algorithmus nach |*R*| Durchgängen.

2. Induktionsanfang *i = 0*
Die Invariante muss zu Beginn des Algorithmus wahr sein. Da die stabile Regionsmenge zu Beginn leer ist, ist die Invariante erfüllt.

3. Induktionsschritt i ⇒ i+1
Betrachten wir nun den Induktionsschritt, d.h., die Invariante muss bei einem Interrationsschritt wahr bleiben. Es ist also nun zu zeigen, dass

$$\bigcup_{j=0}^{i} \Delta_j \Rightarrow \bigcup_{j=0}^{i+1} \Delta_j$$

Bei der Ausführung der WHILE-Schleife wird das nächste Tupel

$$t_{(i+1)} = \text{get}(\langle R \rangle, (i+1))$$

in eine Z-Region ρ mit $Z(t) \in \rho$ eingefügt. Die Region ρ liegt im *dynamischen Bereich* und somit in Θ^D, da nach Definition 7-2 die Sweep-Hyperebene $\Psi_{k,c}$ durch das Tupel mit $c = t_{(i+1),k}$ bestimmt wird. Diese Region wird gegebenenfalls geteilt,

```
If(|page(ρ)| > max_number_of_elements)
{
        split(ρ ,ρ1,ρ2);
        Θ^D = (Θ^D \ {ρ}) ∪ {ρ1} ∪ {ρ2}
}
```

d.h. die Aufteilung des Raums ρ in zwei Teilräume ρ_1 und ρ_2 mit $\rho = \rho_1 \cup \rho_2$.

Die Delta-Menge Δ_{i+1} ist genau die Menge der Regionen, die in der dynamischen Regionszerlegung Θ_{i+1}^D und vollständig im stabilen Bereich $B_{k,t_{(i),k}}^S$ enthalten sind. Dies entspricht

$\Delta_{i+1} = \{ \rho \mid \rho \in \Theta_i^D \wedge \rho \cap B_{k,t_k}^S = \rho \}$.

Da nach Voraussetzung

$\Theta_i^S = \bigcup_{j=0}^{i} \Delta_j$ gilt und Δ_{i+1} disjunkt zu Θ_i^S ist, folgt:

$$\Theta_{i+1}^S = \left(\bigcup_{j=0}^{i} \Delta_j \right) \cup \Delta_{i+1} = \bigcup_{j=0}^{i+1} \Delta_j$$

Somit ist die Invariante erfüllt.

Da die Tupel $t_{(i)}$ monoton steigend bezüglich der Sortierdimension k in den Basisraum Ω eingefügt werden, bewegt sich die Sweep-Hyperebene $\Psi_{k,c}$ mit $c = t_{(i),k}$ in Sortierrichtung über den Basisraum Ω. Bei der Terminierung des Algorithmus ist der Basisraum Ω vollständig abgedeckt und damit statisch. Da die einzelnen Δ-Mengen aus disjunkten Regionen bestehen, die durch Teilung der Initialisierungsregion ρ_{init}, die den ganzen Basisraum Ω abdeckt, entstanden sind, ist $\Theta^S = \Theta_n^S \cup \Theta_n^D$ eine stabile Regionsmenge in der n Tupel enthalten sind.

Zu zeigen ist noch, dass jede Region genau einmal vom Arbeitsspeicher M auf den Sekundärspeicher geschrieben wird.
Nach Definition 7-3 und Definition 7-7 ist jede Region einer Delta-Menge bezüglich der Sortierdimension kleiner als die Sweep-Hyperebene $\Psi_{k,t}$ mit $t = t_{(i)}$. Die aktuelle Delta-Menge ist somit die Menge der Regionen, die noch in der dynamischen Regionszerlegung Θ^D enthalten sind, und durch das Einfügen des aktuellen Tupels $t_{(i)}$ stabil werden, also vollständig im stabilen Bereich B_{k,t_k}^S liegen

[delta-set] $\Delta = \{ \rho \mid \rho \in \Theta^D \text{ and } \rho \cap B_{k,t_k}^S = \rho \}$

Diese wird auf den Sekundärspeicher geschrieben:

[write]
```
if(Δ ≠ ∅)
{
        Θ^S = Θ^S ∪ Δ     // Schreibe die Seiten, die zur Δ-Menge
                          // gehören, auf den Sekundärspeicher

        Θ^D = Θ^D \ Δ     // Entferne die Seiten der Δ-Menge
                          //aus dem Arbeitsspeicher
}
```

q.e.d.

7.1.5 Grundlegende Leistungsmerkmale

Betrachten wir zunächst einmal die grundlegenden Leistungsmerkmale des TempTris-Algorithmus. Unter der Voraussetzung, dass der Cache ausreicht, wird jede Region des UB-Baums genau einmal geschrieben. Um eine mehrdimensionale Zerlegung, bestehend aus P Seiten, zu erzeugen, müssen dementsprechend P E/A-Operationen ausgeführt werden. Somit sind die E/A-Kosten linear abhängig von der Anzahl der Regionen und somit von der Größe der Eingabemenge. Die Korrektheit dieser Aussage ergibt sich direkt aus Satz 7-1. Hierbei ist zu beachten, dass die Anzahl der Seiten abhängig von dem jeweiligen Füllungsgrad der Seiten sind. Da hier der Splitalgorithmus des B-Baums verwendet wird, kann nur ein Füllungsgrad von 50% garantiert werden. In Kapitel 7.2 wird eine Optimierung vorgestellt, die den Füllungsgrad verbessert.

Die dynamische Regionsmenge, die vom TempTris-Algorithmus verwaltet wird, kann bis zu P Regionen enthalten, so dass der gesamte Datenstrom im Arbeitsspeicher gehalten werden muss, um die mehrdimensionale Partitionierung zu erzeugen. Dieser extreme Fall

entsteht genau dann, wenn die Datensätze bezüglich der Sortierdimension den gleichen Wert haben und somit alle in der gleichen Scheibe liegen. In diesem Fall kann sich also die *Trennebene* nicht weiter bewegen, damit bleiben alle Regionen dynamisch, bis das letzte Tupel in den Arbeitsspeicher eingefügt ist. Allgemein bestimmt die Kardinalität in der Sortierdimension die Anzahl der Scheiben. Viele Scheiben wiederum reduzieren den benötigten Arbeitsspeicher. Dies ist abhängig von der vorhandenen Datenverteilung.

Falls eine Scheibe S_i^{-t} nicht vollständig in den Arbeitsspeicher M passt, muss sie durch externes Sortieren (Sortieren durch Verschmelzen) behandelt werden. Die bereits geleistete Arbeit ist jedoch nicht verloren, da die stabilen Regionen sowie der Cache als Läufe wieder verwendet werden können.

Betrachten wir nun die Auswirkungen auf die Anfrage-Optimierung. Zur Erzeugung einer mehrdimensionalen Partitionierung kann das externe Sortieren durch den TempTris-Operator ersetzt werden. Kann der TempTris-Algorithmus die Vorsortierung nicht ausnutzen, kann er dynamisch auf das normale externe Sortieren umschalten, ohne zusätzliche E/A-Operationen zu verursachen.

Für gleichverteilte Daten werden vom TempTris-Algorithmus $\lceil P / | A_k | \rceil$ Seiten im Cache gehalten. $|A_k|$ bezeichnet hierbei die Kardinalität der Sortierdimension.

7.1.6 Allgemeine Optimierungsmöglichkeiten

Die Seitenauslastung der stabilen Regionen ist ein kritischer Punkt des TempTris-Algorithmus: Zum einen führt eine bessere Seitenauslastung zu einer geringeren Zahl von Seiten P und demzufolge zu weniger E/A-Kosten. Zum anderen zeigt ein kompakterer UB-Baum, d.h. ein UB-Baum mit besserer Seitenauslastung, ein besseres Leistungsverhalten bei Anfragen, da weniger Regionen durch eine Bereichsanfrage geschnitten werden.

Der im letzten Abschnitt beschriebene TempTris-Algorithmus teilt eine Seite, sobald ein Überlauf entsteht. Dies führt im schlimmsten Fall zu einer Seitenauslastung von mindestens 50%. Yao zeigt in [Yao78], dass die durchschnittliche Seitenauslastung bei einem B-Baum bei gleichverteilten Daten bei ca. $ln2 \approx 69\%$ liegt. Verwendet man einen verbesserten Seitenteilungs-Algorithmus, der die Nachbarn berücksichtigt [Küs83], erhält man eine durchschnittliche Seitenauslastung von 81%.

Eine weitere Möglichkeit stellt das Konzept der Z-Unterräume da. Hierbei stellt ein Z-Unterraum ein Z-Intervall dar, dessen Inhalt auf n Seiten gespeichert wird. Diese Z-Unterräume werden im Arbeitsspeicher gehalten. Somit hat ein Z-Unterraum keine bestimmte Kapazität, sondern die einzelnen Z-Unterräume werden so lange mit Daten gefüllt, bis der Cache gefüllt ist. Dies führt dann zu einer Seitenauslastung von nahezu 100%. Die 100%-ige Auslastung wird dann nicht erreicht, wenn sich die Anzahl der Tupel, die in dem Z-Unterraum liegen, nicht durch die Kapazität κ einer Seite teilen lässt. Da mehrere Seiten entstehen, kann zum Teil ein Seitenclusterung erzeugt werden.

Eine weitere Optimierungsmöglichkeit ist die Reduzierung der Δ-Mengenberechnung. Diese wird nur dann berechnet, wenn es zu einer Seitenteilung oder Bewegung der Trennebene kommt.

7.2 Der TempTris-Algorithmus für UB-Bäume

In diesem Abschnitt zeigen wir eine konkrete Ausprägung des TempTris-Algorithmus für die mehrdimensionale Datenstruktur UB-Baum [Bay96]. Der Autor möchte darauf hinweisen, dass das Konzept so allgemein ist, dass es auch für andere Datenstrukturen geeignet ist, speziell für andere raumfüllende Kurven (siehe Abschnitt 3.6.1).

7.2.1 Beispiel: Erzeugung einer Z-Regionszerlegung mit Hilfe des TempTris-Algorithmus

Betrachten wir zunächst die Voraussetzungen für den sinnvollen Einsatz des TempTris-Algorithmus zur Erzeugung einer Z-Regionszerlegung, um einen UB-Baum zu erstellen. Wie bereits bekannt, organisiert der UB-Baum seine Daten bezüglich der raumfüllenden Z-Kurve. Da wir die physikalische Organisation auf den Sekundärspeicher, die durch den Schlüssel des Zugriffspfads bestimmt wird, betrachten, muss das Sortierattribut der sortierten Folge ($\langle R\rangle$,<,k) Teil des UB-Baum-Schlüssels sein. Somit trägt das Sortierarttribut zum Schlüssel bei.

Wäre das Sortierattribut nicht Teil des UB-Baum-Schlüssels, könnte eine Trennung zwischen dynamischem und stabilem Bereich nicht vorgenommen werden und man müsste das externe Sortieren einsetzen.

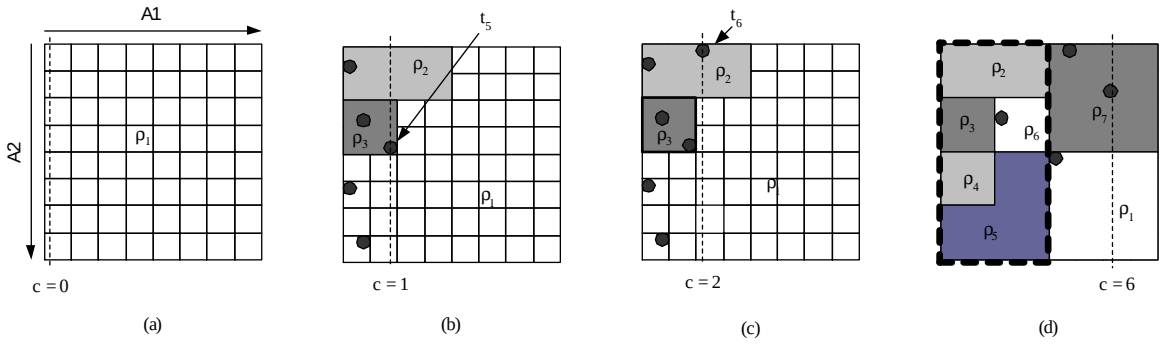


Abbildung 7-3: TempTris-Algorithmus und UB-Baum

Das folgende Beispiel zeigt den TempTris-Algorithmus bei der Erzeugung einer Z-Regionszerlegung Θ^S. Abbildung 7-3 zeigt einige Schritte des TempTris-Algorithmus für den 2-dimensionalen Basisraum Ω mit A_1 als horizontaler Dimension, A_2 als vertikaler Dimension und dem Ursprung (0,0) in der linken oberen Ecke und einer Auflösung von 8x8 Punkten. Die Kapazität einer Seite wird auf 2 festgelegt.

Zu Beginn besteht die *dynamische Regionsmenge* genau aus einer Region $\Theta^D = \{\Omega\} = \{\rho_1\}$. Die stabile Regionsmenge Θ^S ist leer. Wir bezeichnen die Position der Trennebene mit *c*. In Abbildung 7-3.a hat der TempTris-Algorithmus noch nicht begonnen, so dass nur die Region $\Omega \in \Theta^D$ von der Trennebene an Position $c = 0$ geschnitten wird. Abbildung 7-3.b zeigt drei dynamische Regionen, die bereits vom TempTris-Algorithmus erzeugt worden sind. Sie enthalten 5 Tupel von der sortierten Eingabefolge, d.h. $\Theta^D = \{ \rho_1, \rho_2, \rho_3\}$. $t_5 = (1,3)$ ist das zuletzt eingefügte Tupel, so dass die aktuelle Position der Trennebene $c = 1$ ist. In Abbildung 7-3.c, wird das Tupel $t_6 = (2,0)$ in den Basisraum Ω eingefügt, die Trennebene bewegt sich weiter nach c = 2. Durch die Bewegung der Trennebene nach Position $c = 2$ wird die dynamische Region ρ_3 vollständig vom *stabilen Bereich* B^S

überdeckt. Dies hat zur Folge, dass in diese Region keine weiteren Daten (Tupel) eingefügt werden, da nach Satz 4-1 alle Tupel, die einen kleineren Wert als die aktuelle Trennebene $\Psi_{k,t(k)}$ haben, eine kleinere Position in der sortierten Eingabefolge ($\langle R\rangle$, $\leq$, A_1) besitzen und somit bereits eingefügt worden sind. Somit wird die Region ρ_3 stabil, d.h. die Seite, die die Region repräsentiert, kann aus dem Cache entfernt und auf den Sekundärspeicher geschrieben werden. Die *stabile Regionsmenge* besteht nun aus $\Theta^S = \{\rho_3\}$, die *dynamische Regionsmenge* aus $\Theta^D = \{\rho_1, \rho_2\}$. In Abbildung 7-3.d besteht die *stabile Regionsmenge* bereits aus 5 Regionen, $\Theta^S = \{\rho_2, \rho_3, \rho_4, \rho_5, \rho_6\}$. Zu dem Zeitpunkt, da das letzte Tupel der *sortierten Eingabefolge* ($\langle R\rangle$, $\leq$,A_1) verarbeitet worden ist, werden alle Regionen $\Theta^D = \{\rho_1, \rho_7\}$, die noch im Cache sind, zu stabilen Regionen übergeführt und auf den Sekundärspeicher geschrieben. Der Algorithmus terminiert.

7.2.2 Verwaltung der dynamischen Regionsmenge

Die Verwaltung der *dynamischen Regionsmenge* Θ^D ist das zentrale Problem des TempTris-Algorithmus und soll an dieser Stelle näher betrachtet werden. Hierbei werden die einzelnen Regionen im Cache gehalten. Die Verwaltung des noch nicht gelesenen Raumes Φ im Tetris-Algorithmus (siehe Abschnitt 5.9) kann zum Teil zur Lösung dieses Problems adaptiert werden. In diesem Abschnitt wird zunächst die Anforderung, die sich aus dem Problem der Cache-Verwaltung ergibt, betrachtet. Danach wird detailliert auf die einzelnen Komponenten eingegangen.

7.2.2.1 Anforderung

Gesucht ist eine Organisation des Caches, die folgende Operationen auf der dynamischen Regionsmenge Θ^D effizient unterstützt:

1. Finde die Region ρ der dynamischen Regionsmenge Θ^D, die das aktuelle Tupel *t* aufnimmt.
2. Bestimme alle Regionen aus dem Cache, die bezüglich der Sortierdimension *k* kleiner als die aktuelle Sweep-Hyperebene $\Psi_{k,c}$ sind.

7.2.2.2 Regionsbestimmung

Die Anforderung 1 kann formal wie folgt beschrieben werden:

$$\{\rho \mid \rho \in \Theta^D \wedge t \in \rho\} \qquad \text{Gl: 7-1}$$

Die Menge besteht genau aus einer Region ρ, da $\Theta^D \cup \Theta^S = \Theta$ gilt und nach Definition 3-30 eine paarweise disjunkte Zerlegung bezüglich der Z-Kurve des Basisraums Ω ist. Der Basisraum wird vollständig in Z-Intervalle aufgeteilt. Da die *Z-Funktion* eine bijektive Abbildung von $\Omega \rightarrow S$ mit $S \subset \mathbb{N}$ ist, und somit eine totale Ordnung auf Ω (die Z-Ordnung, siehe Definition 3-22) erzeugt, kann nur eine Region ρ die Bedingungen

$$\alpha \leq Z(t) \leq \beta \quad \text{mit } \rho = [\alpha :_Z \beta] \qquad \text{Gl: 7-2}$$

erfüllen. Die Daten werden also bezüglich der Z-Ordnung in den Cache eingefügt. Für die Implementierung des Caches wird folgende C-Struktur verwendet:

```
typedef struct s_region
{

   AD lowerBound; /* lower bound of corresponding z-curve interval */
   AD upperBound; /* upper bound of corresponding z-curve interval */
   AD maxTupInSortDim;   /* tetris address of region's greatest tuple
               according to the sorting dimension */
   STAT_RAM_PG *page;
}
region;
```

Abbildung 7-4: C-Struktur für Region im Cache

Vergleicht man die C-Struktur mit der Implementierung der Z-Region auf dem Sekundärspeicher, fällt auf, dass nicht nur die obere Grenze *upperBound* gespeichert wird, sondern auch die untere Grenze *lowerBound*. Der Grund dafür liegt in der Berechnung der maximalen Tetris-Adresse einer Region. Dies wird für die Bestimmung der Delta-Menge benötigt, deren Beschreibung Bestandteil des nächsten Abschnittes ist und hier nicht weiter behandelt wird.

Aus Forderung 1 aus Abschnitt 7.2.2.1 muss die Datenstruktur folgende Operationen effizient unterstützen:

1. Finde die Region ρ, in die das Tupel t mit Z(t) fällt.
2. Füge das Tupel in die Region ρ ein.
3. Falls ein Überlauf entsteht, teile die Region in zwei Regionen und füge die neue Region entsprechend der Z-Ordnung in die Datenstruktur ein.

Da durch das Einfügen der Tupel eine physikalische Clusterung bezüglich der Z-Kurve erhalten bleiben soll, ist die Indizierung mit Hilfe eines Hash-Verfahrens ungeeignet. Durch den Einsatz der Transformationsfunktion bei einem Hash-Verfahren kann die totale Ordnung der Z-Kurve nicht erhalten bleiben (siehe 3.5).

Eine baumstrukturierte Speicherorganisation ist für diese Aufgabe besonders geeignet, da zum einem ein wahlfreier Zugriff möglich ist und die physikalische Organisation (Clusterung) erhalten bleibt. Wird als Datenstruktur ein balancierter Baum eingesetzt, können die Grundoperationen *Suchen*, *Löschen*, *Einfügen* und *Aktualisieren* in O($\log_2 n$) Operationen durchgeführt werden. Für die Prototyp-Implementierung wird ein AVL-Baum [AdeL62], ein balancierter binärer Baum, eingesetzt. Diese Methode benötigt nur zwei zusätzliche Bits pro Knoten. Die Komplexität für folgende Operationen beträgt O(log n):

1. Finden des Elements mit dem Schlüssel k
2. Einfügen eines Elements bezüglich des Schlüssels k
3. Löschen eines Elements aus dem Index

Eine genaue Kostenanalyse ist in [Knu98v3] und [OttW96] zu finden. Daneben bietet der AVL-Baum einen sequenziellen Zugriff in Sortierreihenfolge bezüglich des Schlüssels in linearer Zeit [Knu98v1]. Alternativ können auch andere balancierte Datenstrukturen, wie z.B. der 2-3 Baum [AhoHU74] oder dessen binäre Repräsentation [Bay71], eingesetzt werden.

Da aus der sortierten Folge bezüglich des Sortierattributs *k* eine Z-Regionszerlegung Θ erzeugt werden soll, wird als Schlüssel *K* die Z-Adresse der oberen Grenze[15] (*upperbound*) benutzt. Wir bezeichnen diesen Index als Z-Index.

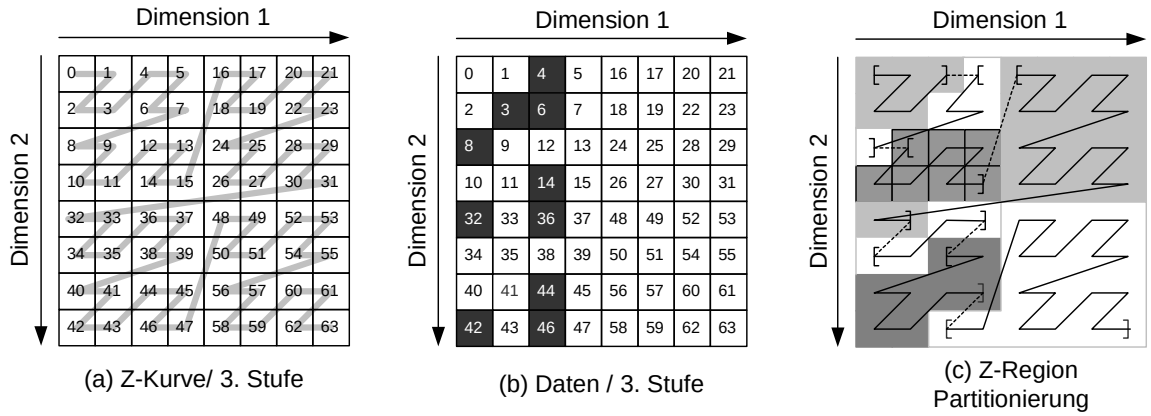
Beispiel:
Gegeben sei eine *sortiere Folge* (⟨*R*⟩,≤,*k*) von 2-dimensionalen Daten. Sie ist nach Dimension 1 total geordnet:

$$(\langle(0,4), (0,2), (0,7), (1,1), (2,4), (2,3), (2,0), (2,1),(2,7), (2,6), ...\rangle, \leq,1) \qquad \texttt{Gl: 7-3}$$

Beide Dimensionen gehen in die Z-Adresse ein. Demzufolge erhält man für die Daten folgende Z-Adressen:

$$(\langle 32_z, 8_z, 42_z, 3_z, 36_z, 14_z, 4_z, 6_z,46_z, 44_z, ...\rangle, \leq,1) \qquad \texttt{Gl: 7-4}$$

Das aktuelle Tupel sei t = (2,6) mit der Adresse 44_z. Die entsprechende Datenverteilung und die dynamische Regionszerlegung Θ^D ist in Abbildung 7-5 dargestellt.



Region	Z-Intervall	Daten
1	[0:5]	{3,4}
2	[6:8]	{6,8}
3	[9:14]	{14}
4	[15:33]	{32}
5	[34:37]	{36}
6	[38:45]	{42,44}
7	[44:63]	{46}

Abbildung 7-5: Dynamische Regionszerlegung

[15] Ohne Beschränkung der Allgemeinheit geben wir hier nur den Algorithmus für aufsteigend sortierte Sequenzen an.

Da keine Region die Bedingung

$$\left(\rho \cap B^{S}_{k,t_k}\right) = \rho \qquad \text{Gl: 7-5}$$

erfüllt, gehören alle Regionen zur dynamischen Regionszerlegung. Um einen wahlfreien Zugriff zu gewährleisten, wird jede einzelne Region der dynamischen Regionszerlegung Θ^D im AVL-Baum (Z-Index) verwaltet. Für die obige sortierte Folge ($\langle R\rangle$,≤,1) ergibt sich folgende Z-Index Struktur:

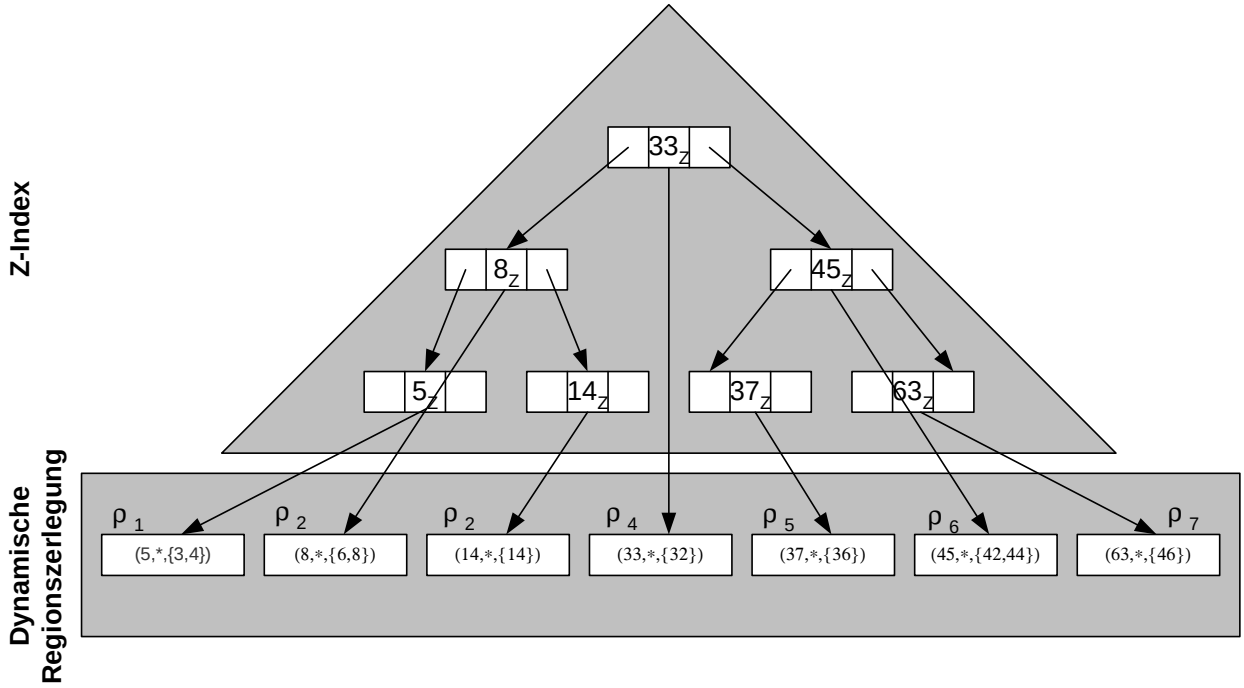


Abbildung 7-6: Z-Index-Struktur

Um für das aktuelle Tupel *t* die entsprechende Z-Region zu finden, muss die entsprechende Z-Adresse $Z(t) = \chi$ berechnet werden und im AVL-Baum mit der Bedingung

$$\alpha \le x \le \beta \quad \text{mit } \rho = [\alpha:_Z\beta] \qquad \text{Gl: 7-6}$$

gesucht werden.

Hierzu muss der normale Such-Algorithmus auf Binär-Bäumen [OttW96] leicht modifiziert werden. Der Grund liegt darin, dass es sich hier nicht um eine genaue Punktanfrage handelt, sondern um ein Intervall. Dazu wird die Vergleichsfunktion wie folgt auf der Region definiert:

$\chi < \rho \Rightarrow \chi < \alpha$ mit $\rho = [\alpha:_Z\beta]$
$\chi \in \rho \Rightarrow \alpha \le \chi \le \beta$ mit $\rho = [\alpha:_Z\beta]$

$$\chi > \rho \Rightarrow \chi > \beta \qquad \text{mit } \rho = [\alpha :_Z \beta]$$

Somit kann die entsprechende Region in O(log *n*) Vergleichen bestimmt werden. *n* bezeichnet die Anzahl der Regionen.

7.2.2.3 Delta-Menge

Die Anforderung 2 (siehe 7.2.2.1) beschäftigt sich mit der Bestimmung der Δ-Menge. Formal ist die Delta-Menge in Kapitel 7.1.2 in Definition 7-7 beschrieben worden. Es ist genau die Differenzregionsmenge, in der sich zwei *stabile Regionsmengen* Θ^S unterscheiden.

7.2.2.3.1 Problembeschreibung und Anforderungen

Die Cache-Verwaltung des TempTris-Algorithmus hält alle Z-Regionen im Arbeitsspeicher, deren kartesische Ausdehnung vom dynamischen Bereich B^D geschnitten werden. Allgemein kann eine Region ρ in drei Kategorien eingeteilt werden.

1. ρ liegt vollständig im statischen Bereich B^S
2. ρ liegt vollständig im dynamischen Bereich B^D
3. ρ schneidet sowohl im statischen als auch im dynamischen Bereich

Hierzu muss jedoch die geometrische Ausdehnung einer Z-Region berücksichtig werden. Die wesentliche Eigenschaft einer Z-Region ist, dass sie aus ein oder maximal zwei nicht zusammenhängenden Bereichen besteht [FriM97], [Mar99]. Deshalb können dynamische Regionen existieren, die nicht die Trennebene Ψ (siehe Definition 4-2) schneiden und dennoch zu beiden Bereichen B^S und B^D gehören.

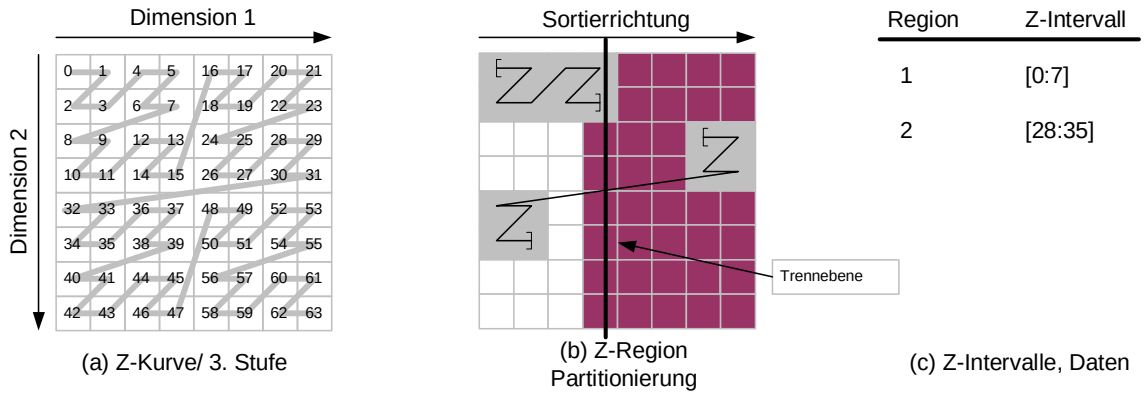


Abbildung 7-7: Schnitt der Z-Region mit dem dynamischen Bereich

Abbildung 7-7.b zeigt einen 2-dimensionalen Basisraum Ω. Die Daten werden sortiert bezüglich Dimension 1 eingefügt. Die Trennebene steht an Position 3 und unterteilt somit Ω in den statischen Bereich B^S (heller Bereich) und den dynamischen Bereich B^D (dunkler Bereich). Region 1 (siehe Abbildung 7-7.c und Abbildung 7-7.b) besteht aus einem einzigen zusammenhängenden Bereich. Da Region 1 sowohl den dynamischen als auch

den statischen Bereich schneidet, wird auch die Trennebene geschnitten. Region 2 ist jedoch eine Sprungregion. Obwohl sie beide Bereiche schneidet, wird die Trennebene nicht von ihr geschnitten. Somit ist der Schnitt mit der Sweepline kein hinreichendes Kriterium, um festzustellen, ob eine Region dynamisch oder statisch ist. Dies führt wieder zum Konzept der Tetris-Ordnung und der maximalen T-Adresse einer Z-Region.

7.2.2.3.2 Tetris-Ordnung

In Kapitel 5.4 wurde die Tetris-Ordnung formal eingeführt (siehe Definition 5-3). Sie erzeugt auf dem n-dimensionalen Basisraum Ω eine totale Ordnung und ist in linearer Zeit berechenbar.

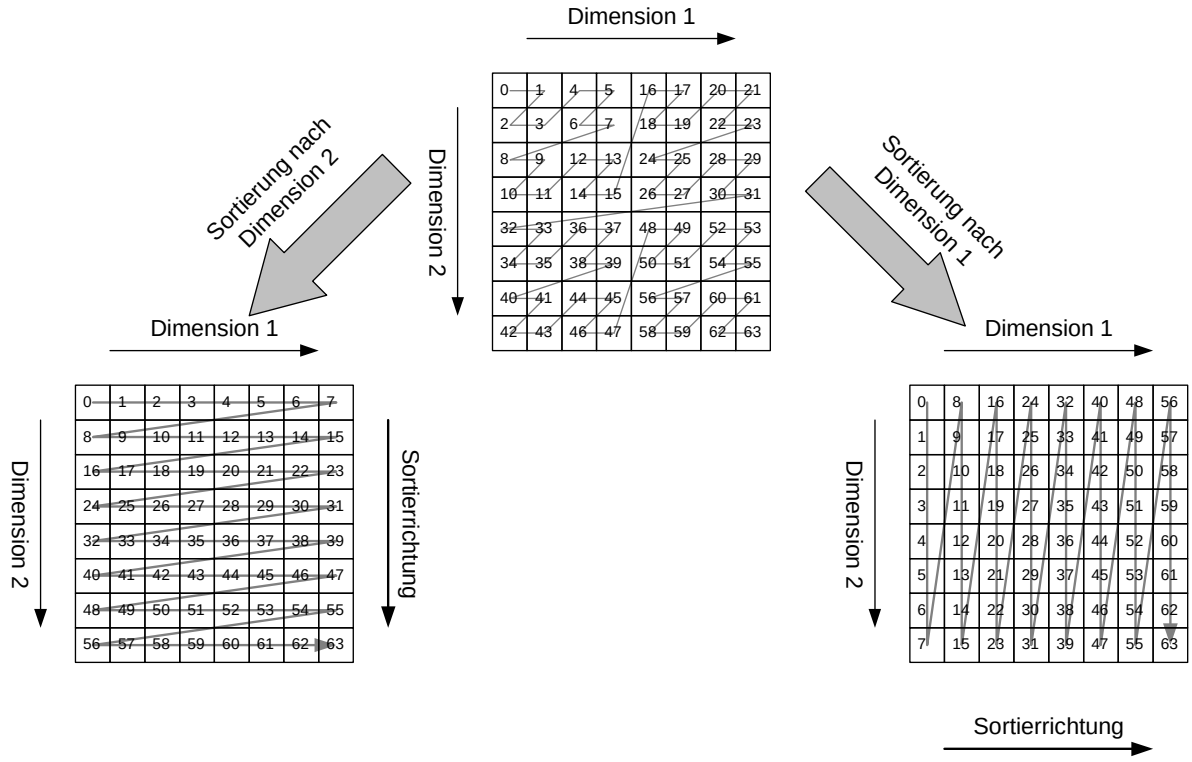


Abbildung 7-8

Abbildung 7-8 zeigt die T-Ordnung für den 2-dimensionalen Fall. Allgemein gibt es *n* T-Ordnungen für einen n-dimensionalen Basisraum Ω.

Durch die Einführung der T-Ordnung wird das n-dimensionale Schnittproblem mit der Sweep-Hyperebene (siehe Definition 4-2) auf ein 1-dimensionales Problem reduziert, d.h. die minimale T-Adresse der Sweep-Hyperebene trennt den dynamischen Bereich B^B vom statischen Bereich B^S . Der Grund liegt darin, dass jede Sweep-Hyperebene

durch ein T-Intervall $[\ \alpha_T, \beta_T\ ]_T$ eindeutig abgebildet werden kann, da die Struktur der binären Darstellung der T-Adresse eine Konkatenierung der Sortierdimension *k* mit der reduzierten Z-Adresse $Z(x|k)$ ist.

$$x_k \circ Z(x \mid k) = \underbrace{x_{k,s-1} \ldots x_{k,0}}_{\text{Sortier-Dimension}} \cdot \underbrace{x_{d,s-1} \ldots x_{k+1,s-1} x_{k-1,s-1} \ldots x_{1,s-1}}_{Z(x|k)} \cdot \ldots \cdot \underbrace{x_{d,0} \ldots x_{k+1,0} x_{k-1,0} \ldots x_{1,0}}_{Z(x|k)}$$

$$= \underbrace{x_{k,s-1} \ldots x_{k,0}}_{\text{T-Stufe0}} \cdot \underbrace{x_{d,s-1} \ldots x_{k+1,s-1} x_{k-1,s-1} \ldots x_{1,s-1}}_{\text{T-Stufe 1}} \cdot \ldots \cdot \underbrace{x_{d,0} \ldots x_{k+1,0} x_{k-1,0} \ldots x_{1,0}}_{\text{T-Stufe } s}$$

Gl: 7-7

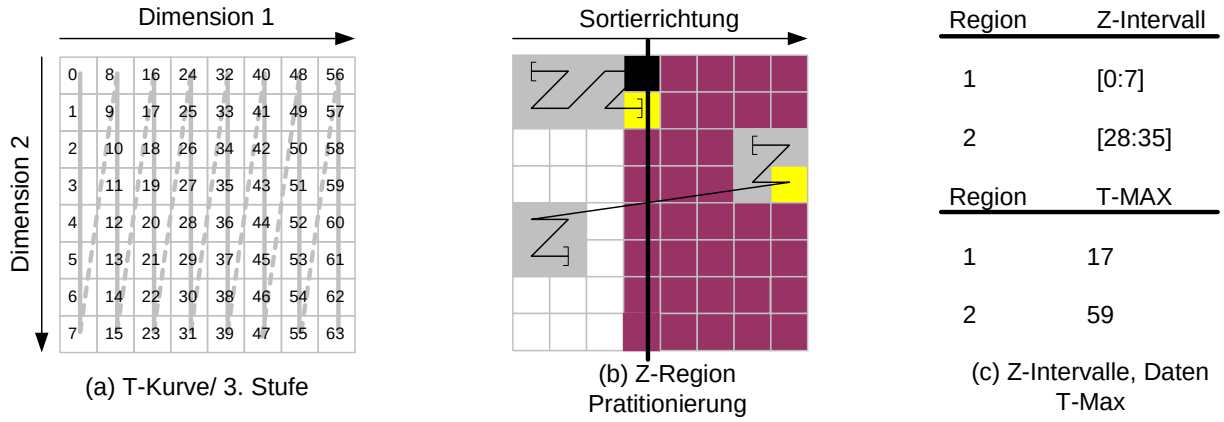
Hierbei ergibt sich α_T und β_T, indem man alle Bits der T-Stufe 1 bis T-Stufe *s* auf 0 bzw. 1 setzt. Formal kann das wie folgt beschrieben werden:

$$\alpha_T = \sum_{j=0}^{s-1} c_j 2^{j+s(d-1)}$$

Gl: 7-8

$$\beta_T = \underbrace{\left(\sum_{j=0}^{s-1} c_j 2^{j+s\cdot(d-1)} \right)}_{\text{Sortierdimension}} + \underbrace{\sum_{j=0}^{s-1} \sum_{i=k+1}^{d} 2^{j\cdot(d-1)+i-2} + \sum_{j=0}^{s-1} \sum_{i=1}^{k-1} 2^{j(d-1)+i-1}}_{Z(x|k)}$$

Gl: 7-9



(a) T-Kurve/ 3. Stufe

(b) Z-Region Pratitionierung

(c) Z-Intervalle, Daten T-Max

Abbildung 7-9

Abbildung 7-9.a zeigt die T-Ordnung auf der 3-ten Stufe für den 2-dimensionalen Fall. Abbildung 7-9.b zeigt die Aufteilung des Basisraumes Ω in den statischen und dynamischen Bereich. Der Trennpunkt zwischen dem dynamischen und statischen Bereich ist somit die minimale T-Adresse, die durch das schwarze Viereck gekennzeichnet ist. Der Bereich hat die T-Adresse 24. Somit werden der statische und der dynamische Bereich durch folgende T-Intervalle bestimmt:

$B^S = [0,23]_T$

$B^D = [24,63]_T$

Die Sweep-Hyperebene wird durch die T-Adresse 24 bestimmt. Wir bezeichnen die Adresse als *T-Sweep-Adresse* ψ_T.

Um nun herauszufinden, ob eine Region noch den dynamischen Bereich schneidet, muss für jede Region, die im Arbeitsspeicher liegt, die maximale T-Adresse τ bestimmt werden. Ist der Wert der maximalen T-Adresse τ einer Region ρ größer als die T-Sweep-Adresse, schneidet sie den dynamischen Bereich und kann deshalb noch nicht aus dem Arbeitsspeicher entfernt werden. Alle Regionen, deren maximale T-Adresse kleiner sind als die T-Sweep-Adresse, liegen vollständig im statischen Bereich. Somit kann die Delta-Menge formal wie folgt bestimmt werden:

$$\Delta = \{\rho | \tau < \psi \wedge \tau \in \rho\} \qquad \text{Gl: 7-10}$$

7.2.2.4 Organisation des Caches

Die dynamischen Regionen werden bezüglich der Z-Ordnung und der Tetris-Ordnung organisiert. Hierzu wird die Struktur in Abbildung 7-6 wie folgt erweitert:

Für jede Region wird die maximale T-Adresse τ berechnet. Die T-Adresse wird als Schlüssel in einen T-Index eingeführt. Hierzu wird wie beim Z-Index ein AVL-Baum eingesetzt.

Für die in Abbildung 7-5 dargestellte Regionszerlegung ergibt sich somit folgender T-Index:

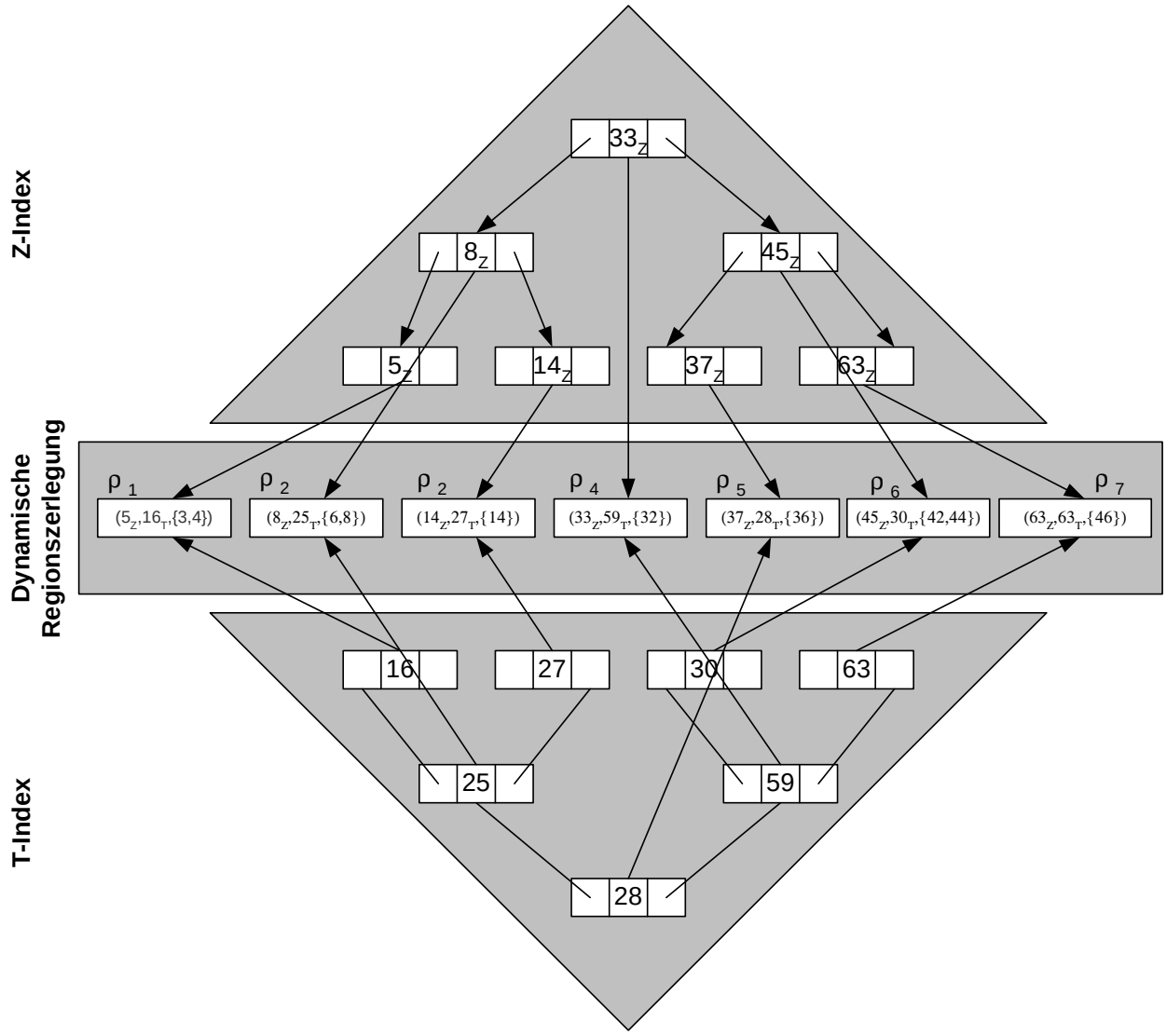


Abbildung 7-10: Cache-Verwaltung

Um nun die aktuelle Δ-Delta-Menge

$$\Delta = \{\rho | \tau < \psi \wedge \tau \in \rho\}$$ Gl: 7-11

zu berechnen, muss eine Bereichsanfrage auf dem T-Index ausgeführt werden. Hierzu wird der Such-Algorithmus auf Binär-Bäumen von Punktanfragen auf Bereichsanfragen mit offenem Intervall $]\infty, \psi]$ modifiziert. Somit kann die Δ-Menge in O(n ln n) Vergleichen bestimmt werden. n bezeichnet die Anzahl der Regionen.

7.2.2.5 Optimierungsmöglichkeiten

Der oben beschriebene Algorithmus entspricht dem allgemeinen Konzept des sortierten Schreibens unter Ausnutzung der Vorsortiertheit in einer Dimension. Es wurden noch keine Optimierungen vorgenommen. Es werden nun zwei wesentliche Optimierungsmöglichkeiten vorgestellt, deren Ziel es ist, die Seitenauslastung zu verbessern.

Die Seitenauslastung oder der Füllungsgrad $f_{Füllungsgrad}$ einer Seite ist das Verhältnis der Tupel einer Seite zu deren Kapazität κ:

$$f_{\text{Füllungsgrad}} = \frac{|p|}{\kappa} \qquad \text{Gl: 7-12}$$

Die durchschnittliche Seitenauslastung oder der durchschnittliche Füllungsgrad $F_{füllungsgrad}$ (Mittelwert) für eine Tabelle oder Relation R ist das arithmetische Mittel aller Seiten P_R der Relation R:

$$F_{\text{Füllungsgrad}} = \frac{1}{P}\left(\sum_{i=1}^{P}\frac{|p_i|}{\kappa}\right) = \frac{|P|}{P\cdot\kappa} \qquad \text{Gl: 7-13}$$

$|P| = \sum_i^P |p_i|$ ist und somit die Anzahl der Tupel, die auf der Seitenmenge P gespeichert sind, d.h. $|P| = |R|$.

7.2.2.5.1 Statische Speicherseiten

Die Seitenauslastung der dauerhaften Regionsmenge, die durch den TempTris-Algorithmus erzeugt wird, ist ein kritischer Leistungsfaktor. Zum einen benötigt man weniger E/A-Operationen, um die multidimensionale Zerlegung als UB-Baum auf den Sekundärspeicher abzulegen, zum anderen reduziert sich die Anzahl der Seitenzugriffe bei Bereichsanfragen, da die Tupel der Relation auf weniger Sekundärspeicherseiten verteilt sind. Dies führt dementsprechend zu einer besseren Antwortzeit des Systems.

Beispiel 7-1:
Ein UB-Baum hat eine garantierte Seitenauslastung von 50%, da die Datenstruktur auf einem B-Baum basiert. Kann die Seitenauslastung auf 100% gesteigert werden, reduziert sich der Platzbedarf für die Tabelle um 50%. Die Kosten für eine Bereichsanfrage sind nach dem Kostenmodell (siehe Kapitel 4.4.1.2) [Mar99] proportional zur Anzahl der geladenen Regionen. Somit ist eine Leistungssteigerung von 100% erreichbar.

Identische Cacheseitengöße und Sekundärspeichergröße

Der unoptimierte TempTris-Algorithmus verwendet identische Cache- und Sekundärseitengrößen, so dass eine Region ρ genau auf eine Cacheseite p_c , mit $\rho \leftrightarrow p_c$, abgebildet wird. Die Cacheseite p_c wird dann in genau einer Sekundärseite abgelegt. Somit ergibt sich eine eindeutige Abbildung der Region auf eine Sekundärspeicherseite.

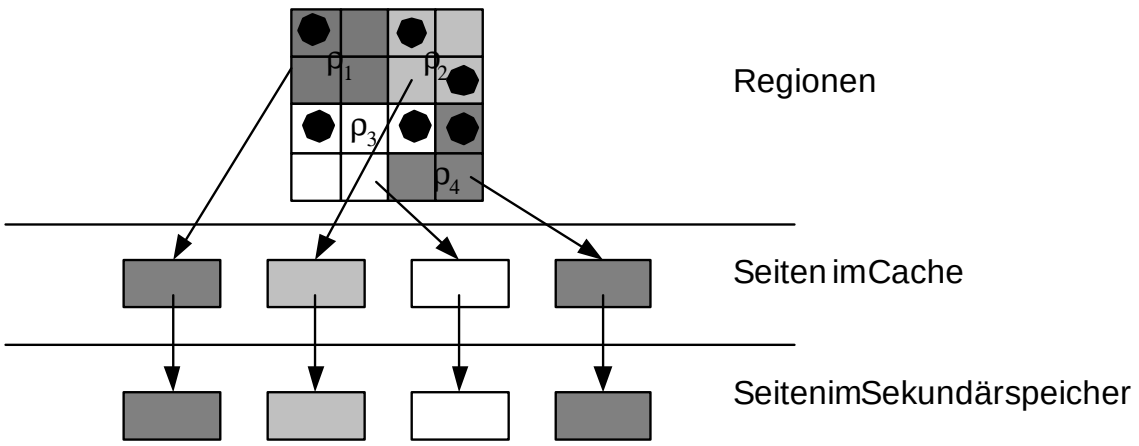


Abbildung 7-11: Regions-, Seiten-Abbildungshierarchie

Dieser Zusammenhang ist in Abbildung 7-11 noch einmal dargestellt. Der TempTris-Algorithmus verwendet den einfachen Splitalgorithmus des B-Baums [BayM72], der eine Seite in der Mitte teilt, falls ein Überlauf in der betreffenden Seite entsteht. Somit liegt der Füllgrad $f_{Füllungsgrad}$ einer Seite p eines B-Baums der Ordnung m (siehe Abschnitt) im Bereich

$$\frac{m}{\kappa} \le f_{Füllungsgrad}(p) \le \frac{2m}{\kappa} \quad , \quad \kappa = 2m$$
$$\Leftrightarrow \quad 50\% \le f_{Füllungsgrad}(p) \le 100\% \qquad \text{Gl: 7-14}$$

Dies kann also im schlechtesten Fall zu einer Seitenauslastung von 50% führen. Dieser Fall tritt genau dann auf, wenn in einem B-Baum oder UB-Baum die Daten sortiert nach dem Schlüssel eingefügt werden, für den UB-Baum also nach der Z-Ordnung. Dies entspricht dem Massenladen einer Relation. Beim *Massenlade-Algorithmus* wird eine einfache Modifikation des Spaltungsalgorithmus, die darin besteht, die aktuelle Seite bei einem Überlauf zu 100% gefüllt zu lassen und eine neue leere Seite zu erzeugen, vorgenommen. Hierbei muss jedoch ein Sonderfall behandelt werden. Falls die letzte Seite nicht zu 50% gefüllt ist, müssen die Daten mit der Vorgängerseite neu aufgeteilt werden. Somit entstehen maximal zwei Seiten mit einem Füllungsgrad zwischen 50% und 100%. Sind die Daten nicht statisch, d.h. sollen zu einem späteren Zeitpunkt noch Einfügungen (Inserts) vorgenommen werden, ist es sinnvoll, die Seitenkapazität κ nicht voll auszunutzen. Dazu wird einfach eine neue Seite erzeugt, wenn der gewünschte Füllungsgrad f erreicht ist. Somit wird die physikalische Clusterung bei einem späterem Einfügen nicht so schnell zerstört. Eine Adaption dieses Algorithmus für das Massenladen für UB-Bäume ist in [FenKM00] zu finden. Allgemein ergeben sich für n Tupel

$$P = \left\lceil \frac{1}{F_{Füllungsgrad} \cdot \kappa} n \right\rceil \qquad \text{Gl: 7-15}$$

Seiten. Hierbei handelt es sich also um eine lineare Funktion mit der Steigung $1/(F_{Füllungsgrad} \cdot \kappa)$. Wird der UB-Baum mit dem oben beschriebenen Massenlade-Algorithmus erzeugt, kann $F_{Füllungsgrad}$ durch $f_{Füllungsgrad}$ ersetzt werden und es gilt:

$$P = \left\lceil \frac{1}{f_{Füllungsgrad} \cdot \kappa} n \right\rceil \qquad \text{Gl: 7-16}$$

Yao zeigt in [Yao78], dass die durchschnittliche Seitenauslastung $f_{Füllungsgrad}$ bei einem B-Baum bei gleichverteilten Daten, der durch wahlfreies Einfügen (random insert) erzeugt wird, bei ca. $ln2 \approx 69$ Prozent liegt. Betrachten wir nun das Verhalten des UB-Baums bei der Erzeugung mit Hilfe des TempTris-Algorithmus.

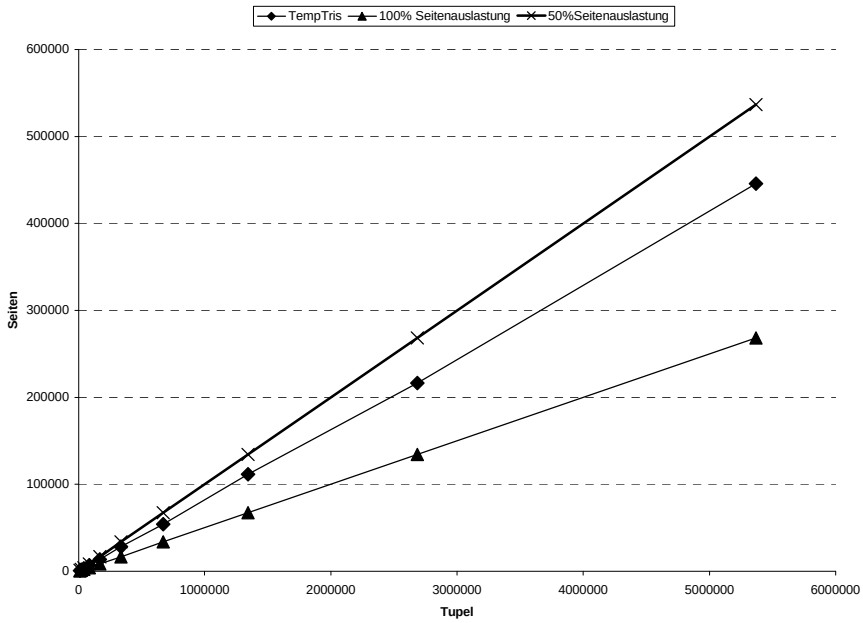


Abbildung 7-12: Anzahl der Seiten, die vom TempTris-Algorithmus erzeugt werden

Messung 7-1:

Auf System *S*1 wurden die Datenbanken *UB1db2d01* bis *UB11db2d11* erzeugt. Hierbei handelt es sich um 2-dimensionale Datenbanken mit einer Tupelgröße $x_{\text{größe}}$ = 100 Byte, wobei der Schlüssel aus zwei Integer-Werten und die Daten aus einer binären Zeichenkette (String) bestehen. Die Daten sind bezüglich des 2-dimensionalen Raums gleichverteilt. Die Größe $p_{größ}$ der Datenbankseite beträgt 2KB, so dass mindestens 10 Tupel und maximal 20 Tupel pro Seite gespeichert werden können. Die Daten wurden sortiert nach Attribut A_1 in den Cache eingefügt.

Messung 7-1 (Abbildung 7-12) zeigt bei einer gegebenen Tupelzahl die erzeugten Seiten. Die vom TempTris-Algorithmus erzeugte Seitenmenge unter Verwendung des einfachen Seitenspaltungsalgorithmus steigt linear und bestätigt somit die Gleichung Gl: 7-15. Der

untere Graph zeigt den optimalen Fall mit einem durchschnittlichen Füllungsgrad *F* von 100%, der obere den schlechtesten mit 50%. Wie zu sehen ist, liegt der TempTris-Algorithmus dazwischen und liegt, wie zu erwarten, näher bei 50% als bei 100%.

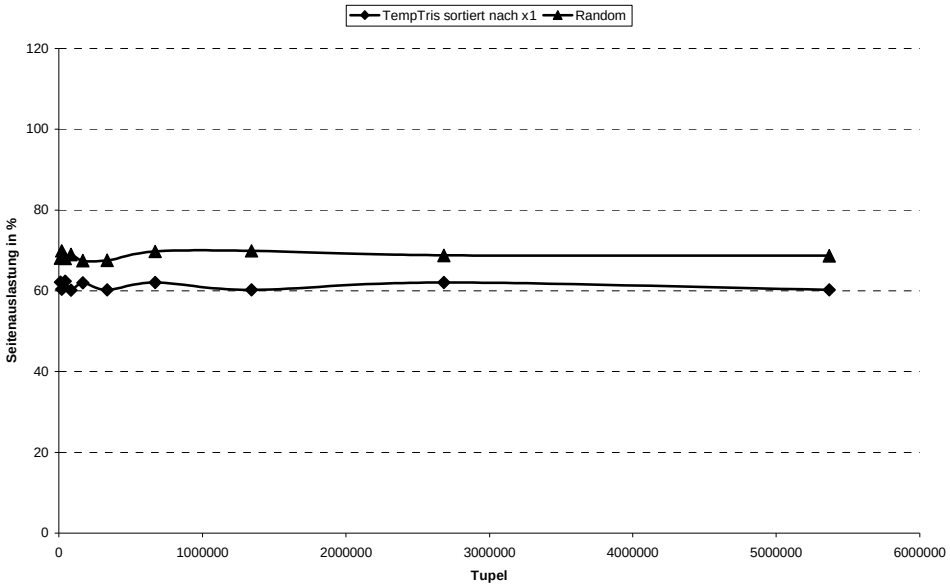


Abbildung 7-13: Seitenauslastung der DB mit TempTris sortiert nach Attribut x1 und freies Einfügen (random insert).

Messung 7-2:

Auf System S1 werden die Datenbanken *UB1db2d01* bis *UB1db2d11* durch wahlfreies Einfügen (random insert) sowie mit dem TempTris-Algorithmus sortiert nach Dimension A_1 erzeugt.

Messung 7-2 (Abbildung 7-13) zeigt den durchschnittlichen Füllungsgrad *F* mit steigender Anzahl der Tupel. Er liegt bei ca. 61,2% und liegt somit unter dem theoretischen Wert von 69%, da nur die Dimension A_2 gleichverteilt ist, jedoch A_1 sortiert. Der durchschnittliche Füllungsgrad *F* für wahlfreies Einfügen liegt bei ca. 68%. Dies entspricht der Theorie.

Diese Seitenauslastung stellt somit kein zufriedenstellendes Ergebnis dar. In [Bae89, Küs83] werden verbesserte Teilungsalgorithmen (Split-Alogrithmen) vorgestellt die eine Seitenauslastung von ca. $2ln(3/2) \approx 81\%$ erzeugen. Sie basieren auf dem Prinzip „*Teile mit dem Nachbarn*“ (sharing). Hierbei werden Daten mit dem benachbarten Knoten (Geschwistern) geteilt. Erst wenn auch der Nachbar zu 100% ausgelastet ist, wird der Knoten geteilt. Die Daten werden gleichmäßig auf die drei Knoten verteilt, so dass ein Füllungsgrad zu 2/3 garantiert werden kann. Da der TempTris-Algorithmus die Seitenteilung im Cache vornimmt, kann dieses Konzept verallgemeinert werden, ohne dass zusätzliche E/A-Operation entstehen. Hierzu wird das Modell um die Begriffe des Unterraums (Subspaces) und Z-Unterraum erweitert (sieh Definition 5-1). Ein Z-Unterraum ist ein beliebiges Z-Intervall im Basisraum Ω. Zu beachten ist hier, dass die Kapazität noch nicht festgelegt ist, d.h. die Anzahl der Datensätze, die in den Z-Unterraum fallen, ist unspezifiziert. Nach Definition 3-23 ist eine Z-Region ρ ein Teilraum, der vollständig durch ein Z-Intervall abgedeckt wird und dessen Elemente (Tupel) genau auf eine Seite *p* des Sekundärspeichers gespeichert werden. Eine Verallgemeinerung dieses Konzepts wird als *Z_u-Region* bezeichnet. Hierbei wird einer Z_u-Region eine

Arbeitsspeicherseite *v* zugeordnet, deren Kapazität $u \cdot \kappa$ beträgt. κ ist die Kapazität einer Datenbankseite *p*.

Definition 7-9: Z_u-Region

Eine *Z_u-Region* $\rho^u = [\alpha:_u\beta]$ ist ein Z-Unterraum $\sigma = [\alpha:\beta]$, der genau einer Arbeitsspeicherseite *v* zugeordnet ist und deren Kapazität $u \cdot \kappa$ beträgt.

Abbildung 7-14 zeigt ein Beispiel für eine Z_2-Regionszerlegung des Basisraums Ω. Eine Z_2-Region hat somit eine Kapazität von zwei Seiten *p* auf dem Sekundärspeicher. In diesem Beispiel hat eine Seite *p* die Kapazität $\kappa = 2$.

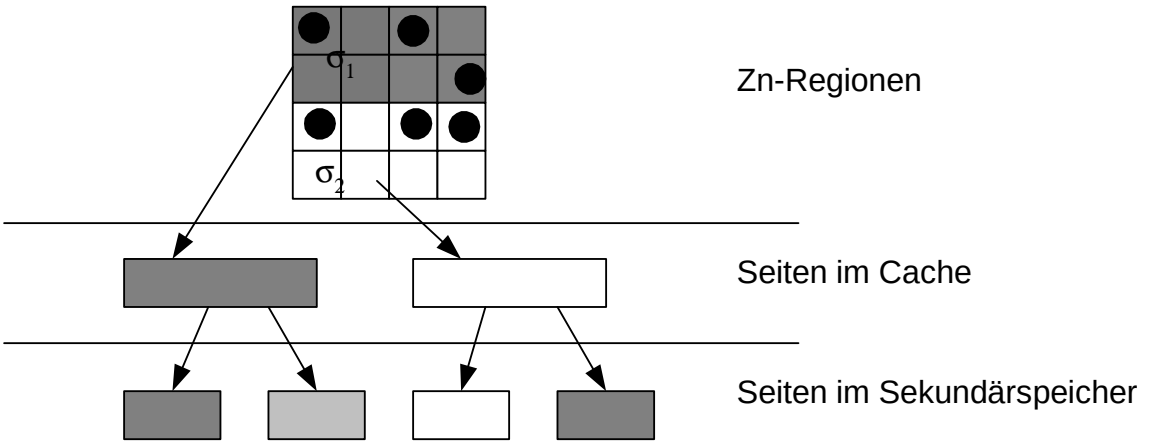


Abbildung 7-14: Z_u-Regionen

Nachdem die Z_u-Region eine *u*-fache Kapazität einer Datenbankseite *p* besitzt, wird jede Z_u-Region auf maximal *u* Datenbankseiten und minimal $\frac{u}{2}$ Datenbankseiten abgebildet. Es gilt:

$$\rho^u \leftrightarrow v \Leftrightarrow (x \in \text{content}(v) \Leftrightarrow x \in \{\rho^u \cap \text{content}(v)\}) \qquad \text{Gl: 7-17}$$

Die Funktion content(*v*) gibt die Tupel zurück, die in der Arbeitsspeicherseite zur Zeit gespeichert sind. Allgemein besitzt somit eine Arbeitsspeicherseite *v*, auf die genau eine ρ^u zugeordnet ist, die *u*-fache Kapazität einer Datenbankseite. Im Folgenden wird der Algorithmus beschrieben, der eine Z_u-Region auf eine Menge von Seiten abbildet. *u* bezeichnet hierbei die Seitenskalierung, *n* die Anzahl der Sekundärspeicherseiten, κ die Kapazität einer Sekundärspeicherseite *p*. |*p*| bezeichnet die Tupelanzahl einer Seite *p*. |*v*| bezeichnet die Anzahl der Tupel, die in der Z_u-Region im Arbeitsspeicher abgelegt sind und in die Z_u-Region fallen. Mit *g* wird die Anzahl der vollen Datenbankseiten *p* bezeichnet, die aus einer Arbeitsspeicherseite *v* erstellt werden können. Es gilt somit:

$$g = \left\lfloor \frac{|v|}{\kappa} \right\rfloor \qquad \text{Gl: 7-18}$$

r bezeichnet die Tupelzahl der nicht vollständig ausgelasteten Datenbankseite *p* und es gilt:

$$r = |v| - g \cdot \kappa \qquad \text{Gl: 7-19}$$

Die Anzahl der Tupel $|v|$, die in einer Arbeitsspeicherseite v abgelegt sind, ist durch folgende Gleichung bestimmt, da die Cacheverwaltung den Splitalgorithmus des B-Baums benutzt.

$$\left\lceil \frac{u \cdot \kappa}{2} \right\rceil \leq |v| \leq u\kappa \qquad \text{Gl: 7-20}$$

Diese Gleichung gilt für alle Knoten, außer für die Wurzel (siehe Definition 3-16). Für die Wurzel gilt unter der Voraussetzung, dass es sich hier um kein Blatt handelt:

$$1 \leq |v| \leq u\kappa \qquad \text{Gl: 7-21}$$

Der Algorithmus basiert auf folgenden Schritten:

- Falls $r \geq \left\lceil \frac{\kappa}{2} \right\rceil$ ist, dann erzeuge aus den ersten $g \cdot \kappa$ Tupeln g Sekundärspeicherseiten maximaler Seitenauslastung. Füge r Tupel in eine weitere Datenbankseite p.
- Gilt $r < \left\lceil \frac{\kappa}{2} \right\rceil$, dann erzeuge aus den ersten $(g-1) \cdot \kappa$ Tupel $g-1$ Sekundärspeicherseiten maximaler Seitenauslastung. Teile die verbleibenden $r + \kappa$ Tupel auf zwei weitere Datenbankseiten auf.

Abbildung 7-15 zeigt die wesentlichen Schritte des Algorithmus[16].

[16] Ein ähnlicher Algorithmus ist in [HickZ00] auf Seite 24 zu finden, der vom Autor dieser Arbeit betreut wurde.

Marke Abbildung Z_n-Region auf Seiten des Sekundärspeichers

```
Eingabe:    v      : Arbeitsspeicherseite
Ausgabe:    {p1,p2,...  pn}    : Region

g = |v| / κ
r = |v| mod κ
if r == 0 then      //Alle Seiten besitzen die maximale
      n = g;        //Seitenauslastung
else
      n = g + 1     // Es existieren zwei Seiten mit nicht maximaler Seitenauslastung
endif

// g –1  Sekundärspeicherseiten können immer  mit einer
// maximalen Seitenauslastung erzeugt werden.
for 1 to g – 1 do
      Erzeuge aus den nächsten κ Tupeln eine neue Datenbankseite
end
if r == 0 then      // es existieren nur Seiten mit maximaler
                    // Seitenauslastung
      Erzeuge aus den letzten κ Tupeln eine neue Seite p
else if r < (κ /2) then // Diese Seite würde der Seitenauslastung von 50 %
                        // mit r Tupel nicht erfüllen.
      Erzeuge aus den nächsten ⌈(k+r)/2⌉ Tupeln eine Seite p;
      Erzeuge aus den restlichen ⌊(k+r)/2⌋ Tupeln eine Seite p;
else // (κ/2) ≤ r < κ
      if n > 1 then
            Erzeuge aus den nächsten k Tupeln eine Seite
      end if
      Erzeuge aus den verbleibenden r Tupeln eine neue Seite
end if
```

Abbildung 7-15: Abbildung von Z_s-Regionen auf Sekundärspeicherseiten

Worst-Case-Abschätzung:

Eine Arbeitsspeicherseite *v* enthält genau |*v*| Tupel. *v* kann somit auf

$$\left\lceil \frac{|v|}{\kappa} \right\rceil \qquad \text{Gl: 7-22}$$

Datenbankseiten aufgeteilt werden. Hierbei entsteht maximal eine Seite, die weniger als κ Tupel enthält. Sie wird als *Problemseite* bezeichnet. Der *durchschnittliche Füllungsgrad F* beträgt weniger als 100% genau dann, wenn mindestens eine *Problemseite* existiert. Nach Gl: 7-12 ist die Seitenauslastung um so geringer, je weniger Tupel auf einer Seite abgelegt sind, da es sich hierbei um eine lineare Funktion handelt. Der durchschnittliche Füllungsgrad *F* einer Seitenmenge *P*, die durch das Programm Abbildung 7-15 erzeugt wird, ist minimal, wenn die Anzahl der Problemseiten maximal ist und das B-Baum-Kriterium der 50%-Seitenauslastung gilt.

min|*v*| bezeichnet die minimale Anzahl der Tupel einer Arbeitsspeicherseite *v*, die den minimalen Füllungsgrad *F* auf den Datenbankseiten *P* erzeugt. κ bezeichnet die Kapazität einer Seite. *u* den Skalierungsfaktor einer Arbeitsspeicherseite *v* zu einer Datenbankseite *p*. Nach Gl: 7-20 enthält eine Arbeitsspeicherseite wegen des Splitalgorithmus des B-Baums mindestens

$$\left\lceil \frac{u \cdot \kappa}{2} \right\rceil \qquad \text{Gl: 7-23}$$

Tupel. Die Anzahl der vollen Datenbankseiten ergibt sich nach Gl: 7-23 aus

$$g = \left\lfloor \frac{|v|}{\kappa} \right\rfloor = \left\lfloor \frac{\left\lceil \frac{u \cdot \kappa}{2} \right\rceil}{\kappa} \right\rfloor \qquad \text{Gl: 7-24}$$

Multipliziert man *g* mit κ erhält man die Anzahl der Tupel, die in den vollen Datenbankseiten gespeichert sind. Ist der Skalierungsfaktor *u* gerade und größer 2 würde man einen durchschnittlichen Füllungsgrad von 1 erhalten, was einer 100%-Seitenauslastung entspricht. Um den schlechtesten Fall zu erhalten, muss also noch ein weiteres Tupel in *v* enthalten sein. Für einen ungeraden Skalierungsfaktor *u* erhält man eine *Problemseite* mit einem Füllungsgrad von $\left\lceil \frac{\kappa}{2} \right\rceil$ Tupel, was mindestens eine 50%-Seitenauslastung der Datenbankseite bedeutet. Die minimale Seitenauslastung für ungerade Skalierungsfaktoren entsteht jedoch, wenn die 50%-Problemseite vollständig gefüllt wird und die nächste Seite nur ein Tupel enthält.

Somit ergibt sich für *min*(|*v*|):

$$\min(|v|) = \begin{cases} \underbrace{\left\lfloor \frac{\left\lceil ((u+1)\cdot\kappa)/2 \right\rceil}{\kappa} \right\rfloor}_{\text{Anzahl der 100\% Seiten}} \cdot \kappa + 1 & , \quad u = (2 \cdot i) - 1 \wedge i \in [2,..,\infty] \\ \underbrace{\left\lfloor \frac{\left\lceil (u\cdot\kappa)/2 \right\rceil}{\kappa} \right\rfloor}_{\text{Anzahl der 100\% Seiten}} \cdot \kappa + 1 & , \quad u = (2 \cdot i) \wedge i \in [2,..,\infty] \end{cases} \qquad \text{Gl: 7-25}$$

Dies entspricht:

$$\min(|v|) = \begin{cases} \frac{(u+1)}{2} \cdot \kappa + 1 & , \quad u = (2 \cdot i) - 1 \wedge i \in [2,..,\infty] \\ \frac{u}{2} \cdot \kappa + 1 & , \quad u = (2 \cdot i) \wedge i \in [2,..,\infty] \end{cases} \qquad \text{Gl: 7-26}$$

Nach Gl: 7-13 ist der durchschnittliche Füllungsgrad F der Quotient aus der Anzahl der gespeicherten Tupel $|P|$ auf der Seitenmenge P geteilt durch die Kapazität der Seitenmenge P. Für min ($|v|$) ergibt sich eine Seitenmenge von:

$$P = \left\lfloor \frac{\min(|v|)}{\kappa} \right\rfloor + 1 \qquad \text{Gl: 7-27}$$

Der Summand 1 entsteht durch die Problemseite, die sowohl für den geraden und ungeraden Fall entstehen. Für den durchschnittlichen Füllungsgrad ergibt sich somit:

$$F_{\text{Füllungsgrad}} = \frac{|P|}{P \cdot \kappa} = \frac{\min(|v|)}{\left(\left\lfloor \frac{\min(|v|)}{\kappa} \right\rfloor + 1\right) \cdot \kappa} \qquad \text{Gl: 7-28}$$

Für gerade und ungerade Skalierungsfaktoren u ist die Funktion des Füllungsgrads streng monoton steigend. Die Funktionen werden jeweils durch die Asymptote y = 1 nach oben begrenzt. Der Füllungsgrad nähert sich mit steigendem Skalierungsfaktor 1 an. Für eine Seitenkapazität von 20 und einen Skalierungsfaktor von 7 kann bereits eine Seitenauslastung von über 80% garantiert werden.

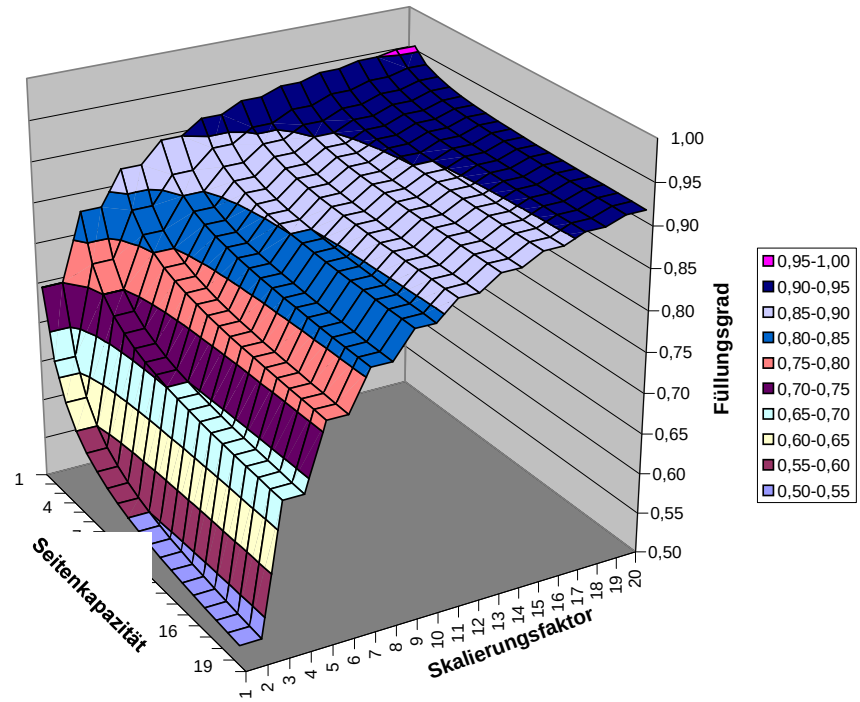


Abbildung 7-16: Füllungsgrad in Abhängigkeit von Seitenkapazität und Skalierungsfaktor pro Seite.

Abbildung 7-16 zeigt die garantierte Seitenauslastung in Abhängigkeit von der Seitenkapazität κ und dem Skalierungsfaktor *u*. Eine Verbesserung der Seitenauslastung entsteht beim Übergang von geraden zu ungeraden Skalierungsfaktoren. Die Steigung nimmt mit zunehmendem Skalierungsfaktor ab.

Messung:

Für das TPC-H Schema wurde die Lineitem-Tabelle 8 Mal aufgebaut. Es wurden jeweils 1000000 Tupel geladen. Hierbei wurde der Skalierungsfaktor *u* von 2 bis 9 variiert.

Skalierungsfaktor *u*	Seiten
2	107557
3	103862
4	97292
5	97419
6	93094
7	94368
8	91113
9	92453

Tabelle 7-2

Tabelle 7-2 und Abbildung 7-17 zeigt die Anzahl der Seiten in Abhängigkeit des Skalierungsgrades *u*. Mit steigendem Skalierungsfaktor u verbessert sich die Seitenauslastung für die geraden und ungeraden Skalierungsfaktoren.

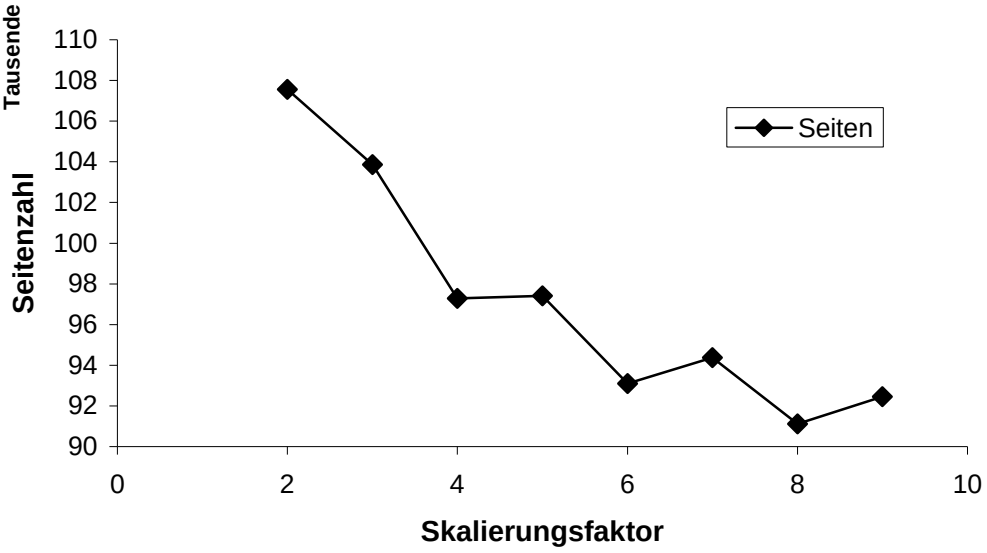


Abbildung 7-17

Beim Übergang vom Skalierungsgrad 4 zu 5 , 6 zu 7 und 8 zu 9 erhöht sich jeweils die Anzahl der Seiten. Dies liegt darin, dass der durchschnittliche Füllungsgrad *F* der Arbeitsspeicherseiten bei der gegebenen Datenverteilung bei 70 % liegt.

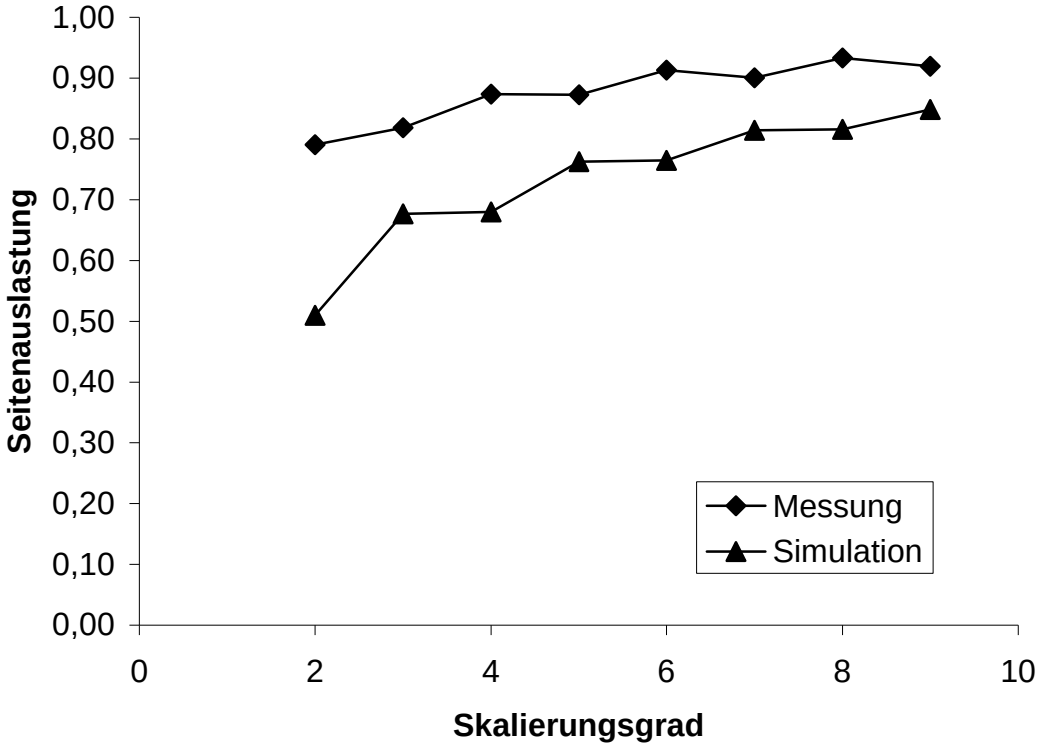


Abbildung 7-18

Abbildung 7-18 zeigt den durchschnittlichen Füllungsgrad in Abhängigkeit vom Skalierungsfaktor. Sie liegt über dem theoretisch hergeleiteten Mindestfüllungsgrad und bestätigt somit das Modell.

7.2.2.5.2 Dynamische Speicherseiten

Im vorangegangenen Abschnitt wurde das Konzept der Z-Unterräume sowie der Z_u-Unterräume eingeführt. Wie bereits erwähnt, hat eine Vergrößerung des Faktors *u* zum einen eine Erhöhung der Seitenauslastung zur Folge, zum anderen steigt jedoch der Bedarf an temporärem Speicher.

Eine weitere Optimierungsmöglichkeit ist das Konzept der variablen Z-Unterräume. Hierbei wird die Forderung, dass ein Z-Unterraum aus *u* Seiten abgelegt werden kann, fallen gelassen. Somit besteht keine bestimmte Kapazität für einen Z-Unterraum. Dies wird nur noch durch den verfügbaren Cache im Arbeitsspeicher bestimmt. Der Cache wird dann

solange mit Tupel gefüllt, bis es zu einem Überlauf kommt. Beim Leeren des Caches werden die einzelnen Z-Unterräume auf die Seiten verteilt. Dies führt dann fast zu einem sequenziellen Schreiben auf dem Sekundärspeicher (Seitenclusterung) mit einer 100%-igen Seitenauslastung wie beim Massenladen. Eine Beschreibung ist in [HickZ00] zu finden.

7.3 Leistungsanalyse

Für die Leistungsanalyse wird das Kostenmodell aus Kapitel 3.2.1.2 herangezogen. Hierbei betrachten wir die Verarbeitungszeit sowie den temporären Speicherbedarf für die Erzeugung einer Z-Zerlegung (Partitionierung) aus einer sortierten Folge ($\langle R\rangle, <_{dim}$). Der theoretische Vergleich berücksichtigt den TempTris-Algorithmus und das externe Sortieren [Knu98v3]. In [FenKM00] wird ein Verfahren zur Erzeugung von UB-Bäumen vorgestellt, das auf externem Sortieren basiert. Dies wird zum Vergleich herangezogen. Für die E/A-Kosten und den Arbeitsspeicherplatz werden Kostenfunktionen angegeben. Für die CPU-Komplexität wird die Klasse des TempTris-Algorithmus bestimmt.

7.3.1 Kostenfunktionen

Unter Verwendung des Kostenmodells aus Kapitel 3.2.1.2 werden die Kosten für eine sortierte Folge ($\langle R\rangle, <, k$) berechnet. P bezeichnet die Größe der sortierten Folge ($\langle R\rangle, <, k$) in Seiten. Für das *Lesen* bzw. *Schreiben* ergeben sich somit nach Gl: 4-59 und Gl: 4-66 folgende *E/A*-Kosten:

$$c_l(P) = c_s(P) = c_{E/A-fts}(P) = \left\lceil \frac{P}{C} \right\rceil c_{E/A} \qquad \text{Gl: 7-29}$$

Die E/A-Kosten für den TempTris-Algorithmus bestehen aus zwei Anteilen, dem Lesen der Daten und dem Schreiben der mehrdimensionalen Partitionierung auf den Sekundärspeicher. Beim Lesevorgang kann der Prefetchfaktor C des Sekundärspeichers vollständig ausgenutzt werden. Beim Schreibvorgang hängt dies von der Datenverteilung und der Cachegröße M ab. Wird die Optimierungsstrategie der Z_n-*Unterräume* oder der *dynamischen Seiten* herangezogen, kann üblicherweise der Prefetchfaktor C auch für das Schreiben ausgenutzt werden. Da die Seitenauslastung nicht 100% ist, muss beim Schreiben der Daten die Seitenauslastung berücksichtigt werden. Das sequenzielle Lesen in der Lesephase in Verbindung mit dem TempTris-Algorithmus ergibt die Kostenfunktion c_{TempTris}. Dies führt zur folgenden Kostenfunktion:

$$c_{TempTris}(P) = c_l(P) + c_s\left(\frac{P}{F_{Füllungsgrad}}\right) \qquad \text{Gl: 7-30}$$

Das Massenladenverfahren, das auf Sortieren durch Verschmelzen (merge sort) [FenKM00] basiert, teilt den Ladevorgang in eine *Initialisierungsphase*, in der die Daten nach und nach in den Arbeitsspeicher gelesen werden, und durch ein internes Sortierverfahren, wie z.B. Quicksort oder Merge-Sort [CorLR90], als sortierte Initialläufe wieder auf den Sekundärspeicher sowie in eine *Verschmelzungsphase* zurückgeschrieben werden. M ist die Größe des Arbeitsspeicherbedarfs in Speicherseiten, m ist der Verschmelzungsgrad beim externen Sortieren durch Verschmelzen. Es wird wieder angenommen, dass $P > M$

gilt. In der Verschmelzungsphase kommt üblicherweise das *Vielweg-Mischen* zum Einsatz, d.h. es werden *m* Läufe parallel miteinander in einem neuen, sortierten Lauf verschmolzen. Wird in der Initialisierungsphase das sequenzielle Lesen (FTS) zur Erzeugung der Initialläufe ($c_l + c_s$) in Verbindung mit dem *Sort-Merge-Algorithmus* herangezogen, erhält man die Kostenfunktion $c_{\text{merge-sort}}$. Da wir nicht zwischen der Lese- und Schreib-Operation unterscheiden, schreiben wir einfach $c_{l/s}(P)$ Mit Gl: 4-12 und Gl: 7-29 erhält man für das externe Sortieren folgende Kostenfunktion:

$$c_{merge-sort} = \begin{cases} \underbrace{2 \cdot c_{l/s}(P)}_{\text{Initialisierungsphase}} & , \quad \text{falls } P \le M \\ \underbrace{2 \cdot c_{l/s}(P)}_{\text{Initialisierungsphase}} + \underbrace{2 \cdot c_{l/s}(P)}_{\text{Mischphase}} & , \quad \text{falls } \frac{P}{M} \le m \\ \underbrace{2 \cdot c_{l/s}(P)}_{\text{Initialisierungsphase}} + \underbrace{2 \cdot c_{l/s}(P) \cdot \left\lceil \log_m \frac{P}{M} \right\rceil}_{\text{Mergephase}} & , \quad \text{sonst} \end{cases} \qquad \text{Gl: 7-31}$$

Falls $M > P$ gilt, handelt es sich nicht um ein externes Sortierproblem. Es kann durch einen internen Sortieralgorithmus verarbeitet werden. In diesem Fall ist der Verschmelzungsfaktor 0. Falls $\frac{P}{M} < m$ gilt, kann das Ergebnis durch einen Verschmelzungsvorgang erzeugt werden und man erhält:

$$c_{merge-sort} = 4 \cdot c_{l/s}(P) \qquad \text{Gl: 7-32}$$

Dies ist also der optimale Fall. Um diesen optimalen Fall möglichst lange zu erhalten, sollte der Verschmelzungsgrad *m* möglichst groß gewählt werden. Der maximale Verschmelzungsgrad ist abhängig von der Größe des Arbeitsspeichers *M* und dem Prefetchingfaktor *C*. Um die optimale Auslastung des Systems zu erhalten, wird für jeden Ausgabestrom ein Puffer der Größe 2 *C* reserviert [Gra93]. Für einen Arbeitsspeicher der Größe *M* ergibt sich ein Verschmelzungsgrad *m*:

$$\begin{aligned} & \overbrace{m \cdot 2C}^{Lesepuffer} + \overbrace{2C}^{Schreibpuffer} = M \\ \Leftrightarrow \quad & 2C(m+1) = M \\ \Leftrightarrow \quad & m+1 = \frac{M}{2C} \\ \Leftrightarrow \quad & m = \frac{M}{2C} - 1 \end{aligned} \qquad \text{Gl: 7-33}$$

Da $m \in \mathbb{N}$ gilt, erhält man

$$m = \left\lfloor \frac{M}{2C} \right\rfloor - 1 \qquad \text{Gl: 7-34}$$

Um den in Gl: 7-32 beschriebenen optimalen Fall zu erhalten, muss für den Verzweigungsgrad weiter gelten:

$$m \geq \left\lfloor \frac{P}{M} \right\rfloor - 1 \qquad \text{Gl: 7-35}$$

Interessant ist der Zusammenhang zwischen dem Arbeitsspeicher und der maximalen Tabellengröße, die in linearer Zeit berechnet werden kann.

Satz 7-2:

Gegeben sei ein Prefetchfaktor C und ein Arbeitsspeicher der Größe $M = 2 \cdot C \cdot (m+1)$. Dann ist die maximale Tabellengröße, die in linearer Zeit berechnet werden kann:

$$P = \frac{M^2}{2C} - M$$

Beweis:

Damit das externe Sortieren linear ist, muss der logarithmische Faktor 0 oder 1 sein (siehe Gl: 4-12 und Gl: 7-30). Ist der Faktor 0, dann ist $P < M$ und es handelt sich um ein internes Sortierproblem, das in 2 P E/A-Operationen abgearbeitet werden kann.

Ist der logarithmische Faktor 1, so werden alle Daten genau 2 Mal gelesen und geschrieben. Bei gegebenem M erhält man:

$$\begin{aligned}
& \log_m \frac{P}{M} = 1 \\
\Leftrightarrow \quad & \frac{P}{M} = m \\
\Leftrightarrow \quad & P = m \cdot M \\
\Leftrightarrow \quad & P = \left(\frac{M}{2C} - 1 \right) M \quad \text{mit } m = \frac{M}{2C} - 1 \\
\Leftrightarrow \quad & P = \frac{M^2}{2C} - M
\end{aligned}$$

q.e.d.

Abbildung 7-19 zeigt das Verhältnis zwischen dem Arbeitsspeicher M und der maximalen Tabellengröße, die mit externem Sortieren in linearer Zeit sortiert werden kann. Hierbei wurde der Prefetchfaktor von 1 bis 16 verändert. Es handelt sich hierbei um eine exponentielle Funktion. Die Funktion mit dem Prefetchfaktor 1 wächst am schnellsten, da der Divisor am kleinsten ist.

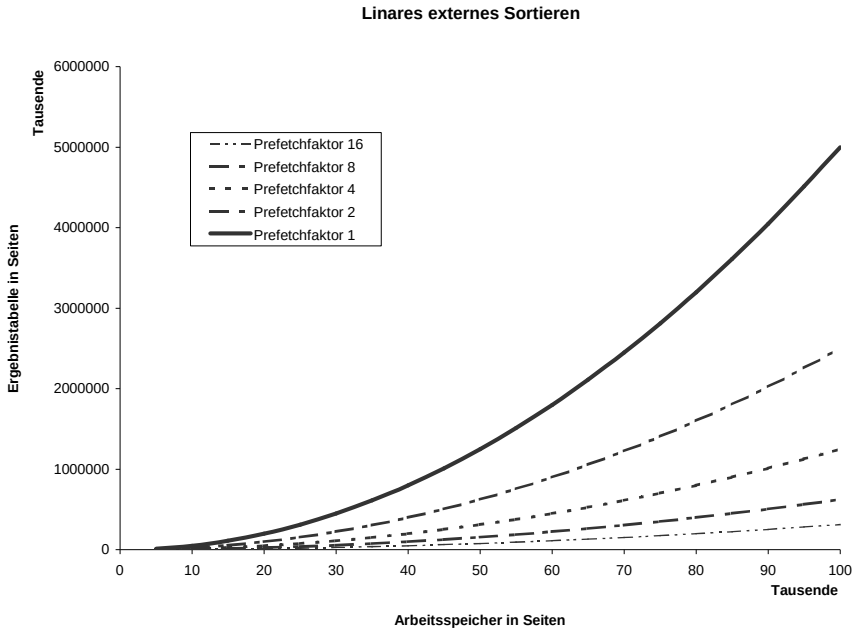


Abbildung 7-19: Externes Sortieren mit logarithmischem Faktor 1

7.3.2 E/A-Komplexität

In heutigen Systemen werden bei einer E/A-Operation 8 Seiten gelesen. Dementsprechend setzten wir den Prefetchfaktor in der theoretischen Analyse mit C = 8. Die durchschnittliche Positionierungszeit sei t_π= 10 *ms*, die durchschnittliche Übertragungszeit sei t_τ = 1 *ms*. Der Cache sei 32 MB, womit man mit einer Seitengröße von 2 KB im besten Fall einen Verschmelzungsgrad von m = 1024 erreicht. Somit kann man maximal eine Tabelle der Größe

$$P = \frac{M^2}{2 \cdot C} - M = \frac{(32MB)^2}{32KB \cdot} - 32MB \approx 32GB \qquad \text{Gl: 7-36}$$

mit linearer E/A-Komplexität sortieren. Wir setzen den Füllungsgrad der erzeugten Seiten auf 81% (siehe 7.2.2.5).

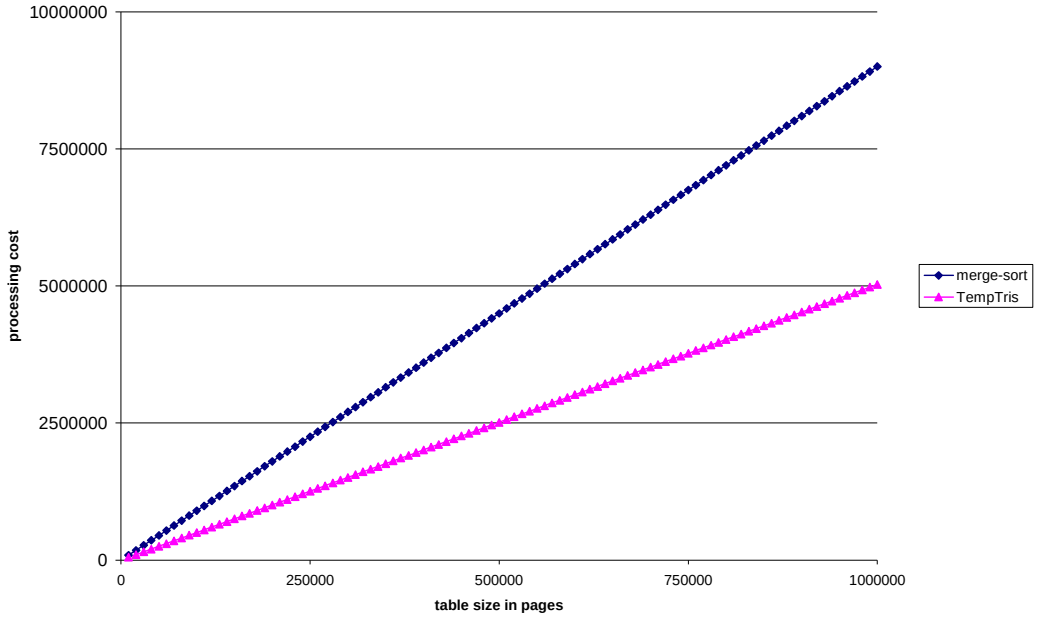


Abbildung 7-20: E/A-Kosten, TempTris-Algorithmus und Sort-Merge-Algorithmus

Abbildung 7-20 zeigt die E/A-Kosten, die bei der Erzeugung der mehrdimensionalen Zerlegung aus einer 1-dimensional sortieren Folge von Daten entstehen. Hierbei wurde die Tabellengröße variiert. Der Graph zeigt, basierend auf dem Kostenmodell von Abschnitt 7.3.1, dass der TempTris-Algorithmus nur die Hälfte der E/A-Operationen benötigt, wie der Sort-Merge-Algorithmus.

7.3.3 Speicherplatz-Komplexität

Die bereitgestellte Cachegröße für den Sort-Merge-Algorithmus hat entscheidenden Einfluss auf die E/A-Komplexität (siehe Gl: 7-31). Für den theoretischen Vergleich betrachten wir zwei Fälle:

Fall 1: 2 $c_{l/s}$ (P)

Geht man davon aus, dass die Folge von Tupeln nur einmal gelesen und geschrieben werden soll, ergibt sich für das externe Sortieren nach Gl: 7-31:

$$M_{merge-sort} = P \qquad \text{Gl: 7-37}$$

Für den TempTris-Algorithmus ergibt sich für gleichverteilte Daten eine Speicherplatz-Komplexität:

Satz 7-3:

Gegeben ist eine d-dimensionale Folge von Tupeln der Größe P. Der benötigte Cache für die Erstellung eines UB-Baums mit dem TempTris-Algorithmus mit einer E/A-Komplexität von 2 $c_{l/s}$ (P) beträgt:

$$cache_{TempTris}(P,d) = \sqrt[d]{P^{d-1}}$$

Beweis:

$$cache_{TempTris}(P,d) = \frac{P}{2^{recursive\_splits_{\dim}}} = \frac{P}{2^{\frac{\log_2 P}{d}}} = \sqrt[d]{P^{d-1}}$$

q.e.d.

Somit benötigt der TempTris-Algorithmus in diesem Fall weniger temporären Speicher als der Sort-Merge-Algorithmus. Hier ist jedoch zu beachten, dass die Cacheverwaltung, also der Z-Index und der T-Index, nicht berücksichtig worden sind.

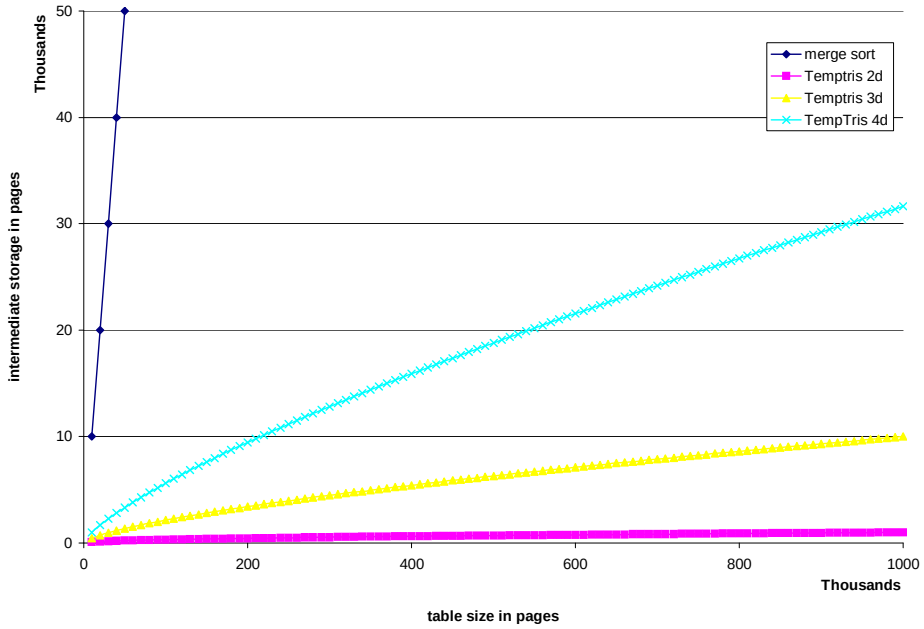


Abbildung 7-21: Größe des temporären Arbeitsspeichers

Abbildung 7-21 zeigt den Bedarf des temporären Arbeitsspeichers des TempTris-Algorithmus bzw. des Sort-Merge-Algorithmus, um einen UB-Baum durch Massenladen zu erzeugen. Die Größe der Tabelle variiert von 10 K bis 1M Seiten.

Nimmt man eine Seitengröße von 2 KB an, benötigt man z.B. für eine 4-dimensionale Zerlegung mit dem TempTris-Algorithmus für eine 2 GB große Tabelle lediglich 64 MB Arbeitsspeicher. Für eine 2-dimensionale Zerlegung benötigt man sogar nur 2 MB Arbeitsspeicher.

Fall 2: $4c_{I/s}$ (P)

Fall 2 betrachtet die Situation, dass die Daten genau 2 Mal gelesen und geschrieben werden. Die Daten werden zu Begin in den Arbeitsspeicher gelesen. Dort werden die Initialläufe erzeugt. Diese werden dann auf den Sekundärspeicher zurückgeschrieben. Die Anzahl der Initialläufe ist kleiner als der Verschmelzungsgrad, so dass das sortierte

Ergebnis durch einmaliges Mischen erzeugt werden kann. Hierdurch entstehen wieder ein Lese- und Schreib-Vorgang. Betrachten wir nun den Fall, dass für die Eingabe gilt:

$$M < P \leq m \cdot M \qquad \text{Gl: 7-38}$$

Um eine Tabelle mit *P* Seiten extern zu sortieren, benötigt man einen Arbeitsspeicher von (siehe Gl: 7-31, Satz 7-2):

$$M = \sqrt{2C \cdot P + C^2} + C \qquad \text{Gl: 7-39}$$

Seiten.

Beweis:

$$\begin{aligned}
& P && = && \frac{M^2}{2 \cdot C} - M && \left| \cdot 2 \cdot C \right. \\
\Leftrightarrow\ & M^2 - M \cdot 2 \cdot C && = && P \cdot 2 \cdot C && \left| + C^2 \right. \\
\Leftrightarrow\ & M^2 - M \cdot 2 \cdot C + C^2 && = && P \cdot 2 \cdot C + C^2 && \\
\Leftrightarrow\ & (M - C)^2 && = && P \cdot 2 \cdot C + C^2 && \left| \sqrt{\ } \right. \\
\Leftrightarrow\ & M - C && = && \sqrt{P \cdot 2 \cdot C + C^2} && \left| + C \right. \\
\Leftrightarrow\ & M && = && C + \sqrt{P \cdot 2 \cdot C + C^2} && \\
\Leftrightarrow\ & M && = && C + \sqrt{C(P \cdot 2 \cdot + C)} &&
\end{aligned} \qquad \text{Gl: 7-40}$$

q.e.d.

Der Arbeitsspeicher für den TempTris-Algorithmus und externes Sortieren sind in Fall 2 sublinear. Beim externen Sortieren handelt es sich um die zweite Wurzel. Sie ist unabhängig von der Anzahl der Dimensionen. Der Arbeitsspeicherbedarf des TempTris-Algorithmus ist jedoch abhängig von der Dimensionalität der Daten. Hierbei benötig man d-te Wurzel aus der (d-1)-ten Potenz der Datenmenge als Arbeitsspeicher.

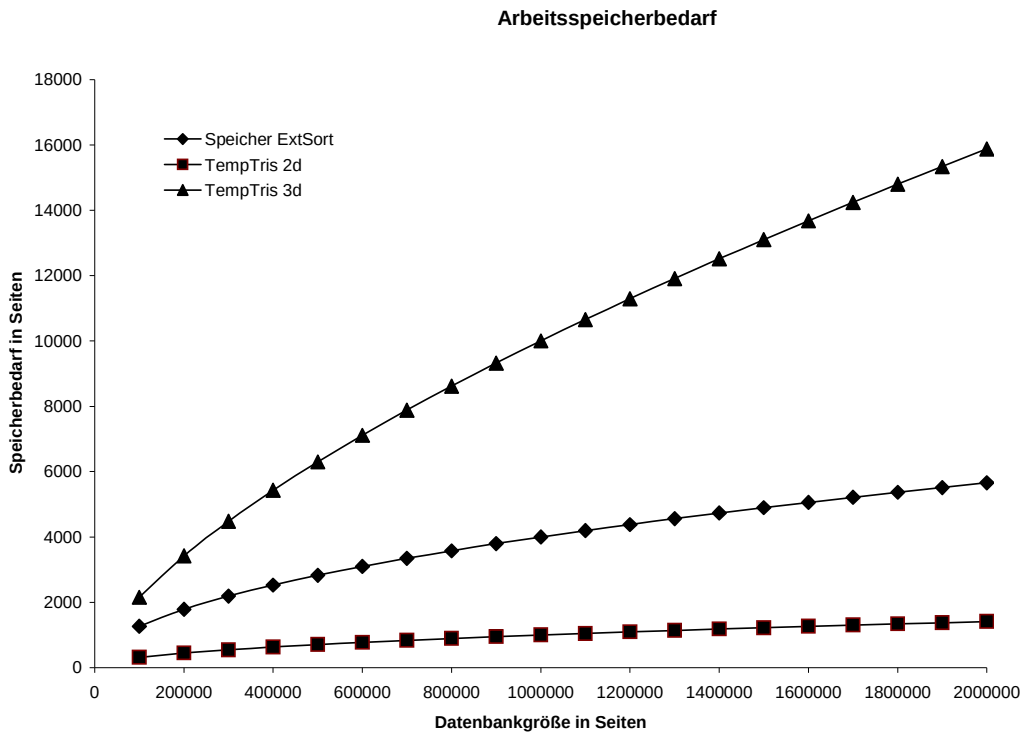


Abbildung 7-22: Arbeitsspeicherbedarf

Abbildung 7-22 zeigt den Arbeitspeicherbedarf des TempTris-Algorithmus und des externen Sortierens in Abhängigkeit der Datenbankgröße in Seiten. Für eine 2-dimensionale Datenbank liegt der Arbeitspeicherbedarf *M* für das externe Sortieren um Faktor 4 über dem TempTris-Algorithmus. Bei einer 3-dimensionalen Datenbank liegt der Arbeitsspeicher des TempTris-Algorithmus über dem des externen Sortierens. Die Verhältnisse sind der Tabelle 7-3 zu entnehmen.

Seiten	ExtSort in Seiten	TempTris 2d in Seiten	E/T2d	TempTris 3d in Seiten	S/T3d
100000	1265	316	4	2154	0,58
200000	1789	447	4	3420	0,52
300000	2191	548	4	4481	0,48
400000	2530	632	4	5429	0,46
500000	2828	707	4	6300	0,44
600000	3098	775	4	7114	0,43
700000	3347	837	4	7884	0,42
800000	3578	894	4	8618	0,41
900000	3795	949	4	9322	0,40
1000000	4000	1000	4	10000	0,40
1100000	4195	1049	4	10656	0,39
1200000	4382	1095	4	11292	0,38
1300000	4561	1140	4	11911	0,38
1400000	4733	1183	4	12515	0,37
1500000	4899	1225	4	13104	0,37
1600000	5060	1265	4	13680	0,36
1700000	5215	1304	4	14244	0,36
1800000	5367	1342	4	14797	0,36
1900000	5514	1378	4	15340	0,35
2000000	5657	1414	4	15874	0,35

Tabelle 7-3

Allgemein ist somit der Arbeitsspeicherbedarf ab einer 3-dimensionalen Datenbank für den TempTris-Algorithmus größer als der für das externe Sortieren.

7.3.4 Seitenauslastung

Wie bereits erwähnt, kann der TempTris-Algorithmus durch Optimierung eine Seitenauslastung von 80 bis knapp 100% erreichen. Dies ist schlechter als das Massenladen mit dem Sort-Merge-Algorithmus, der eine Seitenauslastung von 100% garantieren kann. Dennoch stellt der TempTris-Algorithmus eine Alternative dar, falls:

- keine 100%-ige Seitenauslastung gefordert wird. Dies ist speziell bei OLTP-Systemen gefordert, bei denen Daten während des Betriebs weiter hinzugefügt werden. Dort wird ein gewisser Prozentsatz, ca. 20 bis 30% der Seitenkapazität freigelassen, z.B. durch das Setzen des Parameters PCTFREE in Oracle.
- die mehrdimensionale Zerlegung für Zwischenergebnisse verwendet wird, die danach wieder gelöscht wird. Hier ist z.B. die Berechnung von Aggregationsnetzen zu nennen (siehe Kapitel 9.2.5).
-

Die E/A-Kostenreduzierung des TempTris-Algorithmus bei der Erzeugung einer mehrdimensionalen Partitionierung gegenüber dem externen Sortieren überwiegt bei Weitem die schlechtere Antwortzeit von bis zu 20%, die durch die geringere Seitenauslastung verursacht wird.

7.3.5 CPU-Komplexität

Die CPU-Komplexität des TempTris-Algorithmus wird durch die Speicherverwaltung für den noch zu verarbeitenden Raum Φ bestimmt. Die wesentlichen Kosten sind hierbei das Einfügen und Löschen aus dem T-Index sowie Z-Index und die jeweilige Schlüsselberechnung. Die Berechnung der Z-Adresse sowie die minimale T-Adresse kann in linearer Zeit

$$C_{cpu\text{-}minT\text{-}Adresse} = O(n)$$
$$C_{cpu\text{-}Z\text{-}Adresse} = O(n) \qquad \text{Gl: 7-41}$$

berechnet werden. Da es sich bei dem Z-Index und dem T-Index um einen AVL-Baum [AdeL62] handelt, werden die Operationen in

$$C_{cpu\text{-}einfügen} = O\,(\ln n) \qquad \text{Gl: 7-42}$$

und

$$C_{cpu\text{-}löschen} = O\,(\ln n) \qquad \text{Gl: 7-43}$$

durchgeführt. Somit ergibt sich für die CPU-Komplexität als obere Grenze:

$$C_{cpu\text{-}TempTris} = O\,(\, n \log n) \qquad \text{Gl: 7-44}$$

7.4 Messungen

Die Messungen wurden auf einer Prototypimplementierung des UB-Baums durchgeführt. Eine genaue Beschreibung ist in Kapitel 8 zu finden. Sie ist als Middleware zwischen dem DBMS und der DB-Applikation realisiert worden. Der TempTris-Algorithmus wurde für die TransBase-UB-Baum-Middleware realisiert. Für die Messung wurde ein Intel Pentium III 500 CPU mit 128 MB RAM eingesetzt, für die Datenbank wurde eine 9 GB Festplatte mit einer durchschnittlichen Positionierungszeit von 7,9 *ms* und einer Übertragungszeit von 0,6 *ms* verwendet. Die Messmaschine lief unter SuSE Linux 6.2 Kernel Version 2.2.10 SMP. Um nicht vorhersehbare Cache-Effekte zu unterdrücken, wurde der Schreib-Cache deaktiviert.

Menge	01	02	03	04	05	06
Größe	32 MB	64MB	128MB	256MB	512MB	1024MB
Tupel	335544	655360	1310720	2621440	5242880	10485760

Tabelle 7-4: Größe der Flat-Files

Um die Leistungssteigerung des TempTris-Algorithmus zu zeigen, wurden 6 3-dimensionale Daten-Würfel mit gleichverteilten Daten erzeugt. Jeder Daten-Würfel wurde von unterschiedlichen Daten erzeugt. Die Größe eines Tupels war 100 Bytes, die Seitengröße betrug 2 KB. Die Größe der einzelnen Flat-Files ist in Tabelle 7-4 zusammengefasst.

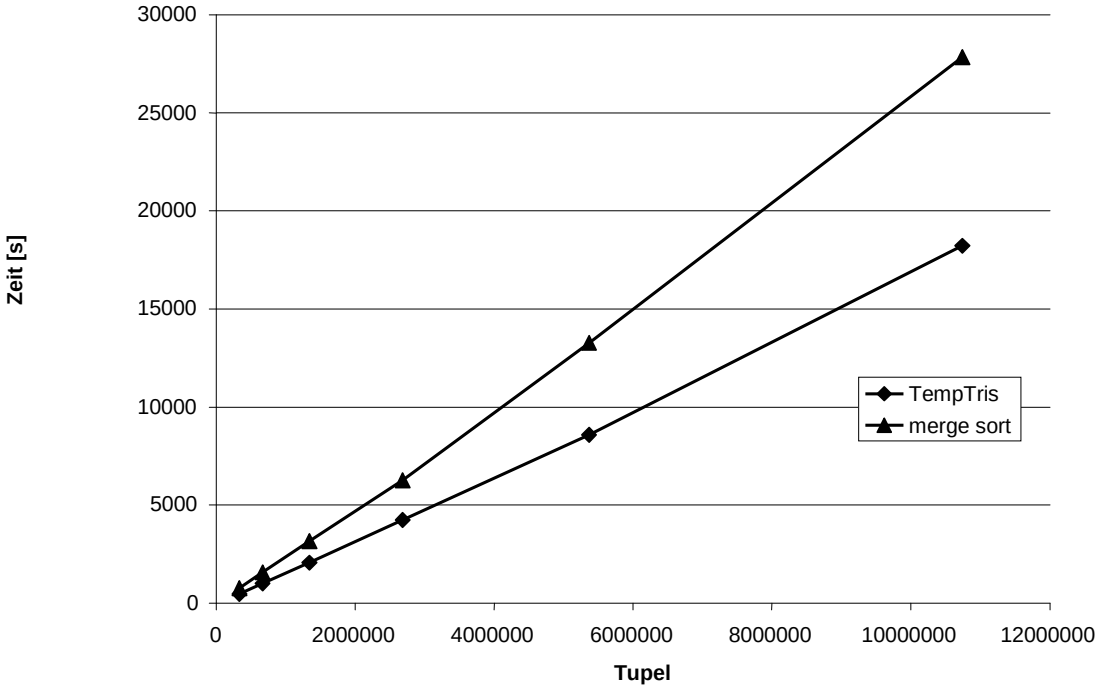


Abbildung 7-23: Ladeleistung

Für die Messung wurde jeder Datenwürfel zum einen mit dem TempTris-Algorithmus und zum anderen durch externes Sortieren aufgebaut. Bei der Implementierung des externen Sortierens wird nicht das Verfahren „Ersetzen und Auswahl“ (replacement selection) eingesetzt [EstW92, Gra93] (siehe Kapitel 4.2). Dies hat jedoch keinen Einfluss auf die Messungen, da der Cache *M*, der vom externen Sortieren und vom TempTris-Algorithmus genutzt wird, maximal *P*/2 ist. Hierbei ist *P* die Größe der Eingabedaten. Des Weiteren ist *P* kleiner als $M \cdot m$, so dass externes Sortieren eine lineare Komplexität hat. Die Seitenauslastung soll mindestens 80% sein, so dass der Skalierungsfaktor *u* für den TempTris-Algorithmus auf 3 gesetzt wird.

Abbildung 7-23 zeigt die Messungen für die Erzeugung von 6 UB-Bäumen auf unterschiedlichen Datenmengen. Jeder UB-Baum wurde sowohl mit dem TempTris-Algorithmus als auch mit dem externen Sortieren aufgebaut. Der TempTris-Algorithmus ist dem externen Sortieren überlegen, da er ca. die Hälfte an E/A-Operationen benötigt. Da die CPU-Kosten beim TempTris-Algorithmus größer als beim externen Sortieren sind, kann die Ladeleistung durch den TempTris-Algorithmus nicht um 100% gesteigert werden. Dies bestätigt somit das theoretische Modell aus Kapitel 7.3. Da der logarithmische Faktor aus Gl: 7-31 für das externe Sortieren 1 ist, müssen die E/A-Kosten linear bezüglich *P* sein. Dies wird durch Abbildung 7-23 bestätigt.

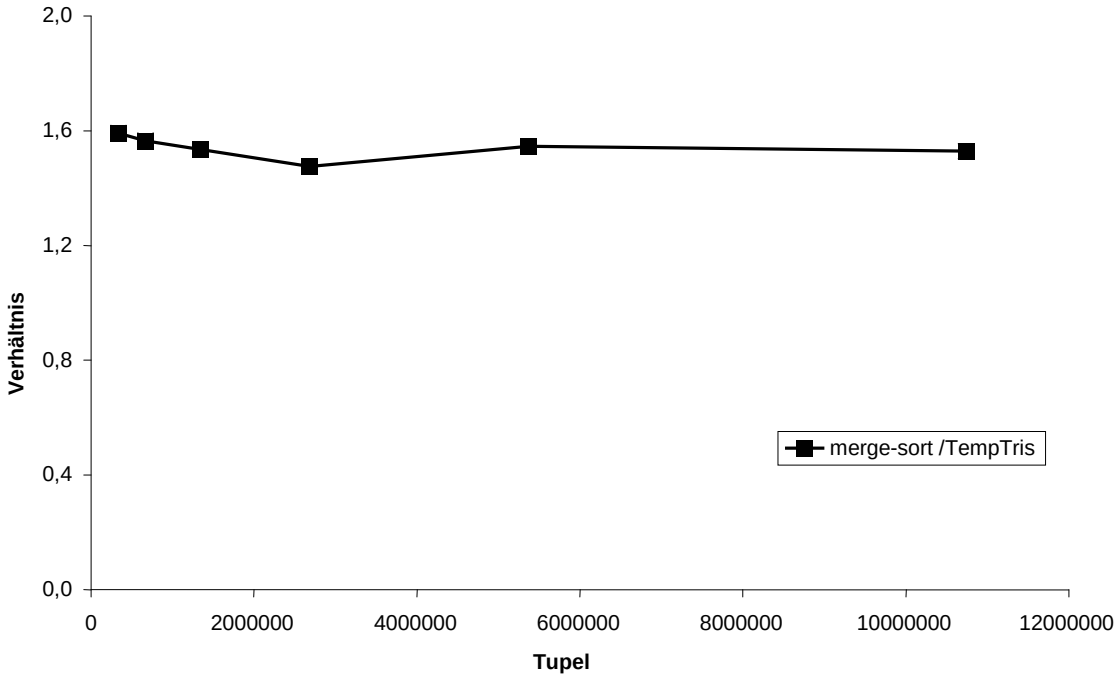


Abbildung 7-24: Verhältnis externes Sortieren und TempTris-Algorithmus

Abbildung 7-24 zeigt das Verhältnis der Kosten zwischen externem Sortieren und dem TempTris-Algorithmus. Mit dem TempTris-Algorithmus kann bei dieser Messumgebung ein durchschnittlicher Beschleunigungsfaktor von 1,5 erreicht werden. Hierbei sind die

CPU- und E/A-Kosten[17] kumuliert. Dies zeigt, dass die CPU-Kosten beim TempTris-Algorithmus größer sind als beim externen Sortieren. Dies ist in der Cacheverwaltung begründet. Bei dieser Messung wurde ein Skalierungsfaktor $u = 4$ verwendet. Dies führt zu einer Seitenauslastung von ca. 82%, das dem theoretischen Wert entspricht (siehe Abbildung 7-25).

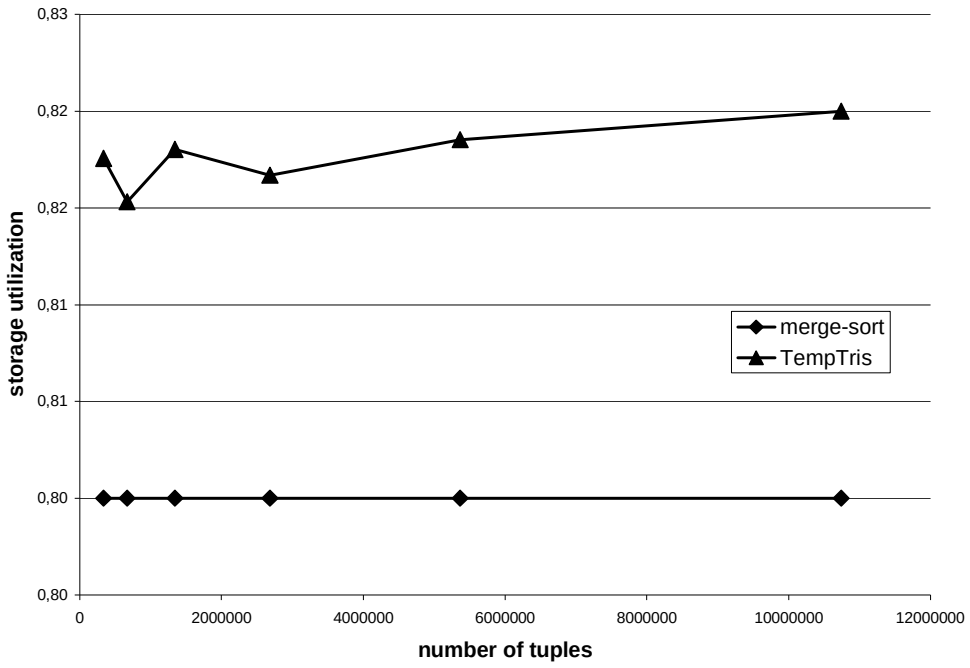


Abbildung 7-25: Seitenauslastung

Wie in Abschnitt 7.3.3 theoretisch hergeleitet wurde, werden für die Erzeugung der mehrdimensionalen Partitionierung des UB-Baums durch den TempTris-Algorithmus lediglich $\sqrt[d]{P^{d-1}}$ Seiten temporärer Speicher (Cache) benötigt. Hierbei bezeichnet *d* die Anzahl der Dimensionen.

[17] Es werden hier die Kosten der E/A-Operationen betrachtet.

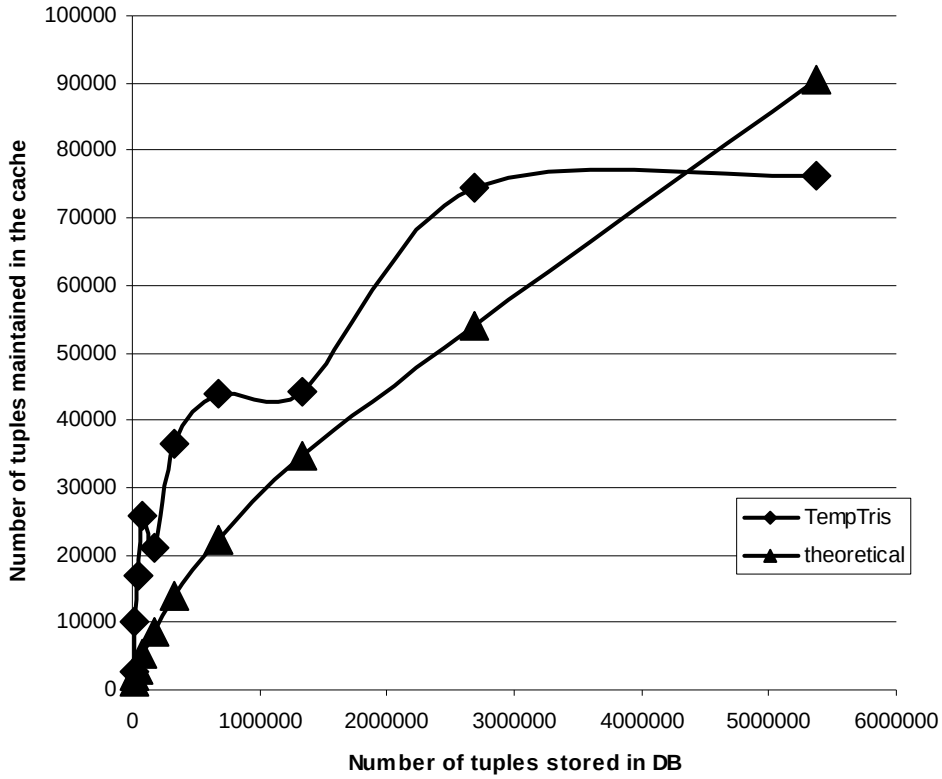


Abbildung 7-26: Cachegröße

Abbildung 7-26 zeigt die theoretische und die gemessene Cachegröße für die in Tabelle 7-4 beschriebene Datenmengen. Für Datenmengen, die kleiner als 300000 sind, ist die Cachegröße größer als die theoretische Cachegröße. Der Grund liegt im verwendeten unoptimierten Splitalgorithmus, der Fransen erzeugt (siehe [FriM97, Mar99]). Dadurch entstehen Regionen, die Bereiche abdecken, die weit in den dynamischen Bereich reichen und somit lange im Cache gehalten werden müssen. Für die 5242880 Datensätze liegt der der Graph unter dem theoretischen Verlauf .Der Grund liegt darin, dass die Formel von der idealisierten gleichverteilten Partitionierung ausgeht (siehe Definition 4-8), in der alle Dimensionen die gleiche Teilungstiefe besitzen. Dies ist in diesem Beispiel nicht der Fall. Die Sortierdimension hat bereits einen Split mehr, so dass die Scheibenanzahl sich verdoppelt und der Speicherbedarf sich verringert.

Um den Algorithmus auch unter Realbedingungen zu testen wurde das Data-Warehouse für die GFK aufgebaut. Hierbei werden die Daten auf 2-Monatsperioden voraggregiert abgespeichert. Die Daten werden in einem 3-dimensionalen Cube mit den Dimension *Produkt*, *Segment* und *Periode* verwaltet.

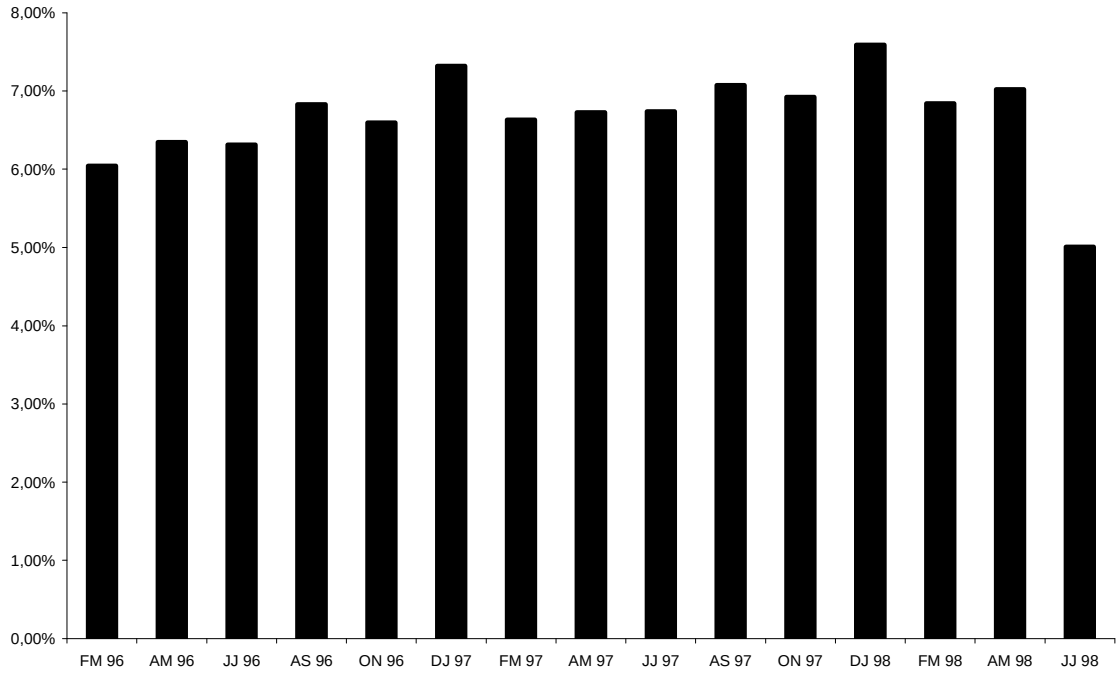


Abbildung 7-27: Datenverteilung Data-Warehouse GfK

Die Faktentabelle des Data-Warehouse der GfK besteht aus 43 Millionen Tupeln. Die Dimension *Periode* besteht auf der Konsolidierungsstufe 0 (siehe Kapitel 1.2) aus 15 2-Monatsperioden. Die Datenverteilung bezüglich der Periodendimension ist in Abbildung 7-27 aufgetragen. Die Größe eines Tupels beträgt 56 Byte. Für die Messung wurde nur die Faktentabelle berücksichtigt, da es sich hier um die größte Tabelle handelt und auf diese Weise die Leistungssteigerung am eindrucksvollsten dargestellt werden kann.

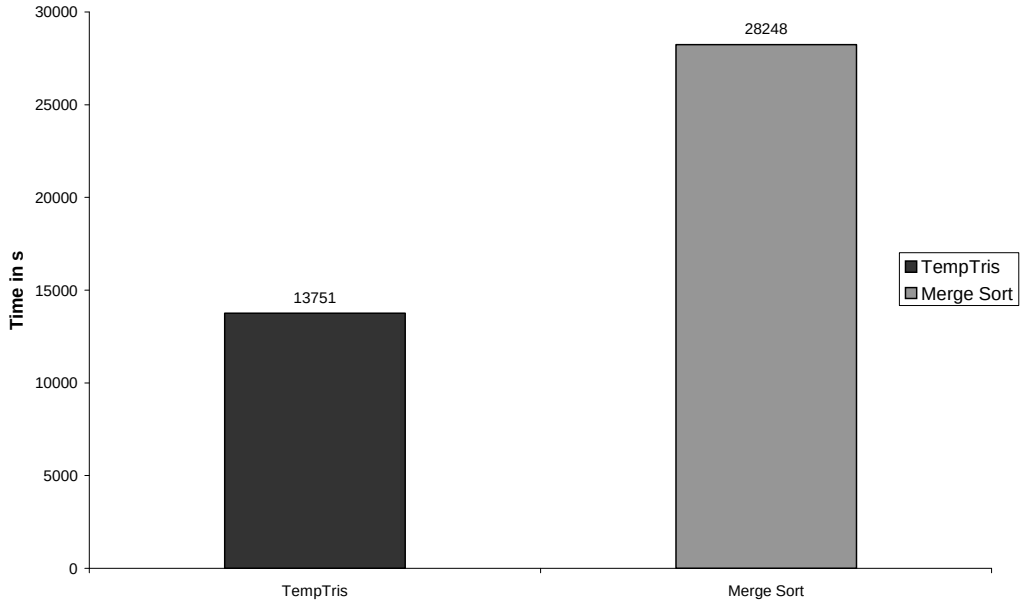


Abbildung 7-28: Ladezeiten des GfK DW

Die Seitengröße beträgt 2KB. Wegen der zusätzlichen internen Seitenverwaltung des Datenbanksystems können jedoch nur 31 statt 36 Tupel auf einer Seite gespeichert werden. Der Skalierungsfaktor wurde auf $u = 4$ festgelegt. Die Datenbank hat somit eine Größe von ca. 3GB.

Die Abbildung 7-28 zeigt die Ladezeit der Faktentabelle des GfK DW sowohl für den TempTris-Algorithmus als auch für das externe Sortieren. Der TempTris-Algorithmus ist dem externen Sortieren deutlich überlegen. Beide Algorithmen erzeugen den mehrdimensionalen Index in linearer Zeit. Die Seitenauslastung wurde beim Merge-Sort-Algorithmus auf 80% festgelegt. Der TempTris-Algorithmus erzeugt bei einem Skalierungsfaktor von 4 bei dieser Datenverteilung eine Seitenauslastung von bereits 87%. Somit muss der Sort-Merge-Algorthmus 8,7% mehr E/A-Operation ausführen.

Für diese Datenverteilung erreicht der TempTris-Algorithmus somit einen Beschleunigungsfaktor von 2,05. Dies entspricht dem erwarteten Wert aus dem theoretischen Modell.

7.5 Zusammenfassung

In diesem Kapitel wurde der TempTris-Algorithmus vorgestellt. Es handelt sich hierbei um eine effiziente Methode, die eine mehr-dimensionale Partitionierung aus einer sortierten Folge von Daten erzeugt. Für das Massenladen eines DW führt diese neue Technik zu einem erheblichen Leistungsgewinn gegenüber dem externen Sortieren. Sogar im linearen Bereich des externen Sortierens können die E/A-Kosten bezogen auf die Datenseiten um 50% reduziert werden. Der größte Vorteil des TempTris-Algorithmus besteht darin, dass die mehrdimensionale Partitionierung ohne Auslagerung von Zwischenergebnissen auf den Sekundärspeicher, wie es beim externen Sortieren der Fall ist, auskommt. Hierfür wird ein moderater zusätzlicher Cache benötigt.

Es wurde der allgemeine Algorithmus beschrieben und für die mehrdimensionale Partitionierung des UB-Baums realisiert. Hier wurde speziell auf die Cacheverwaltung eingegangen, die Regionen nach der Tetris- und Z-Ordnung verwaltet.

Basierend auf dem Kostenmodell aus Abschnitt 7.3 wurde der TempTris-Algorithmus analysiert. Die Analyse ergibt eine Leistungssteigerung gegenüber dem herkömmlichen externen Sortieren bei den E/A-Kosten um den Faktor 2. Die Cachegröße, die der TempTris-Algorithmus benötigt, ist eine Wurzelfunktion der Eingabedaten. Die Cachegröße, die vom externen Sortieren beim einmaligen Lesen der Daten benötigt wird, entspricht der Größe der Eingabedaten. Das theoretische Modell wurde durch Messungen des TempTris und des externen Sortierens auf künstlichen und echten Daten bestätigt.

Der TempTris-Algorithmus ist der inverse Operator des Tetris-Algorithmus, so dass die Ergebnisse zum Teil aus Kapitel 4und 5, wie z.B. die Cachegröße, übernommen werden können.

As the births of living creatures at first are ill-shapen, so are all innovations, which are the births of time.

Francis Bacon (1561–1626)

8 Prototypimplementierung

Im Rahmen des Projekts Mistral bei FORWISS wurde die UB-Library entwickelt, die die Funktionalität des UB-Baums auf ein relationales Datenbanksystem abbildet. Die Grundidee besteht darin, den B-Baum eines kommerziellen Datenbanksystems, wie z.B. TransBase von TransAction, der SQL-Server von Microsoft, 9i von Oracle oder DB2 von IBM, einzusetzen. Der Autor dieser Arbeit, Mitglied der Mistral Forschungsgruppe, hat den TempTris-, UBG- und Tetris-Algorithmus, die in dieser Arbeit vorgestellt wurden, selbst entwickelt und implementiert bzw. deren Implementierung in Form von Diplomarbeiten [LanZ99], [LueZ99], [HickZ00], Studienarbeit [KleZ00] und Fortgeschrittenen-Praktika [PelZ00], [HicZ00] betreut. Die Leistungsmessungen wurden mit Hilfe der UB-Bibliothek, einer C-Bibliothek, durchgeführt. In den folgenden zwei Abschnitten werden die Architektur und die Erweiterungen, die für diese Arbeit vorgenommen wurden, beschrieben. Im letzten Abschnitt wird kurz auf die Integration des SRQ-Algorithmus ins kommerzielle Datenbanksystem TransBase eingegangen. Die Beschreibung der Integration des Bereichsfrage-Algorithmus ist in [RamMF*00], [Ram02] zu finden. Basis für die Entwicklung bildete die UB-Bibliothek, die in [Mar99] ausführlich beschrieben wird. Hier wird lediglich die grundsätzliche Architektur vorgestellt

8.1 Architektur der UB-Bibliothek

Die Architektur der UB-Bibliothek besteht im Wesentlichen aus drei Schichten. Die *Datenbankschicht*, die *UB-Kernschicht* und die *UB-Operatorschicht*.

Ein Anwendungsprogramm, wie z.B. *mibm* (Mistral-Benchmark), *create* (erzeugt einen UB-Baum) und *swApp* (erzeugt einen UB-Baum mit Hilfe des TempTris-Alogrithmus), greift über die UB-API und Operator-Schnittstelle auf die UB-Technologie zu. Die UB-Kernschicht sowie die Datenbankschicht sind transparent für das Anwendungsprogramm. In der Operatorschicht werden Funktionen zur Erzeugung und zum Löschen eines UB-Baums sowie die Bereichsanfrage-Algorithmen auf den UB-Baum bereit gestellt.

Die UB-Algorithmen arbeiten auf Z-Adressen oder T-Adressen, die auf verschiedenen Datentypen angewendet werden können. Die Adressberechnung und die Abbildung auf das jeweilige Datenbanksystem sind in der UB-Kernschicht enthalten.

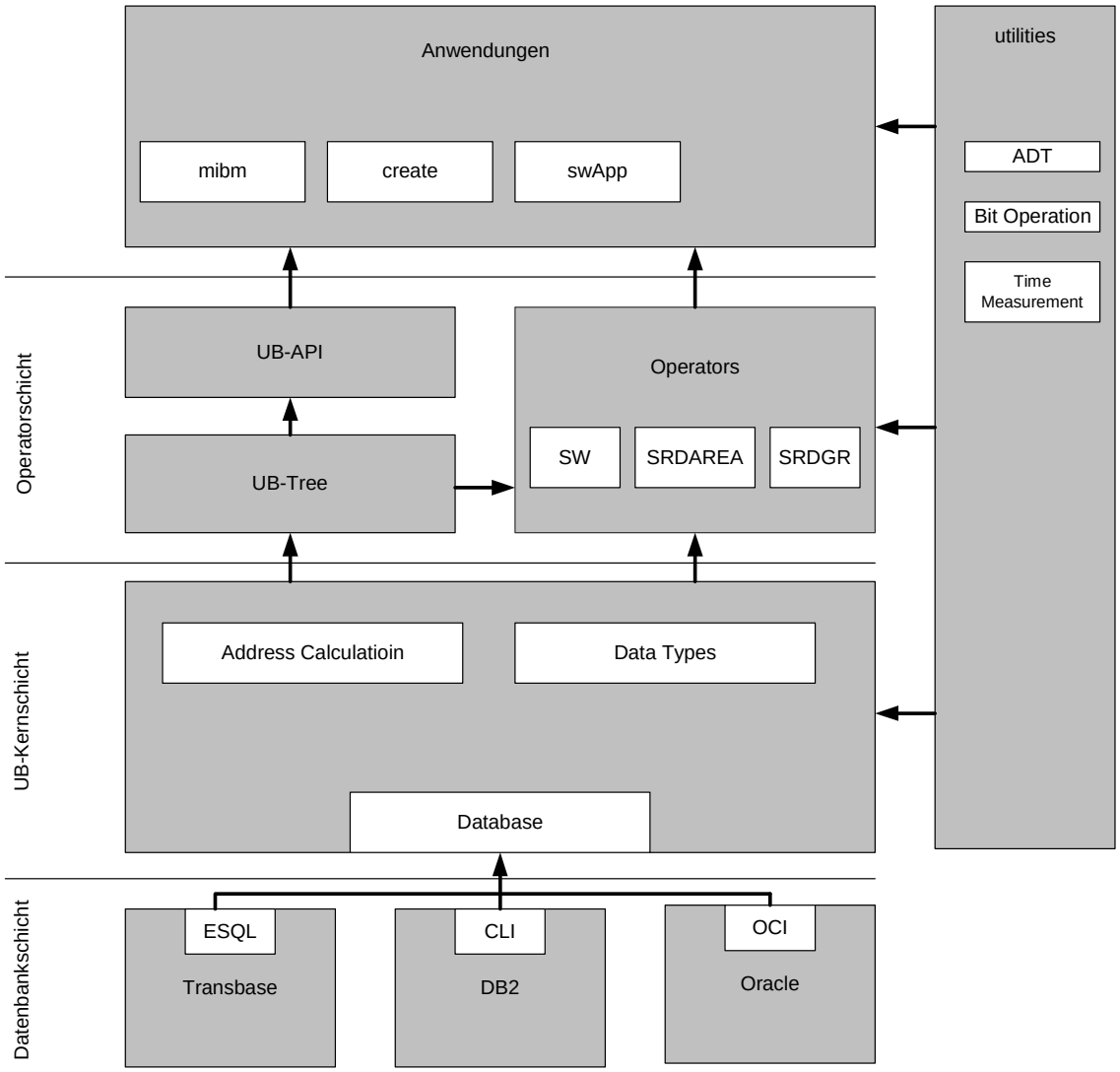


Abbildung 8-1: UB-Bibliothek

Das logische Konstrukt UB-Baum liegt physisch im relationalen Datenbanksystem als relationale Tabelle vor. Die Tabelle wird als *UB-Baum-Index-Tabelle* bezeichnet. Die *UB-Baum-Index-Tabelle* enthält für jede Seite genau ein Tupel, d.h. ein Tupel in der UB-Index-Tabelle entspricht einer Region im UB-Baum. Für die Prototypimplementierung wurde folgendes Datenbankschema verwendet:

Attribut	Domäne	Beschreibung
p	binchar	Obere Z-Adresse α der Z-Region $[\alpha:\beta]$, die zu dieser Seite $p = seit([\alpha:\beta])$ des UB-Baums gehört
cnt	integer	Anzahl der Tupel, die auf Seite *p* gespeichert sind
data	binchar	Inhalt der Seite

Tabelle 8-1: Schema der UB-Baum-Index-Tabelle in TransBase

Tabelle 8-2 zeigt z.B. die ersten 9 Tupel der UB-Baum-Index-Tabelle. Hierbei handelt es sich um eine 3-dimensionale Datenbank mit 2 KB großen Seiten und einer Tupelgröße von 220 Byte. Theoretisch müsste die Seitenkapazität κ für diese Datenbank 9 betragen. Für die interne Seitenverwaltung des Datenbanksystems wird zusätzlicher Speicherplatz benötigt. Dies führt in diesem Beispiel zu einer Seitenkapazität κ von 8 Tupeln.

p	cnt	data
0x00000000000040600	5	0x07db0000df5a000047…
0x00000000001030300	5	0x6e03000071610100e6 …
0x00000000002010200	7	0x68cd0700f93a0300c3 …
0x00000000002050000	7	0xe49d02001777050029 …
0x00000000002070000	5	0x12950300b820040009 …
0x00000000003020000	5	0x61910300a192070078 …
0x00000000003040400	5	0xa62f040044d70700ae …
0x00000000004010300	8	0x2acd05008076040041 …
0x00000000004030204	4	0x946603006c150100ac ...
…		

Tabelle 8-2: Inhalt der UB-Index-Tabelle

Die Datenbank TransBase clustert die Daten immer nach dem Primärschlüssel. Somit wird die UB-Index-Tabelle in TransBase als IOT mit der Z-Adresse als Primärschlüssel implementiert. Da UB-Baum-Seiten physische Datenbankseiten darstellen, muss für ein Zugriff auf eine Z-Adresse ein wahlfreier Zugriff durchgeführt werden. Um den Zugriff auf Z-Adressen und somit auch den Bereichs- und Tetris-Algorithmus zu beschleunigen wird in der Prototypimplementierung in TransBase eine weitere Tabelle, die *Sekundär-Index-Tabelle*, bereitgestellt. Die Sekundär-Index-Tabelle speichert nur die Z-Adressen der jeweiligen UB-Index-Tabelle. Hierdurch wird ein effizienter Zugriff auf Z-Adressbereiche erreicht.

8.2 ADTs

Für die Entwicklung des Tetris-, TempTris- und UBG-Algorithmus wurden abstrakte Datentypen (ADT) definiert, die die geforderten Funktionalitäten bereitstellen. Hier ist z.B. die Cacheverwaltung der Tupel des Tetris-Algorithmus oder die der Äquivalenzklassen der UBG-Algorithmus zu nennen. Die Verwaltung von Φ, der noch nicht verarbeitete Raum, stellt besondere Ansprüche an die Datenstrukturen. Die Implementierung der einzelnen ADTs wurde in dem Modul ADT zusammengefasst.

Für die Cachverwaltung wurde im Tetris-Algorithmus ein Heap eingesetzt. Der Z-Index und der T-Index wurden durch einen AVL-Baum realisiert. Eine genaue Beschreibung und Analyse des Verhaltens dieser Datenstrukturen ist in [Knu98v3] zu finden.

Die Verwaltung von Φ wurde mit Hilfe der Datenstruktur Treap [AraS89],[OttW96] implementiert (siehe 5.9).

8.3 Integration in ein kommerzielles Datenbanksystem

Im ESPRIT Projekt MDA, gefördert durch die Europäische Union, wurde der UB-Baum in das kommerzielle DBMS *TransBase* von *TransAction* Software GmbH, München, integriert. Eine genaue Beschreibung ist in [RamMF*00], [Ram02] zu finden. An dieser Stelle wird primär auf den SRQ-Algorithmus [Bay97b] eingegangen, den der Autor dieser Arbeit als Prototyp integriert hat.

8.3.1 Änderungen und TransBase

TransBase ist ein auf dem SQL-92 Standard basierendes relationales Datenbanksystem. Es ist als Client-Server-Architektur implementiert und benutzt einen B*-Baum als clusternden Index. Da das DBMS einen B^*-Baum bereitstellt, mussten im Wesentlichen folgende Erweiterungen am DBMS vorgenommen werden:

- Erweiterung der Data-Definition-Language (DDL)
- Erweiterung des Anfrage-Optimierers und des Anfrageprozessor um das Kostenmodell des UB-Baums
- Integration der UB-Operatoren, wie z.B. des SRQ-Algorithmus

Die einzelnen Änderungen an den Modulen von TransBase sind in Abbildung 8-2 aufgeführt. Eindrucksvoll ist hier besonders, dass die Änderungen zum einen nicht alle Module betreffen und die Änderungen im Wesentlichen sehr gering sind. Die Änderung an der DDL besteht aus einer Erweiterung des Parsers zur Erzeugung des UB-Indexes sowie aus einer Erweiterung der Katalogeinträge im Katalogmanager. Um den mehrdimensionalen UB-Baumindex bei der Anfrageverarbeitung optimal auszunutzen, ist die Anpassung des Anfrage-Optimierers eine wesentliche Aufgabe. Diese Problematik wird in [Ram02] behandelt.

Abbildung 8-2: Architektur von TransBase

Als letzter Schritt ist die Erweiterung des Zugriffspfad-Managers zu nennen. In diesem Modul wurden der Bereichsanfrage-Algorithmus und der SRQ-Algorithmus integriert. Leistungsmessungen für den Bereichsanfrage-Algorithmus sind in [RamMF*00],[Ram02] zu finden.

There are no such things as applied sciences, only applications of science.

Louis Pasteur (1822–1895)

9 Data-Warehousing

In diesem Kapitel wird formal ein DW eingeführt. Hierbei liegt der wesentliche Schwerpunkt auf der physischen Modellierung des DW-Systems. Auf dieser Modellierung werden wir anhand des GfK-DWs verschiedene Techniken für den Cube-Operator vorstellen, die auf der UB-Technologie basieren.

9.1 DW-Modell

In dieser Arbeit wird nur die Modellierung eines DW betrachtet, das auf relationalen Datenbanksystemen basiert. Für einen guten allgemeinen Überblick wird auf [CodCS93],[ChaD97],[Kim96], [Kur99], [GünB00] verwiesen.

9.1.1 Das Star-Schema

DW-Anwendungen werden üblicherweise in Form eines mehrdimensionalen Modells modelliert. Die Fakten (Kennzahlen, numerische Daten) vom Typ F_1, F_2, F_3,..., F_m , (z.B. Verkäufe, Kosten, etc.), auf denen das Augenmerk der Analyse liegt, werden bezüglich verschiedener Dimensionen organisiert. Eine Dimension besteht aus kategorisierenden Daten (z.B. Containergröße eines Produkts, Bayern für die geographische Kategorisierung von Kunden), die den Kontext der Kennzahlen bestimmen. Sie beschreiben die mögliche Sicht des Anwenders zu den assoziierten betriebswirtschaftlichen Kennzahlen.

Definition 9-1: Dimension

Eine Dimension[18] $D_{\text{<Name>}}$ ist eine endliche Menge von atomaren Elementen.

Eine Menge von Dimensionen $\{D_1, D_2, D_3, \ldots, D_n\}$ definiert einen n-dimensionalen Raum. Jeder Punkt kann durch die Koordinaten $(d_1, d_2, \ldots, d_n)$ der einzelnen Dimensionen bestimmt werden.

[18] Als Symbol einer Dimension wird der Buchstabe *D* verwendet. Dies entspricht dem Symbol für Domäne aus Kapitel 3. Sowohl eine Domäne also auch eine Dimension bestehen aus atomaren Elementen. Eine Dimension hat die zusätzliche Eigenschaft, dass sie Bestandteil des Schlüssels im Datenwürfel W ist. Falls es aus dem Zusammenhang nicht ersichtlich ist, ob es sich um eine Domäne oder Dimension handelt, wird dies explizit genannt.

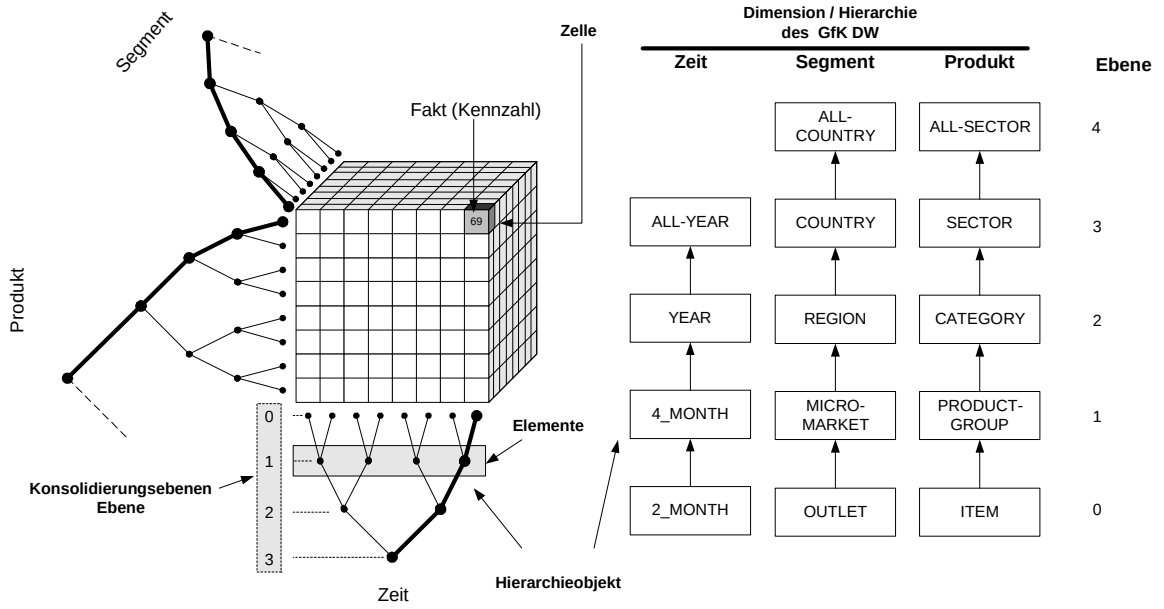


Abbildung 9-1: GfK DW

Den Punkt des Raumes bezeichnet man als *Zelle*, die Koordinaten als *Zelladresse*. Die Dimensionen bilden zusammen mit den Fakten den Datenwürfel (Datacube).

Definition 9-2:Datenwürfel [Bay02]

Ein Datenwürfel *W* mit *n* Dimensionen $D_1, D_2, D_3, \ldots, D_n$ und *m* Fakten $F_1, F_2, F_3, \ldots, F_m$ ist das kartesische Produkt der Dimension, verbunden mit den Fakten $(f_1, f_2, f_3, \ldots, f_m)$

$$W = \{(d_1, d_2, \ldots, d_n, f_1, f_2, f_3, \ldots, f_m) \mid (\forall i \in \{1,\ldots,n\}: d_i \in D_i) \wedge$$

$$(\forall j \in \{1, \ldots, m\}: f_j \in F_j) \wedge (d_1, d_2, \ldots, d_n) \text{ is key } \}$$

Das *Star-Schema* ist das Standardmodell in der relationalen Modellierung [Kim96], [GünB00]. Hierbei werden die Fakten in einer großen Faktentabelle organisiert. Der Schlüssel eines Tupels in der Faktentabelle besteht aus der Konkatenation der einzelnen Fremdschlüssel, die die jeweiligen Dimensionsschlüssel $d_1, \ldots, d_n$ referenzieren.

Abbildung 9-2 zeigt das Star-Schema der GfK. Der Schlüssel *K* der Zeitdimension etwa besteht aus der Konkatenation der Attribute *YEAR_ID, MONTH4_PERIOD_ID* und *MONTH2_PERIOD_ID*. Der Schlüssel stellt somit ein atomares Element nach Definition 9-1 dar. Da die Kombination der einzelnen Dimensionen eine Zelle im CUBE referenzieren, besteht der Schlüssel der Faktentabelle aus der Konkatenation der Dimensionsschlüssel. Bei dem GfK-Schema besteht der Schlüssel der Faktentabelle aus insgesamt 11 Fremdschlüsseln, die durch das Präfix (FK) gekennzeichnet sind. Die Modellierung besticht durch ihre Einfachheit.

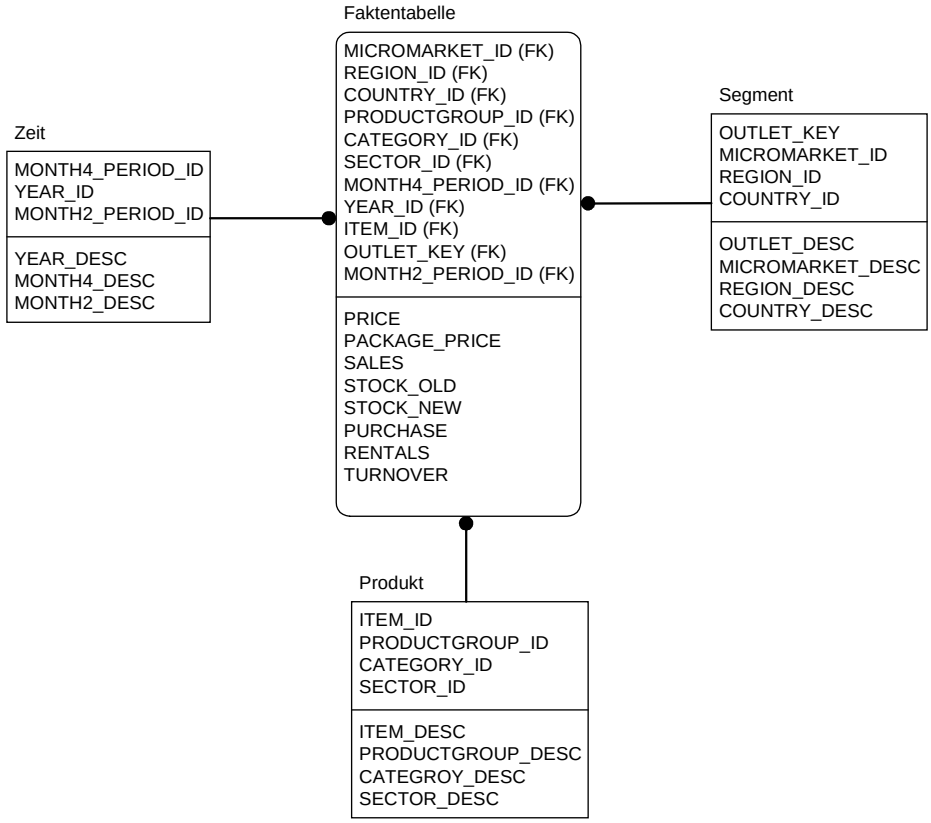


Abbildung 9-2: GfK Star-Schema[19]

Die Modellierung zeigt jedoch bereits hier ihre wesentlichen Schwächen. Geht man davon aus, dass die Dimensionen sowie die Faktentabelle als IOT organisiert sind, stellt man fest, dass beim GfK Star-Schema Bereiche auf den Dimensionstabellen effizient berechnet werden können. Es handelt sich um einen 1-dimensionalen Bereich (siehe [Mar99], [MarZB99]). Da die physikalische Organisation durch den Schlüssel bestimmt wird (siehe Kapitel 3, [Mar99]), kann nur die Selektivität des ersten Fremdschlüssels *MIRCOMARKET_ID (FK)* ausgenutzt werden. Die Selektivität auf den übrigen Fremdschlüsseln führen zu Punktanfragen.

Typischerweise besitzen Attribute innerhalb einer Dimensionstabelle funktionale Abhängigkeiten [Leh98]. Innerhalb der Produktdimension bestimmt z.B. die Zugehörigkeit eines *ITEM* zu einer *PRODUCTGROUP* auch die Zugehörigkeit zu einer *CATEGORY* und einem *SECTOR* (siehe Abbildung 9-2: GfK Star-Schema). Dies führt zu den bekannten Anomalien, wie *Updateanomalie*, *Einfügeanomalie* und *Löschanomalie* [EmE96]. Nach dem Ladeprozess ist die wesentliche Operation im DW die Lese-Operation. Nachteil des Star-Schemas ist somit die redundante Datenhaltung in den Dimensionstabellen.

[19] IDEF1X Notation

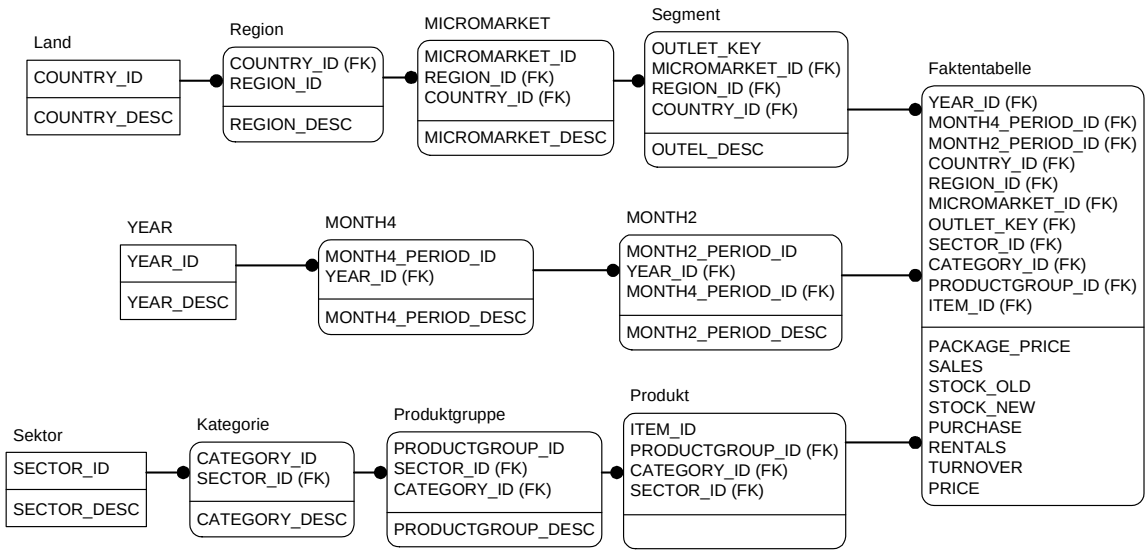


Abbildung 9-3: GfK-Snowflake-Schema[20]

Eine Normalisierung der Dimensionstabellen führt zum *Snowflake-Schema* (siehe Abbildung 9-3). Der wesentliche Vorteil liegt somit in der Vermeidung der Redundanz in den Dimensionstabellen. Allgemein kommt es durch die Normalisierung zu zusätzlichen Verbundoperatoren, um eine mehrdimensionale Bereichsanfrage auf die Faktentabelle auszuführen. Das führt zu einer Verschlechterung der Antwortzeiten.

9.1.2 Snowflake-Schema mit UB-Baum

Dieses Kapitel stellt eine Modellierungstechnik des Snowflake-Schemas, basierend auf dem UB-Baum vor. Hierzu erweitern wir Definition 9-1 wie folgt:

Definition 9-3: Dimension

Eine Dimension $D_{<\text{Name}>}$ ist eine endliche Menge von atomaren Elementen, auf der eine Ordnung $\leq$ definiert ist.

Existiert auf einer Dimension *D* keine natürliche Ordnung, muss eine beliebige Ordnung festgelegt werden.

Ein wichtiges Konzept im Bereich OLAP ist das Konstrukt der *Dimensionshierarchie*. Hierarchien werden benutzt, um Dimensionen zu strukturieren. Üblicherweise können die Daten innerhalb einer Dimension in *Hierarchieobjekte* zusammengefasst werden, die untereinander in einer semantischen Beziehung stehen. Sie repräsentieren unterschiedliche Konsolidierungsebenen der assoziierten betriebswirtschaftlichen Kennzahlen. *YEAR_ALL*, *YEAR*, *4_MONTH_PERIOD* und *2_MONTH_PERIOD* sind die Hierarchieobjekte der Zeit-Dimension des GfK-Schemas (siehe Abbildung 9-1).

[20] IDEF1X Notation

Um die Dimensionshierarchie und das Hierarchieobjekt zu definieren, führen wir zunächst den Begriff der ordnungserhaltenden Zerlegung:

Definition 9-4: *Ordnungserhaltende Zerlegung*

Eine ordnungserhaltende Zerlegung $\underline{\boldsymbol{H}}$ ist eine Partitionierung einer Menge D in Teilmengen $\{A_1,A_2 , ... , A_n\}$, wobei die Mengen untereinander kein gemeinsames Element besitzen und die Teilmengen eine totale Ordnung

$$A_1 < A_2 < A_3 < \ldots < A_n$$

bilden. Hierbei bezeichnen wir eine Menge $A < B$ genau dann, wenn gilt:

$$A < B \Rightarrow \forall a \in A, b \in B: a < b$$

Die Vereinigung der einzelnen Teilmengen $A_1,A_2, ..., A_n$ ist wieder die Menge D. Formal gilt somit:

$$\bigcup_{i=1}^{n} A_i = D$$

Basierend auf Definition 9-4 kann nun der Begriff Hierarchieobjekt definiert werden:

Definition 9-5: Hierarchieobjekt

Ein *Hierarchieobjekt* $\underline{\boldsymbol{L}}$ über eine Dimension D ist eine ordnungserhaltende Zerlegung von D.

Definition 9-6: Ableitbarkeit

Seien $\underline{\boldsymbol{A}}$ und $\underline{\boldsymbol{J}}$ zwei *Zerlegungen* (*Partitionierungen*) auf einem gemeinsamen Bereich von Elementen der Menge B, dann ist $\underline{\boldsymbol{A}}$ aus $\underline{\boldsymbol{J}}$ ableitbar ($\boldsymbol{J} \rightarrow \boldsymbol{A}$) genau dann, wenn gilt:

- $\bigcup A \subseteq \bigcup J$ mit $A \in \underline{\boldsymbol{A}}$ und $J \in \underline{\boldsymbol{J}}$
- $\forall J \, \exists A \, (((A \in \underline{\boldsymbol{A}}) \wedge (J \in \underline{\boldsymbol{J}}) \wedge (A \cap J \neq \emptyset)) \Rightarrow (J \subseteq A))$

Der Begriff der *Dimensionshierarchie* ist dann wie folgt definiert:

Definition 9-7: Dimensionshierarchie, Dimensionsebene

Eine Folge von Hierarchieobjekten $\underline{\boldsymbol{H}} = \langle \underline{\boldsymbol{L}}^0, \underline{\boldsymbol{L}}^1, \underline{\boldsymbol{L}}^2, \ldots, \underline{\boldsymbol{L}}^{h-1} \rangle$ über eine Dimension D ist eine *Dimensionshierarchie* genau dann, wenn gilt:

1. $\underline{\boldsymbol{L}}^i$ ist Hierarchieobjekt von D
2. $\underline{\boldsymbol{L}}^0 \rightarrow \underline{\boldsymbol{L}}^1 \rightarrow \underline{\boldsymbol{L}}^2 \rightarrow^{\cdots} \rightarrow \underline{\boldsymbol{L}}^{h-1}$; $\rightarrow$ bezeichnet die Ableitbarkeit der Zerlegung.
3. $|\underline{\boldsymbol{L}}^{h-1}| = 1$.

i bezeichnet die Hierarchieebene.

Man beachte, dass Bedingung 2 eine totale Ordnung ≤ auf jede Dimension *D* voraussetzt. Diese Bedingung ist notwendig, um Punktanfragen auf höheren Konsolidierungsebenen einer Hierarchie auf eine eindeutige Bereichsanfrage auf Ebene 0 abzubilden. Diese Eigenschaft erzeugt einen *n*-dimensionalen Teilraum *Q* auf dem Basisraum Ω, der effizient durch den UB-Baum-Bereichsanfragealgorithmus abgearbeitet werden kann [Mar99] (siehe Kapitel 4.4.1).

Basierend auf dem Konstrukt der *Dimensionshierarchie* und des *Hierarchieobjekts* kann man die *Mitgliedsmenge* (*member sets*) bzw. das *Mitglied* (*member*) wie folgt definieren:

Definition 9-8: Mitgliedsmenge, Mitglied

Eine *Mitgliedsmenge* (*member_set*) *A* ist eine Teilmenge von *D*, die durch das Hierarchieobjekt ***L*** bestimmt wird:

$$\text{member\_set}_{D,\underline{\boldsymbol{L}}} = (A \in \underline{\boldsymbol{L}}(D))$$

Die Teilmengen $A_1, A_2, \ldots, A_n$ werden als Mitglieder (member) des Hierarchieobjekts ***L*** bezeichnet.

Um zu betonen, dass ein Element *A* ein Element eines Hierarchieobjekts ist, bezeichnen wir es als *Mitglied* (member) eines Hierarchieobjekts. Wenn der Mengenaspekt betont werden soll, wird *E* als *Mitgliedsmenge*, die aus atomaren Elementen der Dimension *D* besteht und die ein zusammenhängendes Intervall auf *D* bildet, bezeichnet.

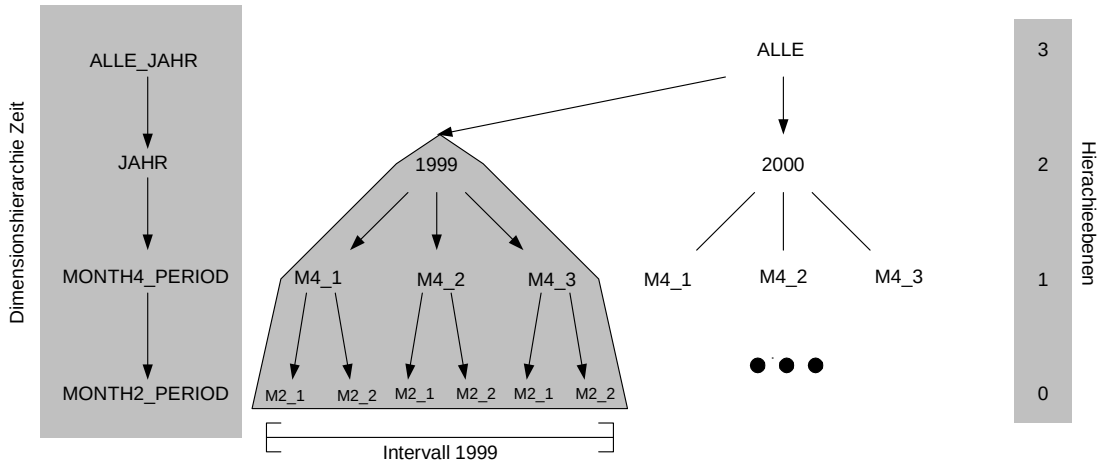


Abbildung 9-4: GfK Dimension Zeit

Beispiel 9-1: Hierarchie, Hierarchieebene, Mitglied, Mitgliedsmenge

Abbildung 9-4 zeigt die Zeit-Dimension des 3-dimensionalen DW der GfK (siehe Abbildung 9-1). Sie besteht aus den vier Hierarchieobjekten ALLE_JAHR, JAHR, MONTH4_PERIOD und MONTH2_PERIOD. Die Hierarchieebene 3 (ALLE_JAHR) besteht aus dem Mitglied ALLE. Auf dieser Konsolidierungsebene wird die Dimension zu einem Wert zusammengefasst. Das Hierarchieobjekt JAHR besteht in Abbildung 9-4 aus zwei Mitgliedern, 1999 und 2000. Auf Ebene 1 besteht die Zeitdimension aus drei Vier-Monatsperioden, die jeweils zwei Zwei-Monatsperioden überspannen.

Durch die hierarchische ordnungserhaltende Strukturierung der Dimension kann eine Restriktion auf höherer Hierarchieebene auf einen Bereich der Hierarchieebene 0 eindeutig abgebildet werden. So kann die SQL-Anfrage,

```
SELECT f.YEARD_ID, SUM(f.PRICE)
FROM  FAKTENTABELLE f, ZEIT z
WHERE f.YEAR_ID =z.YEAR_ID AND
      z.YEAR_DESC = '1999'
GROUP BY f.YEAR_ID;
```

die eine Punktrestriktion in der Zeitdimension auf das Jahr 1999 hat, auf das Intervall wie folgt abgebildet werden:

```
SELECT f.YEAR_ID, SUM (f.PRICE)
FROM FAKATENTABELLE f, ZEIT z
WHERE z.MONTH2_PERIOD_DESC >= '1999.M4_1.M2_1' AND
       z.NONTH2_PERIOD_DESC <= '1999.M4_3.M2_2' AND
       z.MONTH2_PERIOD = f.MONTH2_PERIOD
GROUP BY f.YEAR_ID;
```

In [MarRB99] wird eine Methode vorgestellt, die eine hierarchische, ordnungserhaltende Dimension auf künstliche Schlüssel, die so genannten Surrogate, abbildet. Hierbei wird für jedes Hierarchieobjekt die Kardinalität bestimmt und den einzelnen Mitgliedern des jeweiligen Hierarchieobjekts eine Ordinalzahl, die zwischen 0 und der Kardinalität des Hierarchieobjekts liegt, zugewiesen. Da das Surrogat als Binärwert dargestellt wird, ist die Stelligkeit pro Hierarchieobjekt wie folgt definiert:

$$\lceil \log_2 |\text{Hierarchieobjekt}| \rceil \qquad \text{Gl: 9-1}$$

Für die Zeitdimension ergibt sich folgende Stelligkeit:

Hierarchieobjekt	Kardinalität
YEAR	2
MONTH4_PERIOD	3
MONTH2_PERIOD	2

Tabelle 9-1: Stelligkeit der Surrogate

Wendet man die Surrogate auf die Zeitdimension an, erhält man die in Abbildung 9-5 dargestellte Struktur.

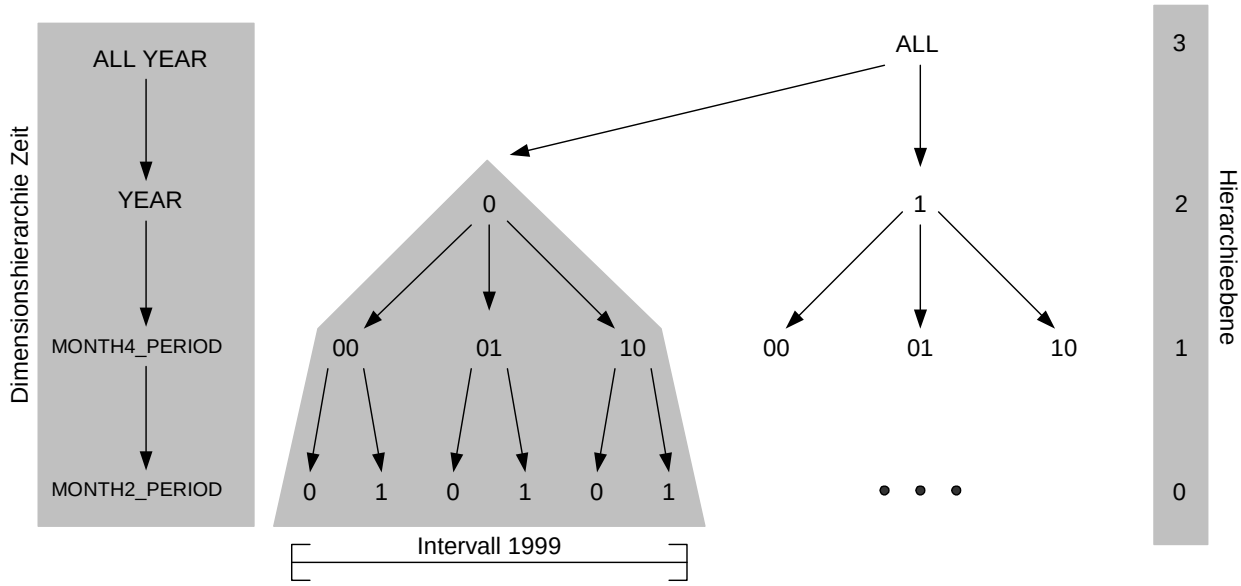


Abbildung 9-5: Dimension Zeit mit entsprechenden Surrogaten

Das Jahr 1999 wird auf das Intervall [0.00.0: 1.10.1] abgebildet.

9.2 CUBE-Operator

In diesem Kapitel werden verschiedene Möglichkeiten der Anfrageverarbeitung für den CUBE-Operator unter Ausnutzung des UB-Baums vorgestellt. Diese Techniken basieren auf Aggregationsgittern. Für die Berechnung des Aggregationsgitters wird der Tetris- mit dem TempTris- und UBG-Algorithmus [Zir00b], [ZirMB00a], [ZirMB01], [MarZB99] kombiniert, so dass die E/A-Kosten erheblich reduziert werden können.

Der CUBE-Operator [GraCB+97] im DW ist ein besonders geeignetes Anwendungsgebiet für Aggregationsgitter. Der CUBE-Operator berechnet $2^{|Attr|}$ Aggregate, wobei $|Attr|$ die Anzahl der Gruppierungsattribute darstellt. Abbildung 9-6 zeigt ein Beispiel einer CUBE-Anweisung auf das GfK DW.

```
SELECT f.MONTH2_PERIOD_ID, f.MONTH4_PERIOD_ID,f.YEAR_ID
       f.ITEM_ID, f.PRODUCTGROUP_ID, f.CATEGORY_ID,
       f.SECTOR_ID
       f.Outlet_ID, f.MICROMARKET_ID, f.REGION_ID,
       f.COUNTRY
       SUM(f.SALES) AS Sales
FROM FAKTENTABELLE f, ZEIT z, SEGMENT s
WHERE  z.YEAR_DESC > '1996' AND
       z.YEAR_DESC < '1999' AND
       z.YEAR_ID  = f.YEAR_ID AND
       s.COUNTRY_DESC = 'Germany' AND
       s.COUNTRY_ID = f.COUNTRY_ID
GROUP BY f.MONTH2_PERIOD_ID, f.MONTH4_PERIOD_ID,
       f.YEAR_ID,f.ITEM_ID, f.PRODUCTGROUP_ID,
       f.CATEGORY_ID, f.SECTOR_ID,
       f.Outlet_ID, f.MICROMARKET_ID, f.REGION_ID,
       f.COUNTRY  WITH CUBE
```

Abbildung 9-6: CUBE-Anweisung

Berechnet man den vollständigen Cube, ergeben sich für das obige Beispiel $2^{11} = 2048$ Gruppierungen. Hierbei sind jedoch einige Gruppierungen semantisch nicht sinnvoll. Berücksichtigt man noch die funktionalen Abhängigkeiten, die in den einzelnen Hierarchien existieren, reduziert sich die Anzahl der Gruppierungen erheblich. Für jedes Hierarchieobjekt einer Hierarchie wird eine Gruppierung durchgeführt. Da die einzelnen Hierarchien orthogonal zueinander stehen, ist die Anzahl der Gruppierungen der einzelnen Dimensionen untereinander multiplikativ.

Dimension	Anzahl der Gruppierungen
Zeit	4
Produkt	5
Segment	5

Abbildung 9-7: Anzahl der Gruppierung pro Dimension

Für das GfK-Schema ergeben sich somit

$$4 \cdot 5 \cdot 5 = 100$$

Gruppierungen.

Die SQL-Anfrage in Abbildung 9-6 besteht im Wesentlichen aus zwei Operatoren:

- mehr-dimensionaler Bereichsanfrage-Operator
- Gruppierungs-Operator

9.2.1 Mehrdimensionale Restriktionen

Um die mehr-dimensionale Bereichsanfrage effizient abzuarbeiten, wird die Faktentabelle als UB-Baum organisiert. Hierbei werden die Surrogatschlüssel der Dimension *Zeit*, *Produkt* und *Segment* Schlüssel des UB-Baums. Durch diese Organisation kann somit die Selektivität in allen drei Dimensionen vollständig ausgenutzt werden [Mar99], [ZirMB00a].

9.2.2 Aggregatsbildung und Duplikatsentfernung

Die Aggregatsbildung ist ein sehr wichtiges statistisches Konzept im Datenbank- und DW-Bereich. Durch die Aggregatsbildung können den Anwendern verschiedene Konsolidierungsebenen auf den Daten bereitgestellt werden. Die Idee der Aggregatsbildung ist somit, eine Menge von Datensätzen durch eine Kenngröße zu repräsentieren oder die Menge in Gruppen zu klassifizieren und der jeweiligen Gruppe eine Kenngröße zuzuweisen.

Die meisten Datenbanksysteme unterstützen statistische Funktionen, wie *Summe*, *Anzahl*, *Durchschnitt*, *Maximum*, *Minimum*, sowie *Standardabweichung*.

Aggregationsfunktion	Beschreibung
SUM([ALL | DISTINCT] expression)	Summe der numerischen Werte

AVG([ALL | DISTINCT] expression)	Durchschnitt der numerischen Werte
COUNT([ALL | DISTINCT] expression)	Anzahl der Werte im Ausdruck
COUNT(*)	Anzahl der ausgewählten Tupel
MAX(expression)	Maximaler Wert, der durch den Ausdruck spezifiziert wird
MIN(expression)	Minimaler Wert, der durch den Ausdruck spezifiziert wird
STDEV(expression)	Standardabweichung für alle Werte, die durch den Ausdruck spezifiziert werden
VAR**(**expression**)**	Varianz für alle Werte, die durch den Ausdruck spezifiziert werden

Tabelle 9-2: Aggregationsfunktionen des SQL-Server 2000

Nach [Klu82] lässt sich die relationale Aggregationsoperation für eine Menge *Agg* vorgegebener Aggregationsfunktionen *f* wie folgt definieren:

Definition 9-9: Aggregationsoperation

Gegeben sei eine Relation R mit der Attributmenge **sch**(R), eine Attributmenge X mit $X \subseteq$ **sch**(R) und eine Aggregationsfunktion $f \in Agg$. Dann ist die Aggregationsopertion $\alpha_{X,f}R$:

$$\alpha_{X,f} R = \left\{ \pi_X r \bullet s \middle| r \in R \wedge s = f \right\} \text{ mit}$$

$$f\left(\left\{ y \middle| y \in R \wedge \left(\pi_X r = \pi_X y\right)\right\}\right)$$

Die Aggregation ist ein relationaler Operator, d.h. sie erhält als Eingabe eine Relation und erzeugt als Ergebnis wieder eine Relation. Hierbei wird eine Projektion bez. Attributmenge *X* auf die Relation *R* durchgeführt. Das Ergebnis der Projektion wird mit dem numerischen Ergebnis der Aggregationsfunktion konkateniert. Die Aggregationsfunktion wird jeweils auf die Attributwerte aller Tupel angewandt, deren Ausprägungen in den Attributen der Menge *X* übereinstimmen. Eine Aggregationsfunktion *f* berechnet somit aus einer Menge von Werten einen skalaren Wert *s*.

Für die Implementierung einer Aggregationsfunktion ist eine Gruppierung nach den Gruppierungsattributen notwendig. Betrachten wir hierzu folgende Anfrage:

Berechne die Summe der Verkäufe aufgegliedert nach Jahr, Produktgruppe und Region. Diese Anfrage kann in SQL wie folgt formuliert werden:

```
SELECT YEAR, PRODUCTGROUP,REGION, SUM(PRICE)
FROM GFK-CUBE
GROUP BY YEAR, PRODUCTGROUP,REGION
```

Konventionelle Aggregationsalgorithmen sortieren oder hashen zunächst die Quellrelation bezüglich der Gruppierungsattribute. Nach der Sortierung kann die Aggregagtionsberechnung durch einfaches, sequenzielles Lesen abgearbeitet werden,

wobei für jedes Tupel einer Gruppe die Aggregationsfunktion aufgerufen wird (siehe Kapitel 6).

Grundsätzlich ist Sortieren oder Hashen der Flaschenhals für die Anfragenverarbeitung, da die Ergebnisse erst weitergereicht werden können, wenn die Quellrelation vollständig verarbeitet worden ist. Kann das Ergebnis trotz *früher Aggregation* [BitW83], d.h. ein Lauf kann nie mehr Tupel haben als das Ergebnis, nicht im Arbeitsspeicher gehalten werden, kommt es zu einem *externen Sortieren*. Um nun den drei-dimensionalen Cube zu berechnen, müssen bereits 8 Gruppierungen berechnet werden. Das Ergebnis ist dann die Vereinigung der Gruppierungen.

9.2.3 Klassifikation von Aggregationsfunktionen

Um die Aggregationsfunktionen zu klassifizieren, definieren wir zunächst die Aggregationsfunktion formal.

Definition 9-10: Aggregationsfunktion

Eine Aggregationsfunktion ist eine Funktion *h*(), die ein Element *E* der Potenzmenge des Wertebereiche *F* auf einen einzelnen Wert *s* aus dem Bildbereich *D* abbildet:

$$H : 2^F \rightarrow D$$

und es gilt:

$$H(E) = s$$

E ist eine Menge von Fakten. Auf dieser Menge *E* wird eine Funktion *h()* angewendet. Die Funktion *h*(*E*) berechnet den skalaren Wert *s*, der wiederum als Fakt betrachtet werden kann. Somit wirkt sich eine Aggregationsfunktion immer verdichtend auf eine gegebene Datenmenge *E* aus (siehe 9.2.2).

Beispiel:
Auf die Menge E wird nun die Aggregationsfunktion *h*( ) angewendet, die den skalaren Wert *s* liefert. Sei *h*( ) die Aggregationsfunktion SUM.

$$SUM(E)=s$$

E hat folgende Ausprägung:

$$E = \{1,2,3,4,5,6,7,8\}$$

Wendet man nun die Aggregationsfunktion auf die Menge E an, erhält man den skalaren Wert 36.

Basierend auf Definition 9-10 können wir nun die Aggregationsfunktionen nach ihrer Additivität klassifizieren.

Definition 9-11: Semi-Additivität von Aggregationsfunktionen

Gegeben sei eine Grundmenge $E \in 2^F$ sowie eine disjunkte Zerlegung $\mathbb{E} = \{A_1, A_2, \ldots, A_n\}$ auf der Grundmenge E. Eine Aggregationsfunktion $h(\,)$ ist semi-additiv, wenn eine Aggregationsfunktion $g(\,)$ existiert, so dass gilt.

$$h(A_1) = s_1 \,,\, h(A_2) = s_2, \ldots, h(A_3) = s_n$$

und

$$g(\{s_1, \ldots, s_n\}) = h(E)$$

Ist die Aggregationsfunktion $g(\,)$ identisch mit der Aggregationsfunktion $h()$, so ist der Aggregationsoperator $h(\,)$ *additiv*.

Dieser Sachverhalt ist für die additive Aggregationsfunktion SUM( ) in Abbildung 9-8 illustriert. Hierbei wird ein 2-dimensionaler Basisraum **Ω** mit einer Auflösung von 4*x*4 Punkten betrachtet. Jeder Zelle ist ein Wert, die Fakten, zugeordnet. Wendet man die Summenfunktion zunächst auf Dimension 1 an, so dass man die einzelnen Zeilen zu einem Wert aufaddiert, reduziert sich die Dimension zu einer Spalte. Wendet man dann die Summenfunktion erneut auf die gerade berechnete Spalte an, erhält man genau einen Wert, wie in der Definition 9-10 gefordert. Das gleiche Ergebnis erhält man, wenn man zunächst entlang Dimension 2 die Spalten aufsummiert und dann die neu erzeugte Zeile zu einem einzigen Wert aufaddiert.

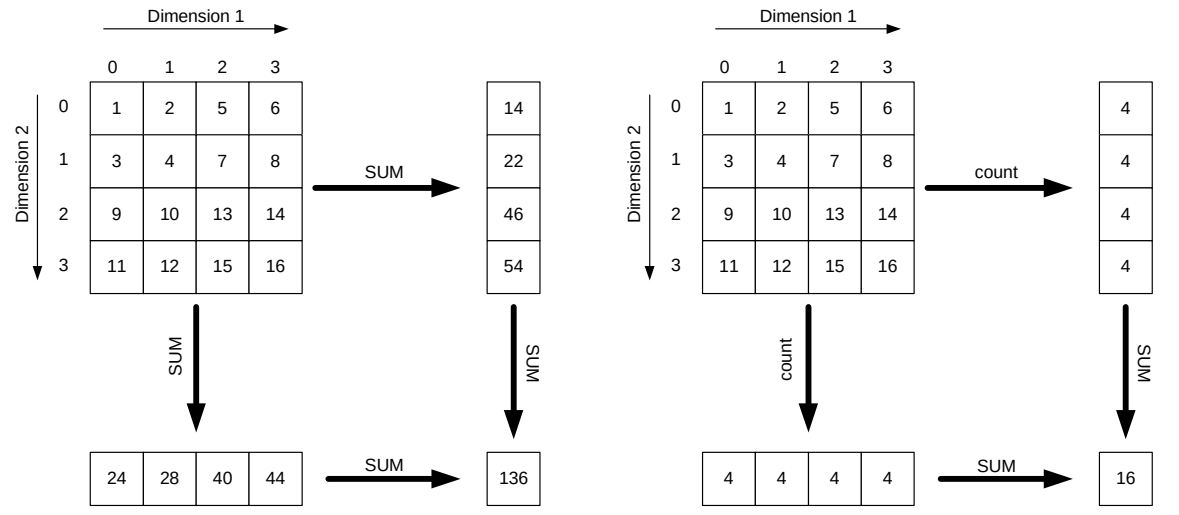


Abbildung 9-8:Additivität der SUM() und semi-additive Count()

Neben der Summe sind die Minimum- und Maximum-Aggregationsfunktionen *additiv*. Die Kardinalität (in SQL: COUNT(*)) hingegen ist *semi-additiv*, da die Aggregationsfunktionen $g()$ und $h()$ nicht identisch sind. Dieser Sachverhalt ist in Abbildung 9-8 verdeutlicht. Hierbei wird zunächst die Kardinalitätsfunktion COUNT auf den Basisraum **Ω** bezüglich einer Dimension angewendet, so dass $h()$ durch COUNT

repräsentiert wird. Die daraus entstehenden Kardinalitätszahlen werden dann mit der Summierungsfunktion SUM zu einem einzelnen Wert aufaddiert, so dass g() auf SUM abgebildet wird. Das Ergebnis ist im obigen Beispiel dementsprechend 16, da der Ω aus 16 Tupel besteht.

Eine weitere Klasse für Aggregationsfunktionen ist die *indirekte Additivität*, die wie folgt definiert ist:

Definition 9-12: Indirekte Additivität

Eine Aggregationsfunktion *h*() ist *indirekt additiv* bezüglich einer Grundmenge *E* genau dann, wenn sie durch einen endlichen und konstanten algebraischen Ausdruck über (*semi-*)*additive* Aggregationsfunktionen definiert werden kann.

Betrachten wir hierzu die Aggregationsfunktion AVG(), die das *arithmetische Mittel* für eine Grundmenge *E* berechnet. Formal ist das *arithmetische Mittel* wie folgt definiert:

Gl: 9-2

$$AVG(E) = \frac{\sum_{i=1}^{n} a_i}{n} \text{ mit } a_i \in E$$

Die arithmetische Mittelwertberechung besteht somit aus der Summe der einzelnen Elemente der Grundmenge *E*, deren Ergebnis durch die Anzahl der Elemente, also deren Kardinalität |*E*|, geteilt wird. Die Aggregationsfunktion SUM ist *additiv* und die Kardinalitätsfunktion *semi-additiv.*

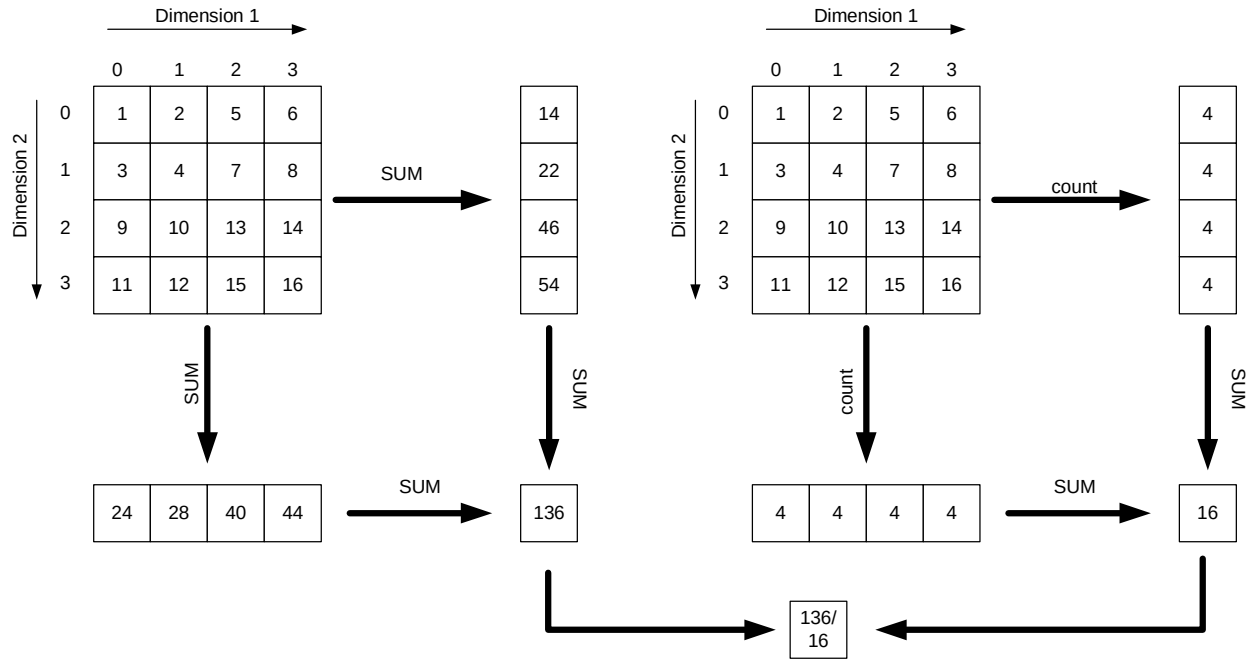


Abbildung 9-9: semi-additive Arithmetische Mittelwertberechnung

Durch die Kombination der beiden (*semi-*)*additiven* Aggregationsfunktionen kann somit der arithmetische Mittelwert berechnet werden. Der vollständige algebraische Ausdruck lautet:

$$AVG(E) = \frac{SUM(E)}{COUNT(E)} \qquad \text{Gl: 9-3}$$

Da es sich um einen endlichen und konstanten algebraischen Ausdruck von (semi)-additiven Aggregationsfunktionen handelt, ist das arithmetische Mittel eine *indirekte additive* Aggregationsfunktion. Die indirekte Additivität von AVG () ist in Abbildung 9-9 illustriert.

Die dritte Klasse bilden die *nicht-additiven Aggreationsfunktionen*, womit die Klassifikation vervollständigt wird.

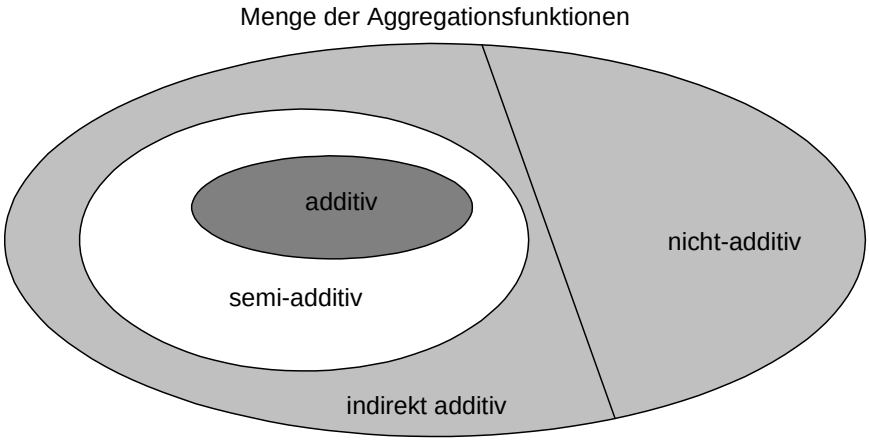
Definition 9-13:Nicht-additive Aggregationsfunktionen

Eine Aggregationsfunktion, die weder (semi)-additiv noch indirekt-additv ist, heißt *nicht-additive* Aggregationsfunktion.

Eine Instanz dieser Klasse ist der Median. Hierbei wird aus einer Grundmenge *E* genau das Element *a* gesucht, das bezüglich einer Ordnungsrelation „<“ genau in der Mitte liegt, so dass genauso viel Vorgänger wie Nachfolger existieren[21]. Betrachten wir dazu zunächst die formale Definition des Medians:

$$MEDIAN(\langle E \rangle) = a_i \text{ mit } i = \lceil |E|/2 \rceil \qquad \text{Gl: 9-4}$$

Da zur Berechnung stets auf die Gesamtheit der Ursprungsmenge Bezug genommen werden muss, handelt es sich bei dem *Median* um eine *nicht-additive* Aggregationsfunktion.



[21] |E| muss dann ungerade sein.

Abbildung 9-10:Klassifikation von Aggregationsfunktionen

Aggregationsfunktion	Berechnung	Klassifikation
SUM	SUM(SUM(X_i))	Additiv
MIN	MIN(MIN(X))	Additiv
MAX	MAX(MAX(X))	Additiv
COUNT	SUM(COUNT(X))	semi-additiv

Tabelle 9-3: Additive und semi-additive Aggregationsfunktionen

9.2.4 Ableitbarkeit von Aggregationsgittern

Basierend auf der Klassifikation von Abschnitt 9.2.3 führen wir nun das Konzept der Ableitbarkeit von Aggregationsfunktionen ein. Dieser Abschnitt basiert auf [Leh98] und [ZirMB00a].

Die Quintessenz dieser Klassifikation ist die Erkenntnis, dass man Aggregate von anderen (Teil-)Aggregaten zur Berechnung ableiten kann. Dies ist die Voraussetzung, um sinnvoll Präaggregationsstrategien zur Beschleunigung von Cube-Anfragen in multidimensionalen Datenbanksystemen einzusetzen.

Die Voraussetzung dafür, dass ein Aggregat von einem bereits berechneten Aggregat abgeleitet werden kann, ist eine geeignete Zerlegung der Grundmenge *B*. Die Ableitungsrelation, die eine feinere Zerlegung ***J*** auf eine Zerlegung ***A*** abbildet, wird als Verbindungsrelation bezeichnet.

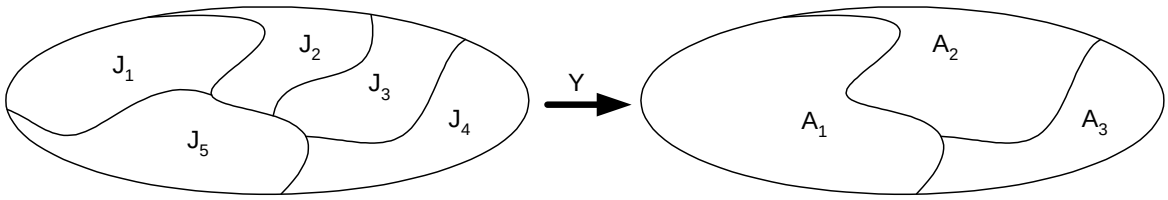


Abbildung 9-11: Verbindungsrelation

Definition 9-14: Verbindungsrelation

Eine Verbindungsrelation *Y* ist ein Relation, die die Klassifikationsregeln zwischen zwei Zerlegungen ***A*** und ***J*** wie folgt festlegt:

$$Y = \{(J,A) | (A \in \mathbf{A}) \wedge (J \in \mathbf{J}) \wedge (J \cap A \neq \emptyset)\}$$

Die Verbindungsrelation enthält alle geordneten Mengenpaare, deren Schnitt nicht leer ist. Für die in Abbildung 9-11 dargestellten Zerlegungen ergibt sich folgende Verbindungsrelation *Y*:

J_1	A_1
J_5	A_1
J_2	A_2
J_3	A_2
J_4	A_3

Tabelle 9-4: Verbindungsrelation *Y*

Mit Definition 9-6 und Definition 9-14 kann man den Satz über die Existenz einer Ableitung wie folgt formulieren:

Satz 9-1: Existenz einer Ableitung über eine Verbindungsrelation

Gegeben sind zwei *Zerlegungen* $\underline{\boldsymbol{J}}$ und $\underline{\boldsymbol{A}}$ und eine Verbindungsrelation *Y*. Eine Zerlegung $\underline{\boldsymbol{A}}$ ist von einer Zerlegung $\underline{\boldsymbol{J}}$ ($\underline{\boldsymbol{J}} \rightarrow \underline{\boldsymbol{A}}$) durch *Y* ableitbar genau dann, wenn gilt:

$$\forall A \in \underline{\boldsymbol{A}}: \bigcup Y^{-1}(A) = A \text{ mit } Y^{-1}(A) := \left\{ J \middle| (J, A) \in Y \right\}$$

Ein ausführlicher Beweis von Satz 9-1 ist in [Sato81] zu finden.

Basierend auf diesem Satz kann man die Ableitbarkeit wie folgt definieren:

Definition 9-15: Ableitbarkeit (konstruktiv)

Erfüllt die Verbindungsrelation *Y* den Satz 9-1, so wird *Y* als Reklassifikationsregel von $\underline{\boldsymbol{J}}$ nach $\underline{\boldsymbol{A}}$ bezeichnet. $\underline{\boldsymbol{A}}$ ist dann über die Verbindungsrelation *V* von $\underline{\boldsymbol{J}}$ ableitbar ($\underline{\boldsymbol{J}} \xrightarrow{Y} \underline{\boldsymbol{A}}$).

Da es sich bei der Ableitbarkeit um eine Überführung einer feineren Zerlegung einer Basismenge in eine gröbere Zerlegung handelt, sind die bekannten Sätze der *Reflexivität* und *Transitivität* für Zerlegungen hier anwendbar. Ein Beweis ist in [Stu95] zu finden.

Kombiniert man die Eigenschaften der Ableitbarkeit von Aggregationsfunktionen mit ihrer Eigenschaft der Additivität, kann bei einer gegebenen Partitionierung angegeben werden, welche Aggregate von bereits berechneten Aggregaten abgeleitet werden können. Dies führt zu dem Begriff der *zerlegungsorientierten Berechnung von Aggregationsoperationen*.

Satz 9-2: Zerlegungsorientierte Berechnung von Aggregationsoperationen

Eine Aggregationsoperation für eine gegebene Zerlegung $\underline{\boldsymbol{A}}$ der Basismenge *B* und einer Aggregationsfunktion *g*() ist aus der Zerlegung $\underline{\boldsymbol{J}}$ genau dann berechenbar, wenn *g*() additiv, semi-additiv oder indirekt additiv und $\underline{\boldsymbol{A}}$ über die Verbindungsrelation *Y* von $\underline{\boldsymbol{J}}$ ableitbar ($\underline{\boldsymbol{J}} \xrightarrow{Y} \underline{\boldsymbol{A}}$) ist.

9.2.5 Aggregationsgitter

Aggregationsgitter werden benutzt, um eine Menge von Aggregaten effizient zu berechnen [AgaAD+96],[HarRU96]. Der wesentliche Leistungsgewinn bei der Verwendung von Aggregationsgittern besteht in der Tatsache, dass einige Aggregationsfunktionen nicht von den Basisdaten berechnet werden müssen, sondern von bereits berechneten Teilaggregaten abgeleitet werden können (siehe Kapitel 9.2.4).

Betrachten wir nun das Aggregationsgitter für die CUBE-Anweisung in Abbildung 9-6. Jeder Knoten repräsentiert eine Gruppierung des Cubes.

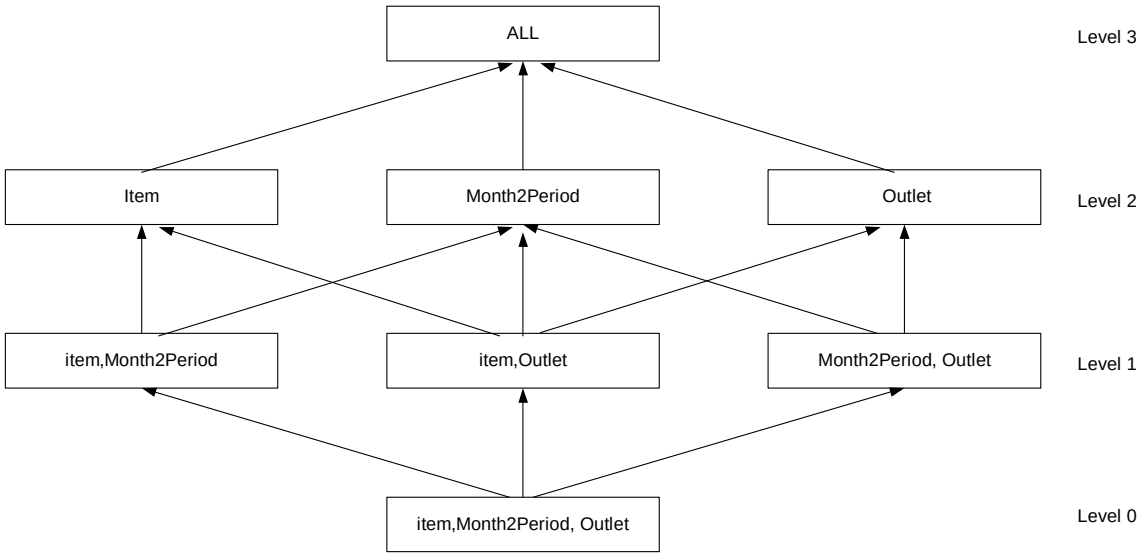


Abbildung 9-12: Aggregationsgitter für Cube Anweisung

Üblicherweise wird ein Aggregationsgitter wie folgt abgearbeitet: Für jeden Knoten wird der konventionelle Aggregationsalgorithmus für die jeweilige Gruppe ausgeführt (siehe Kapitel 9.2.2). Beginnend mit Ebene 0 der Faktentabelle wird das Ergebnis der Ebene *i* als Eingabe für die Ebene *i* + 1 benutzt. Dies setzt die Ableitbarkeit der Aggregationsfunktion *h*( ) voraus.

9.2.6 Konventionelle Berechnung des Datenwürfels durch Sortieren

Bei den hier vorgestellten Methoden werden Aspekte der parallelen Verarbeitung nicht berücksichtigt. Dennoch sei an dieser Stelle darauf hingewiesen, dass die hier vorgestellten Techniken durch kleine Änderungen auch in einem parallelen Datenbanksystem eingesetzt werden können.

Basierend auf unserem Kostenmodell aus Kapitel 3.2.1.2 berechnen wir zunächst die E/A-Kosten des Aggregationsgitters unter der Annahme, dass keine zusätzlichen E/A-Operationen notwendig sind, falls das Zwischenergebnis in den Arbeitsspeicher passt. Für die weitere Betrachtung nehmen wir an, dass der Arbeitsspeicher einen Cache für die Berechnung des Aggregationsgitters von 16000 Seiten bereit stellt, d.h. für eine Seitengröße von 2 KB ergibt sich eine Cachegröße von ca. 32 MB.

Zur Berechnung des Datenwürfels nehmen wir den Pfad durch das Aggregationsgitter, der auf folgender Heuristik basiert:

- Benutze bereits existierende Ordnungen
- Leite das Aggregat aus den kleinsten Vorgängeraggregaten ab

Diese Heuristik erwies sich für viele praktische Anwendungen als geeignet [Leh98]. Abbildung 9-13 zeigt die Berechnungspfade für das in Abbildung 9-12 dargestellt Aggregationsgitter.

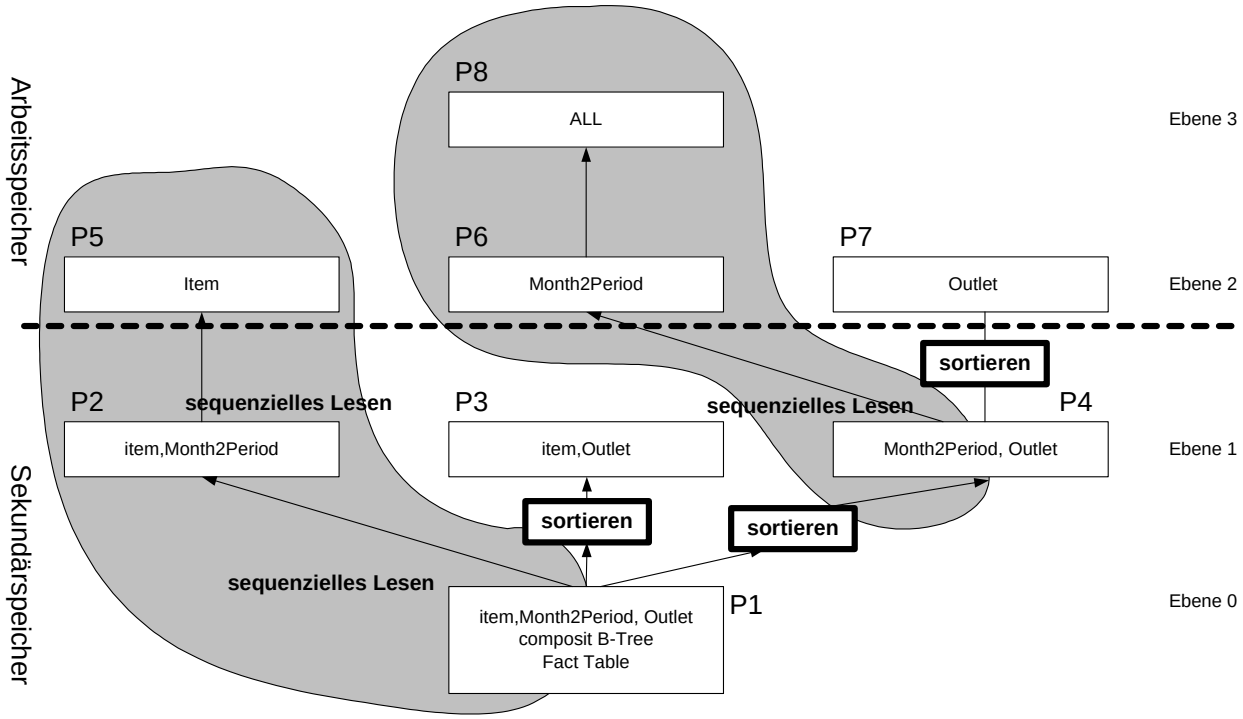


Abbildung 9-13: Aggregationsgitter mit Sortieren

Die Größen der einzelnen Knoten des Aggregationsgitters und der Speicherort sind in Tabelle 9-5 zusammengefasst.

Aggregate	Größe in Seiten	Speicherort	Beschriftung
Item, Month2Period, Outlet	1929735	Sekundärspeicher	P1
Item, Month2Period	138033	Sekundärspeicher	P2
Item, Outlet	473488	Sekundärspeicher	P3
Month2Period, Outlet	24953	Sekundärspeicher	P4
Item	1803	Arbeitsspeicher	P5
Month2Period	1	Arbeitsspeicher	P6
Outlet	477	Arbeitsspeicher	P7
ALL	1	Arbeitsspeicher	P8

Tabelle 9-5: Größe der Aggregate

Der Operator „Externes Sortieren“ ist als Kasten mit der Beschriftung „sortieren“ in das Aggregationsgitter aufgenommen worden, da zur Erzeugung des Aggregats auf der nächsten Ebene Zwischenergebnisse (Läufe) auf den Sekundärspeicher geschrieben werden. Nach Gleichung Gl: 4-12 ergeben sich für das externe Sortieren folgende E/A-Kosten:

$$C_{sortieren}(P) = 4 \cdot c_{lesen/schreiben}(P) \qquad \text{Gl: 9-5}$$

Die Faktentabelle ist ein zusammengesetzter, clusternder B-Baum. Der Schlüssel besteht aus den Attributen *ITEM*, *Mont2Period* und *Outlet* in lexikographischer Ordnung. Zur Berechnung des Aggregats <Item, Month2Period> muss der B-Baum sequenziell gelesen werden, d.h. es müssen P_1 Seiten gelesen und P_2 Seiten geschrieben werden. Somit erhält man für die Berechnung des Aggregats <Item, Month2Period>

$$c_{\langle Item, Month2Period \rangle} = c_{lesen}(P_1) + c_{schreiben}(P_2) \qquad \text{Gl: 9-6}$$

Die Berechnung des Aggregats <Item> kann auf dem Ergebnis <Item, Mont2Period> aufbauen und benötigt somit ein sequenzielles Lesen und Schreiben.

$$c_{\langle Item \rangle} = c_{lesen}(P_2) + c_{schreiben}(P_5) \qquad \text{Gl: 9-7}$$

Um nun den Knoten ⟨Item, Outlet⟩ zu berechnen, muss die Faktentabelle sortiert werden. Dies führt zu P_1 Leseoperationen, um die Initialläufe zu erzeugen. Diese werden dann auf den Sekundärspeicher zurückgeschrieben. Die Verschmelzung der Läufe sowie die Aggregatsberechnung benötig weitere P_1 Lese-Operationen. Schließlich muss das Ergebnis der Aggregation zurückgeschrieben werden. Dies führt zu P_3 Schreiboperationen.

$$c_{\langle Item, Outlet \rangle} = 2 \cdot c_{lesen}(P_1) + c_{schreiben}(P_1) + c_{write}(P_3) \qquad \text{Gl: 9-8}$$

Um ⟨Month2Period, Outlet⟩ zu berechnen, muss die Faktentabelle nach *Mont2Period, Outlet* sortiert werden. Dies führt zu folgenden Kosten:

$$c_{\langle Month2Period, Outlet \rangle} = 2 \cdot c_{lesen}(P_1) + c_{schreiben}(P_1) + c_{schreiben}(P_4) \qquad \text{Gl: 9-9}$$

Die Ableitung von <Month2Period> vom Vorgängeraggregat benötigt nur ein sequenzielles Lesen, da die Zielordnung bereits existiert, sowie das Schreiben der Aggregate.

$$c_{\langle Month2Period \rangle} = c_{lesen}(P_4) + c_{schreiben}(P_6) \qquad \text{Gl: 9-10}$$

Schließlich kann der Knoten ⟨Outlet⟩ durch Sortieren des Knotens ⟨Mont2Period, Outlet⟩ berechnet werden. Hierbei entstehen folgende Kosten:

$$c_{\langle Outlet \rangle} = 2 \cdot c_{lesen}(P_4) + c_{schreiben}(P_4) + c_{schreiben}(P_7) \qquad \text{Gl: 9-11}$$

Somit ergeben sich folgende E/A-Kosten :

$$c_{cube}^{aggregate/sort} = 7 \cdot c_{lesen/schreiben}(P_1) + 2 \cdot c_{lesen/schreiben}(P_2) + c_{lesen/schreiben}(P_3) + 5 \cdot c_{lesen/schreiben}(P_4) + \qquad \text{Gl: 9-12}$$

$$+ c_{lesen/schreiben}(P_5) + c_{lesen/schreiben}(P_6) + c_{lesen/schreiben}(P_7)$$

$$= 7 \cdot c_{lesen/schreiben}(1929735) + 2 \cdot c_{lesen/schreiben}(138033) + c_{lesen/schreiben}(473488) + 5 \cdot c_{lesen/schreiben}(24953) +$$

$$+ c_{lesen/schreiben}(1803) + c_{lesen/schreiben}(1) + c_{lesen/schreiben}(477)$$

$$= 14.384.745 \; c_{lesen/schreiben}$$

9.2.7 Berechnung mit TempTris- und UBG-Algorithmus

Die hier vorgestellte neue Verarbeitungstechnik kombiniert den TempTris- und UBG-Algorithmus. Hierbei wird die Faktentabelle des GfK-Schemas als UB-Baum organisiert. Der UBG-Algorithmus, eine Weiterentwicklung des Tetris-Algorithmus [MarZB99],[MarB98], liest die Faktentabelle und die temporären UB-Bäume sortiert nach einer Dimension und berechnet das Aggregat. Der TempTris-Algorithmus wiederum erzeugt aus einem sortierten Tupel-Strom, der vom UBG-Algorithmus erzeugt wird, einen temporären UB-Baum. Der wesentliche Vorteil dieser Methode ist:

- Reduzierung der E/A-Kosten um ca. 50% beim Sortieren durch Ausnutzung der UB-Baumpartitionierung gegenüber externem Sortieren
- Mehrfachverwendung von temporären UB-Bäumen

Die temporären UB-Bäume stellen Knoten im Aggregationsgitter dar. Das Aggregat <Month2Period, Outlet> wird somit als UB-Baum organisiert abgelegt. Es dient somit als Quelle für die Aggregate <Month2Period> und <Outlet>.

Nach Kapitel 6.5.2 und Gl: 6-32 benötigt der UBG-Algorithmus für *n* Gruppierungsattribute

$$cache_{UBG}(P) = 2^{\frac{\log_2 P}{d}} \cdot n - 1$$

Es werden somit für die Erzeugung des Knoten <Month2Period, Outlet> nur 477 Seiten benötigt, da $n = 2$ ist und die Daten sortiert nach Month2Period gelesen werden und die Ausprägungen von Outlet nach Tabelle 9-5 genau 477 Seiten benötigen.

Hier ist jedoch zu beachten, dass bei dieser Formel von einer Gleichverteilung ausgegangen wird. Der zusätzliche Speicherbedarf für die internen Strukturen des UBG-Algorithmus sind hier nicht berücksichtigt und Bestandteil weiterer Untersuchungen.

Der Cachebedarf für den TempTris-Algorithmus entspricht dem des Tetris-Algorithmus, wie in Kapitel 7.1 und [ZirMB01] zu entnehmen ist.

Im Gegensatz zur Verarbeitungstechnik, die in Abbildung 9-13 dargestellt ist, erlaubt die UB-Organisation der Faktentabelle den Einsatz des UBG-Algorithmus. Somit können die

Aggregate <Item, Month2Period> und <Item, Outlet> ohne externes Sortieren verarbeitet werden. Zusätzlich erzeugt der UBG-Algorithmus einen sortierten Storm von Tupel, entweder sortiert nach Month2Periods oder Outlet. Dieser Strom kann dann wiederum als Eingabe für den TempTris-Algorithmus fungieren. Das Ergebnis ist ein temporärer UB-Baum.

Aggregate	Größe in Seiten	Speicherort	Beschriftung
Item, Month2Period, Outlet	1929735	Sekundärspeicher	P1
Item, Month2Period	138033	Sekundärspeicher	P2
Item, Outlet	473488	Sekundärspeicher	P3
Month2Period, Outlet	31192	Sekundärspeicher	P4
Item	1803	Arbeitsspeicher	P5
Month2Period	1	Arbeitsspeicher	P6
Outlet	477	Arbeitsspeicher	P7
ALL	1	Arbeitsspeicher	P8

Tabelle 9-6

Da der TempTris-Algorithmus keine 100%-ige Seitenauslastung garantieren kann, ist die Größe des temporären UB-Baums größer als das Zwischenergebnis des externen Sortierens (Merge-Sort). Geht man von einer Seitenauslastung von 81-% aus (siehe 7.3.5), erzeugt der TempTris-Algorithmus einen UB-Baum von 31192 Seiten. Der entsprechende B-Baum, der durch Merge-Sort entsteht, enthält 24953 Seiten. Durch das Erzeugen eines temporären UB-Baums kann der UBG-Algorithmus eingesetzt werden, um die Aggregate <Month2Period> und <Outlet> zu erzeugen. Diese Strategie verhindert externes Sortieren und reduziert somit die E/A-Kosten.

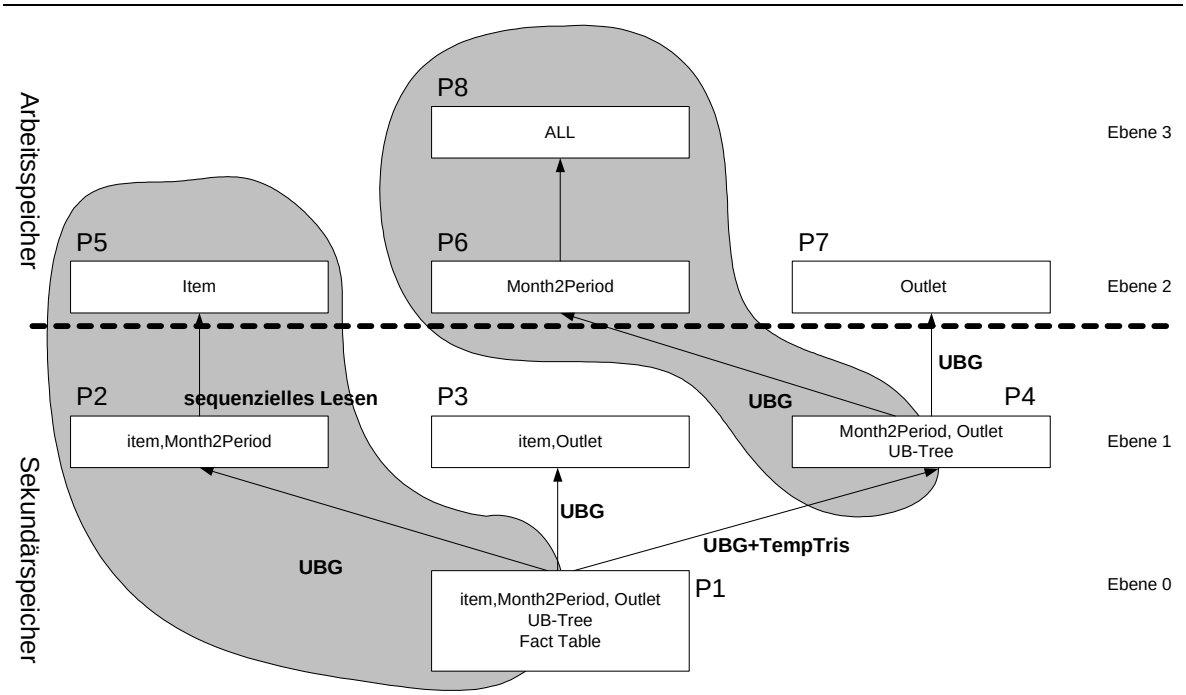


Abbildung 9-14

Abbildung 9-14 zeigt den Verarbeitungsprozess für die CUBE-Berechnung. Für die E/A-Kosten, die durch den TempTris- und UBG-Algorithmus erzeugt werden, ergibt sich für obiges Beispiel folgende E/A-Kosten.

$$c_{\langle Item,Month2Period\rangle} = c_{read}(P_1) + c_{write}(P_2) \qquad \text{Gl: 9-13}$$

$$c_{\langle Item\rangle} = c_{read}(P_2) + c_{write}(P_5)$$

$$c_{\langle Item,Outlet\rangle} = c_{read}(P_1) + c_{write}(P_3)$$

$$c_{\langle Month2Period,Outlet\rangle} = c_{read}(P_1) + c_{write}(P_4)$$

$$c_{\langle Month2Period\rangle} = c_{read}(P_4) + c_{write}(P_6)$$

$$c_{\langle Outlet\rangle} = c_{read}(P_4) + c_{write}(P_7)$$

Die Gesamtkosten sind somit:

$$c_{cube}^{temptris/UBG} = 3\cdot c_{read/write}(P_1) + 2\cdot c_{read/write}(P_2) + c_{read/write}(P_3) + 3\cdot c_{read/write}(P_4) + \qquad \text{Gl: 9-14}$$

$$+ c_{read/write}(P_5) + c_{read/write}(P_6) + c_{read/write}(P_7)$$

$$= 3\cdot c_{read/write}(1929735) + 2\cdot c_{read/write}(138033) + c_{read/write}(473488) + 3\cdot c_{read/write}(31192) +$$

$$+ c_{read/write}(1803) + c_{read/write}(1) + c_{read/write}(477) = 6634616\cdot c_{read/write}$$

Für dieses Beispiel ergeben sich für die E/A-Kosten durch das externe Sortieren (Merge-Sort) 14384745 Seitenzugriffe. Das sind ca. zwei Mal soviel Seitenzugriffe wie vom TempTris- und UBG-Algorithmus benötigt werden. Die Verbesserung beruht im Wesentlichen auf zwei Fakten:

- Der UBG-Algorithmus liest die Faktentabelle 4 Mal weniger als das externe Sortieren
- Der temporäre UB-Baum <Month2Period, Outlet> in Verbindung mit dem Tetris-Algorithmus spart zwei Mal das Lesen des temporären Ergebnisses.

Im Allgemeinen kann durch die Kombination UBG-Algorithmus und TempTris-Algorithmus die E/A-Komplexität um fast 50% reduziert werden.

9.3 Zusammenfassung

Dieses Kapitel führt zunächst formal das DW-Modell ein. Ein wesentlicher Punkt ist hier die Dimensionshierarchie und deren Ableitbarkeit. Basierend auf der ordnungserhaltenden Strukturierung der Dimensionen kann eine Restriktion auf höherer Hierarchieebene auf einen Bereich der Hierarchieebene 0 eindeutig abgebildet werden. Somit ist es möglich den Bereichsfrage-, den Tetris-, den SRQ- oder den UBG-Algorithmus für Anfragen mit Restriktionen auf höheren Hierarchieebenen einzusetzen und das Ergebnis aus den Basisdaten zu berechnen.

Danach wird der CUBE-Operator analysiert. Der CUBE-Operator besteht im Wesentlichen aus zwei Operatoren, aus dem mehr-dimensionalen Bereichsanfrage-Operator und dem Gruppierungs-Operator. Um die mehr-dimensionalen Bereichsanfragen optimal auszunutzen wird die Faktentabelle als UB-Baum organisiert. Der Gruppierungsoperator wird durch den UBG-Algorithmus realisiert. Da der CUBE-Operator den vollständigen Cube berechnet, werden Aggregationsgitter eingesetzt. Aggregationsgitter basieren auf der Erkenntnis, dass Aggregate abhängig von ihrer Aggregationsfunktion additiv, semi-additiv, indirekt-additiv oder nicht additiv sind. Ist eine Aggregationsfunktion additiv, kann ein Aggregat aus Teilaggregaten abgeleitet werden. Um Teilaggregate ohne Umsortierung als Quelle für weitere Aggregate zu nutzen, werden sie als UB-Bäume organisiert. Dies für zum Einsatz des TempTris-Algorithmus. Das wesentliche Ergebnis der Kombination aus UBG-Algorithmus, der die Daten sortiert nach einem Attribute in linearer E/A-Komplexität aus dem Sekundärspeicher liest und der TempTris-Algorithmus, der aus einem sortierten Strom von Tupeln in linearer E/A-Komplexität einen UB-Baum erzeugt, sind somit lineare E/A-Kosten, da beide Algorithmen auf externes Sortieren verzichten. Die Effizienz dieses Verfahrens wurde durch eine Kostenanalyse auf Basis des GfK-Datawarehouses gemacht. Für die Berechnung von 7 Aggregaten konnte eine E/A-Kosten Reduzierung von über 50 % erreicht werden.

Somit kann die Anzahl der vorberechneten Aggregate reduziert werden, da die Berechnung der Aggregate von den Basisdaten aus mit linearen E/A-Kosten erzeugt werden kann.

In nature there are no rewards or punishments; there are consequences.

Horace Annesley Vachell (1861–1955)

10 Zusammenfassung und Ausblick

Der Einsatz der mehrdimensionalen Zugriffsstruktur UB-Baum in komplexen Geschäfts-Anwendungen (z.B. SAP R/3), statistischen Datenbanken, Data-Warehouse und Data-Mining haben gezeigt, dass Leistungssteigerungen bis Faktor 10 für die Verarbeitung von Bereichsanfragen möglich sind. Weitere Verbesserungen wurden durch die Integration in den Datenbankkern Transbase von TransAction erzielt. Diese Arbeit untersucht den UB-Baum unter Berücksichtigung der Sortierung.

10.1 Ergebnisse

Um eine Relation zu sortieren oder einen Verbundoperator auszuwerten, sollten Restriktionen, die auf den einzelnen Attributen existieren, ausgenutzt werden, um so die E/A- und CPU-Kosten zu reduzieren.

Der Autor entwickelte mehrere neue Algorithmen, die aus der Z-Ordnung effizient eine Zielordnung erzeugen. Dies führt zu zwei Verbunds-Algorithmen (SRQ- und Tetris-Algorithmus) und einem Gruppierungs-Algorithmus (UBG-Algorithmus). Die Methode wird als sortiertes Lesen bezeichnet. Die wesentliche Innovation dieser Algorithmen ist die Einführung einer neuen Ordnung, der Tetris-Ordnung.

Das sortierte Lesen liest die Regionen aus einem UB-Baum sortiert nach dem Sortierattribut, um eine Bereichsanfrage mit ORDER-Klausel effizient abzuarbeiten. Das sortierte Lesen benutzt die hervorragende mehrdimensionale Zugriffsstruktur UB-Baum und eine Sweepline-Technik, um Bereichsanfrage und Sortieren zu kombinieren. Hierdurch kann die Selektionsphase, d.h. die Auswertung der mehrdimensionalen Bereichsanfrage, und die Sortierphase in einem Schritt durchgeführt werden.

Es wurde gezeigt, dass das sortierte Lesen für mehrdimensionale Anfragen, die typisch in relationalen Datenbanksystemen sind, eine lineare E/A-Komplexität und sublineare Speicher-Kosten für den benötigten Cache hat. Ein wesentlicher Vorteil besteht darin, dass das sortierte Lesen das Teilergebnis kontinuierlich erzeugt, da die Scheiben in Sortierrichtung geladen werden und somit bereits sortiert werden können, obwohl der Rest der Daten noch nicht geladen worden ist. Der Merge-Sort-Algorithmus hingegen kann erst das Ergebnis erzeugen, wenn alle Daten geladen worden sind. Durch das sortierte Lesen geht der Sortier-Operator von einem blockenden zu einem nicht blockenden Operator über. Verglichen mit den existierenden Techniken kann das erste Teilergebnis durch das sortierte Lesen erheblich schneller erzeugt werden. Dies führt zu einer besseren Antwortzeit. Die Technik des sortierten Lesens ermöglicht somit den Einsatz des Pipelinings in der

Anfrageverarbeitung, die ein Sortieroperator enthält. Die Benchmark-Ergebnisse des TPC-H des sortierten Lesens zeigen die Praxisrelevanz dieser Technik. So konnte z.B. die Verarbeitungszeit für *Q*3 gegenüber dem FTS und IOT um Faktor 3 gesteigert werden.

Es wurden zwei Implementierungen für das sortierte Lesen entwickelt, der SRQ-Algorithmus, der aus konstanten Scheiben besteht, und der Tetris-Algorithmus, der immer die minimale Scheibe liest. Die Grundidee des SRQ-Algorithmus besteht darin den Anfragebereich in Sortierrichtung in Scheiben mit konstanter Breite aufzuteilen. Die einzelnen Scheiben werden mit dem Bereichsanfrage-Algorithmus abgearbeitet. Die Nachteile dieses Verfahrens bestehen zum einen darin, dass man statistische Daten über den UB-Baum besitzen muss, wie zum Beispiel seine Dimensionalität und die Größe in Seiten. Zum anderen besteht die Gefahr Daten mehrmals zu lesen. Für nicht gleichverteilte Daten kommt es zu einer Vergrößerung der benötigten Cachegröße. Dies kann im schlimmsten Fall dazu führen, dass die gesamte Ergebnismenge in einer Scheibe enthalten ist. Das folgende Kapitel führt einen Algorithmus ein, der dieses Problem löst.

Die Alternative ist der Tetris-Algorithmus. Die Grundidee besteht darin, die Sweepline nur soweit in Sortierrichtung zu verschieben, bis wieder eine vollständige Scheibe im Arbeitsspeicher enthalten ist. Durch die Einführung einer Raum füllenden Kurve, der Tetris-Kurve, die auf der Tetris-Ordnung basiert, wird das *n*-dimensionale Schnittproblem auf ein 1-dimensionales Schnittproblem reduziert. Der Tetris-Algorithmus lädt jeweils die minimale Scheibe aus dem UB-Baum, in dem der Anfragebereich nach der T- Ordnung abgearbeitet wird. Somit wird jede Region nur einmal gelesen.

Nach der formalen Einführung der Tetris-Ordnung wurden die Eigenschaften der T-Kurve untersucht. Es konnte bewiesen werden, dass die T-Kurve eine raumfüllende Funktion auf einem diskreten Basisraum Ω ist. Die wesentliche Erkenntnis dieser Untersuchung besteht darin, dass die Sweep-Hyperebene, die den statischen Bereich vom dynamischen Bereich trennt, durch ein zusammenhängendes Intervall dargestellt werden kann und dass jedem Z-Intervall genau eine minimale und maximale T-Adresse zugeordnet werden kann. Somit kann die Hyperebene durch einen Punkt, der als Separator zwischen dem dynamischen und dem statischen Bereich fungiert, ersetzt werden. Anhand dieses Separators in Kombination mit der minimalen oder der maximalen T-Adresse kann entschieden werden, ob ein Z-Intervall den dynamischen Bereich schneidet. Es wurde ein Algorithmus vorgestellt, der die minimale bzw. maximale T-Adresse in $O(n)$ berechnet, wobei n die Anzahl der Bits bezeichnet, die zur Repräsentation der T-Adresse benötigt wird

Bei den Vergleichmessungen zwischen dem SRQ- und dem Tetris-Algorithmus zeigten beide Algorithmen ähnliche Gesamtkosten. Der Tetris-Algorithmus liest immer die minimale Anzahl der Seiten aus dem Sekundärspeicher und entspricht somit dem RQ-Algorithmus. Der SRQ-Algorithmus, der die Scheiben jeweils durch einen RQ-Algorithmus abarbeitet, liest durch die Struktur der Partitionierung des UB-Baums Regionen mehrmals und ist somit dem Tetris-Algorithmus unterlegen. Bei den Messungen für gleichverteilte Daten wurden durch den SRQ-Algorithmus 14% mehr Seiten gelesen als durch den Tetris-Algorithmus. Der SRQ-Algorithmus hat jedoch ein besseres E/A-Verhalten, da der SRQ-Algorithmus die Daten nach der Z-Kurve aus dem Sekundärspeicher liest und somit den Prefetchfaktor *C* besser ausnutzt. Betrachtet man die Zeit, die benötigt wird, um die Seiten zu laden, war der SRQ-Algorithmus dem Tetris-Algorithmus um 6% überlegen. Die Verwaltung des Cache beim Tetris-Algorithmus erzeugt zusätzliche CPU-Kosten. Bei dieser Prototyp-Implementierung erhielt man im Mittel für das Verhältnis SRQ-Algorithmus zum Tetris-Algorithmus den Wert 1:15 an

CPU-Kosten. Beide Algorithmen sind nicht CPU-Bound. Somit können die Ergebnisse aus Kapitel 4 übernommen werden.

Der UBG-Algorithmus berechnet effizient Gruppierungen, auf denen dann Aggregationsfunktionen ausgeführt werden, und entspricht bis auf eine Funktion dem Tetris-Algorithmus. Der Unterschied liegt in der Cachverwaltung. Im Cache werden nur Äquivalenzklassen $[x_k]$ zwischengespeichert, wobei k die Gruppierungsdimension ist. Die Kostenanalyse zeigt, dass die Vorteile der E/A-Kosten des UBG-Algorithmus gegenüber dem IOT und FTS dem des Tetris-Algorithmus entsprechen. Die wesentlichen Kosten entstehen in der Selektionsphase. Durch die Aggregation kann es zu einer Verdichtung der Daten kommen, so dass $P_E \geq P_A$ gilt. Dies führt zu einer Reduzierung der E/A-Kosten gegenüber dem Tetris-Algorithmus. Ein wesentlicher Vorteil liegt in der Reduzierung der Speicherkosten. Für gleichverteilte Daten benötigt der UBG-Algorithmus einen Cache von einer Seite, wenn nur nach der Sortierdimension gruppiert wird. Allgemein kann dieses Prinzip auch für den SRQ-Algorithmus eingesetzt werden.

Der TempTris-Algorithmus löst das Problem, um aus einer sortierten Folge von Daten effizient den UB-Baum zu erzeugen. Für das Massenladen eines DW führt diese neue Technik zu einem erheblichen Leistungsgewinn gegenüber dem externen Sortieren. Sogar im linearen Bereich des externen Sortierens können die E/A-Kosten bezogen auf die Datenseiten um 50% reduziert werden. Der größte Vorteil des TempTris-Algorithmus besteht darin, dass die mehrdimensionale Partitionierung ohne Auslagerung von Zwischenergebnissen auf den Sekundärspeicher, wie es beim externen Sortieren der Fall ist, auskommt. Hierfür wird ein moderater zusätzlicher Cache benötigt. Basierend auf einem Kostenmodell wurde der TempTris-Algorithmus analysiert. Die Analyse ergibt eine Leistungssteigerung gegenüber dem herkömmlichen externen Sortieren bei den E/A-Kosten um den Faktor 2. Die Cachegröße, die der TempTris-Algorithmus benötigt, ist eine Wurzelfunktion der Eingabedaten. Die Cachegröße, die vom externen Sortieren beim einmaligen Lesen der Daten benötigt wird, entspricht der Größe der Eingabedaten. Das theoretische Modell wurde durch Messungen des TempTris und des externen Sortierens auf künstlichen und echten Daten bestätigt.

Die Arbeit schließt mit einer neuen Verarbeitungstechnik des CUBE-Operators und mit dem Einfluss der neuen Techniken auf die relationale Algebra. Es wird eine Verarbeitungstechnik vorgestellt, die ein Aggregationsgitter berechnet. Hierbei wird eine Kombination aus UBG- und TempTris-Algorithmus benutzt. Das wesentliche Ergebnis ist, dass die E/A-Kosten linear sind und somit die E/A-Kosten gegenüber dem externen Sortieren um 50% reduziert werden können.

10.2 Ausblick

Die einzelnen Algorithmen wurden primär auf ihre Anwendbarkeit auf den Sekundärspeicher untersucht. Die Arbeit stellt eine Grundlage für weitere Untersuchungen im Bereich Tertiärspeicher dar. Ein besonders interessantes Gebiet ist die Archivierung, da heutige Datawarehouse-Systeme mit dieser Problematik konfrontiert sind und für diese Problematik keine sinnvolle Lösung existiert. Hier könnte der UB-Baum eine Lösung darstellen.

11 Anhang

11.1 Datenverteilung im Cache

Durch die Ausdehnung der Z-Region werden bereits Tupel geladen, die noch nicht zur aktuellen Scheibe gehören. Diese Tupel müssen diese im Cache gehalten werden. Somit besteht der Cache aus n_{old} Tupel, die bereits im Cache liegen und n_{new} Tupel, die neu eingefügt werden. Die Verteilung der Daten variiert stark zur jeweils verarbeiteten Sweep-Hyperebene. Diese hängt von verschiedenen Faktoren ab wie z.B. von der Anzahl der Tupel, die bereits im Cache enthalten sind, und der Anzahl der neuen Tupel. Diese Faktoren hängen wiederum von der aktuellen Datenverteilung im UB-Baum ab.

Ohne Beschränkung der Allgemeinheit wird hier der gesamte Basisraum Ω betrachtet. Zunächst wird die erste Sweep-Hyperebene $\Psi_{k,0}$ betrachtet. Bei der Verarbeitung der ersten Scheibe müssen Parameter, wie z.B. bereits existierende Tupel im Cache, die Position im Basisraum Ω, nicht berücksichtigt werden, so dass die wesentlichen Eigenschaften deutlich herausgearbeitet werden können.

In diesem Kapitel werden folgende Symbole verwendet.

Symbol	Bezeichnung
s	Anzahl der Stufen
d	Anzahl der Dimensionen
k	Sortierdimension
$\alpha_{[j],(i)}$	Bits auf Stufe j auf Dimension i
Ψ	Sweep-Hyperebene

11.1.1 Erste Sweep-Hyperebene

Für alle Punkte des Basisraums Ω, die Elemente der Sweep-Hyperebene $\Psi_{k,0}$ sind, ergeben sich folgende Bedingungen:

$$\alpha_{[j],(i)} = 0 \qquad \text{falls } i = k$$
$$\alpha_{[j],(i)} \in \{0,1\} \qquad \text{falls } i \neq k \wedge i \in \{1,...,d\} \wedge j \in \{0,...,s-1\}$$

Gl: 11-1

Somit haben die Z-Adressen folgende Struktur:

$$\alpha_{[0],d} \ldots \alpha_{[0],(k+1)}\, \mathbf{0}\, \alpha_{[0],(k-1)} \ldots \alpha_{[0],(1)} \ldots \alpha_{[s-1],(d)} \ldots \alpha_{[s-1],(k+1)}\, \mathbf{0}\, \alpha_{[s-1],(1)} \ldots a_{[s-1],(1)} \qquad \text{Gl: 11-2}$$

Es können somit $s(d-1)$ Bits frei gewählt werden. Somit enthält eine Sweep-Hyperebene

$$2^{s(d-1)} \qquad \text{Gl: 11-3}$$

Punkte.

Da alle Elemente, die in der aktuellen Sweep-Hyperebene liegen, sofort aus dem Cache entfernt werden können, sobald sich die Sweep-Hyperebene bewegt, stellen sie kein Problem dar. Interessant ist jedoch, wie weit sich eine Region, die die erste Sweep-Hyperebene schneidet, ausdehnen muss, um eine Sweep-Hyperebene $\Psi_{k,i}$ mit $i \in \Omega_k$ zu schneiden. Das wiederum bedeutet, dass die Daten, die Elemente der Sweep-Hyperebene $\Psi_{k,i}$ sind, im schlechtesten Fall nach i Scheibenverarbeitungen aus dem Cache entfernt werden können.

Hierzu wird der Begriff Z-Abstand auf der Z-Kurve eingeführt. Der Abstand auf der Z-Kurve von zwei Punkten a und b ist der Betrag der Differenz der jeweiligen Z-Werte:

Definition 11-1: Z-Abstand

Der Z-Abstand von zwei Punkten a und b im d-dimensionalen Basisraum Ω ist:

$$\text{Z-Abstand} = \left|Z(a) - Z(b)\right|$$

Abbildung 11-1 zeigt jeweils einen 2-dimensionalen Basisrum Ω mit einer Auflösung r = 8. Gegeben sei der Punkt a mit der Z-Adresse 10 und der Punkt b mit der Z-Adresse 16. Dann ist der Z-Abstand zwischen den beiden Punkten a und b der Betrag $|Z(a) - Z(b)| = |10 - 16| = 6$.

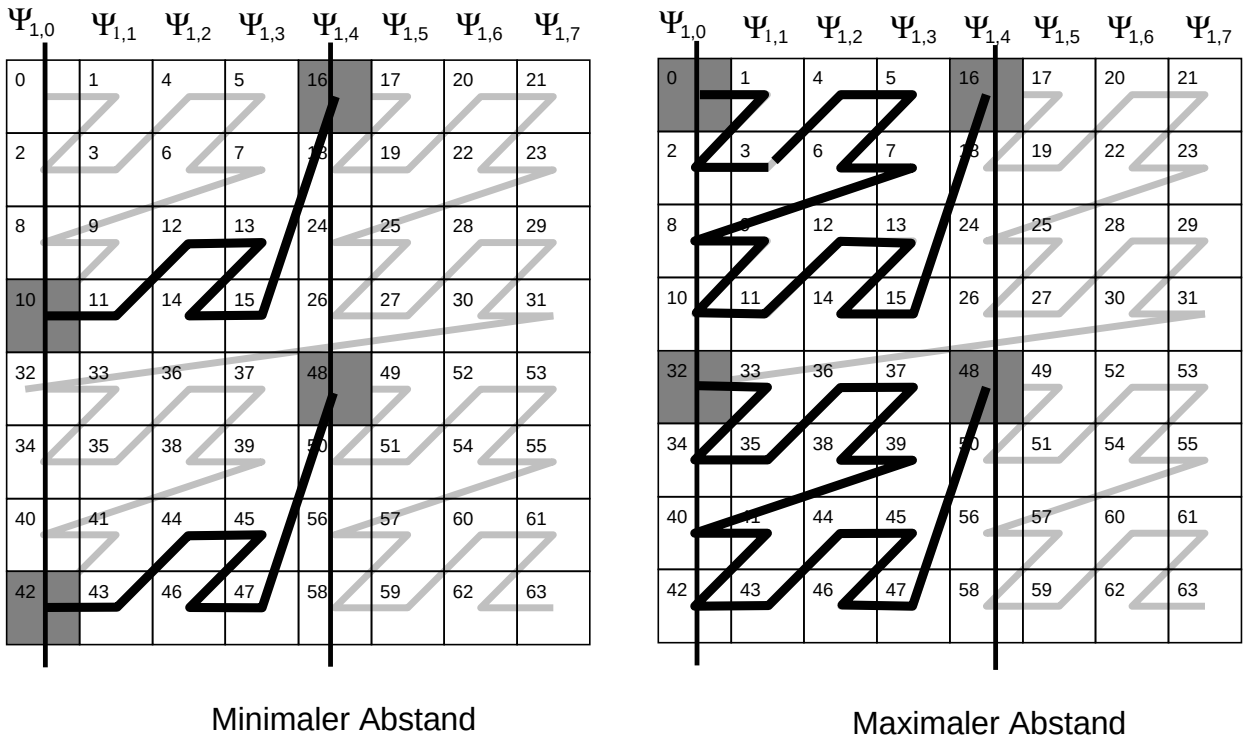


Abbildung 11-1

Als weiteres führen wir den Begriff *Stufenlinie-k,i* in Dimension *k* ein. Hierbei repräsentiert eine Stufenlinie-*i* die Teilung des Raums auf Stufe *i* bezüglich Dimension *k*.

Definition 11-2: Stufenlinie-k,i

> Eine Stufenlinie-k,i ist eine Teilung des gesamten Basisraums auf der i-ten Stufe bezüglich der Dimension *k*.

Da die Teilung des Basisraums Ω rekursiv ist, existiert auf Stufe 0 genau 1 Stufenlinie für jede Dimension des Basisraums, auf Stufe *i* genau 2^i Stufenlinien. Abbildung 11-2 zeigt die Stufenlinien für den 2-dimensionalen Fall.

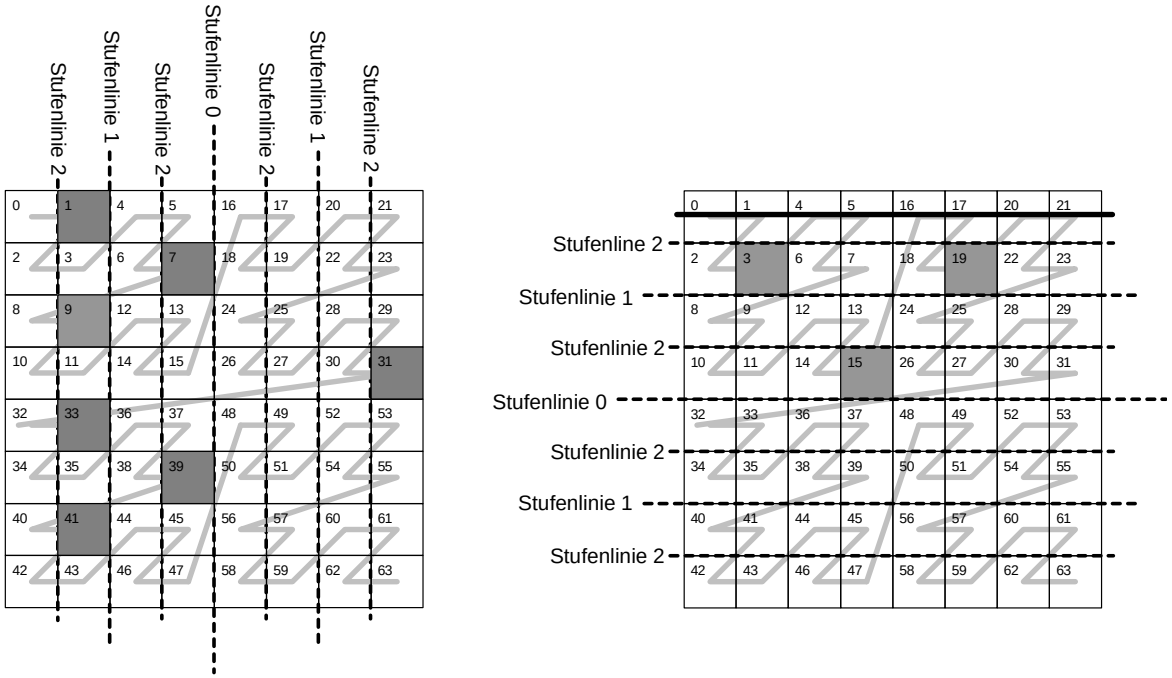


Abbildung 11-2

Es wird nun der minimale und maximale Abstand betrachtet, um von der Sweep-Hyperebene $\Psi_{k,i}$ auf die $\Psi_{k,n}$ Sweep-Hyperebene zu kommen. *k* bezeichnet die Sortierdimension, *i* und *n* die Position der Sweep-Hyperebene auf der Dimension *k*. Für die Untersuchung betrachten wir die Sweep-Hyperebene, für die gilt:

$$n = 2^j \text{ mit } j \in \{0,...,s-1\} \qquad \text{Gl: 11-4}$$

Für die erste Sweep-Hyperebene $\Psi_{1,0}$ betrachten wir somit den Z-Abstand zur Sweep-Hyperebene $\Psi_{1,1}$,$\Psi_{1,2}$ und $\Psi_{1,4}$. Dies entspricht der rekursiven Unterteilung, d.h. es wird eine Stufenlinie-1,1, Stufenlinie1,1 und die Stufenlinie-1,0 überschritten.

Es wird wieder die Struktur der Z-Adresse betrachtet. Damit ein Punkt auf der Sweep-Hyperebene $\Psi_{1,0}$ liegt, müssen die Bits der Z-Adresse $\alpha_{[j],(1)} = 0$ sein. Die Struktur ist in Gl: 11-2 dargestellt. Für die Sweep-Hyperebene $\Psi_{1,n}$ mit $n = 2^j$ gilt:

$$\begin{aligned} \alpha_{[m],(i)} &= 1 \quad \text{falls } m = s-1-j \wedge i = k \\ \alpha_{[m],(i)} &= 0 \quad \text{falls } m \neq s-1-j \wedge i = k \\ \alpha_{[m],(i)} &\in \{0,1\} \quad \text{falls } i \neq k \wedge i \end{aligned} \qquad \text{Gl: 11-5}$$

Somit haben die Z-Adressen folgende Struktur:

$$\alpha_{[0],d} \ldots \alpha_{[0],(k+1)}\, 0\, \alpha_{[0],(k-1)} \ldots \alpha_{[0],(1)} \ldots \underbrace{\alpha_{[s-j],(d)} \ldots \alpha_{[s-j],(k+1)}\, 1\, \alpha_{[s-j],(k-1)} \ldots \alpha_{[s-j],(1)}}_{\text{Stufe } s-1-j} \ldots \quad \text{Gl: 11-6}$$

$$\alpha_{[s-1],(d)} \ldots \alpha_{[s-1],(k+1)}\; 0\, \alpha_{[s-1],(k-1)} \ldots \alpha_{[s-1],(1)}$$

Es werden nun der minimale und maximale Abstand angegeben um die 2^j-te Sweep-Hyperebene von der 0-ten Sweep-Hyperebene zu erreichen. Der minimale und maximale Abstand für den 2-dimensionalen Basisraum Ω von $\Psi_{1,0}$ zu $\Psi_{1,4}$ ist in Abbildung 11-1 dargestellt. Für das Minimum ergibt sicht 6, für das Maximum 16.

Formal kann dies wie folgt beschrieben werden: Der minimale Z-Abstand ergibt sich genau dann, wenn alle Bits $m > s$-1-j außer den Bits der Sortierdimension der Stufe auf 1 gesetzt sind. Die Bits der Sortierdimension werden auf den Wert 0 gesetzt, da dies durch die Sweep-Hyperebene bestimmt ist. Die Bits in Stufe *s*-1-*j*, deren Position kleiner als *k* ist, werden auch auf 1 gesetzt. Die Bits der Sortierdimension der restlichen Stufen werden wegen der Sweep-Hyperebene auf 0 gesetzt.

$$\begin{aligned} &\alpha_{[m],(i)} = 1 \qquad \text{falls } (m = s-1-j) \wedge (i < k) \\ &\alpha_{[m],(i)} = 1 \qquad \text{falls } (m > s-1-j) \wedge i \neq k \\ &\alpha_{[m],(i)} = 0 \qquad \text{falls } i = k \end{aligned} \quad \text{Gl: 11-7}$$

Somit haben die Z-Adressen folgende Struktur:

$$\alpha_{[0],d} \ldots \alpha_{[0],(k+1)}\, 0\, \alpha_{[0],(k-1)} \ldots \alpha_{[0],(1)} \ldots \underbrace{\alpha_{[s-j],(d)} \ldots \alpha_{[s-j],(k+1)}\, \overset{k}{\overbrace{0 1 \ldots 1}}}_{\text{Stufe } s-1-j} \ldots \underbrace{1 \ldots 1\, \overset{k}{\overbrace{0 1 \ldots 1}}}_{\text{Stufe s-1}} \quad \text{Gl: 11-8}$$

Um nun die Sweep-Hyperebene $\Psi_{1,m}$ mit m = 2^j zu erreichen, muss ein Überlauf auf Bitposition $\alpha_{[s-1-j],(k-1)}$ entstehen, damit $\alpha_{[s-1-j],(k)}$ = 1 wird und alle anderen Bits unterhalb von Bit $\alpha_{[s-1-j],(k)}$ auf 0 stehen. Dies entsteht dadurch, das man alle existierenden Nullen in Stufe *s-2-j* bis *s-1* von 0 auf 1 setzt. Addiert man

$$\sum_{i=0}^{j-1} 2^{id+k-1} \text{ für } j > 0 \quad \text{Gl: 11-9}$$

auf die Z-Adresse, sind alle Bits in Stufe *s-2-j* bis *s-1* auf 1 gesetzt. Für die Bits $\alpha_{[s-1-j],(i)}$ in Stufe s-1-j, für die i < k gilt, ist der Wert jeweils 1. Addiert man nun den Wert 1 auf die Z-Adresse, ist die Sweep-Hyperebene erreicht. Somit ergibt sich folgende Formel:

$$\begin{cases} 1 & \text{für } j = 0 \\ 1 + \sum_{i=0}^{j-1} 2^{id+k-1} & \text{für } j > 0 \end{cases} \qquad \text{Gl: 11-10}$$

Der maximale Z-Abstand entsteht, wenn man alle Bits „rechts“ von $\alpha_{[s-1-j],(k)}$ auf 0 setzt.

$$\alpha_{[0],d} \ldots \alpha_{[0],(k+1)}\, 0\, \alpha_{[0],(k-1)} \ldots \alpha_{[0],(1)} \ldots \underbrace{\alpha_{[s-j],(d)} \ldots \alpha_{[s-j],(k+1)}\, \overbrace{0\,0\ldots0}^{k} \ldots 0}_{\text{Stufe } s-1-j} \underbrace{0 \ldots 0\, \overbrace{0\,0\ldots0}^{k}}_{\text{Stufe s-1}} \qquad \text{Gl: 11-11}$$

Somit ergibt sich für den maximalen Z-Abstand

$$2^{jd+k-1} \qquad \text{Gl: 11-12}$$

Eine Anwendung dieser Formel ist für den 2-dimensionalen Fall in Tabelle 11-1 aufgelistet.

Sweep-Hy.	Sortierdimension k = 1		Sortierdimension k =2	
$\Psi_{k,1}$	min 1	max 1	min 1	max 2
$\Psi_{k,2}$	min 2	max 4	min 3	max 8
$\Psi_{k,4}$	min 6	max 16	min 11	max 32
$\Psi_{k,8}$	min 22	max 64	min 43	max 128
$\Psi_{k,16}$	min 86	max 128	min 171	max 512
$\Psi_{k,m}\ m = 2^j$	min $1 + \sum_{i=0}^{j-1} 2^{i \cdot 2}$	max $2^{j \cdot 2}$	min $1 + \sum_{i=0}^{j-1} 2^{i \cdot 2+1}$	max $2^{j \cdot 2+1}$

Tabelle 11-1: Minimaler und Maximaler Z-Abstand für den 2-dimensionalen Fall

Der maximale Z-Abstand ist somit eine Potenzfunktion der Form $y = x^d$ mit $x = 2^j$, $j \in \mathbb{N}$. Der minimale Z-Abstand ist eine Summe von Potenzfunktionen. Betrachten wir zunächst die Basis. Es wird angenommen, dass eine Gleichverteilung der Daten vorliegt. Da die Potenzfunktion nach dem Monotoniegesetz der Basis monoton mit

$$dx^{d-1} \qquad \text{Gl: 11-13}$$

steigt (sieheTabelle 11-1), stellt der Graph somit eine Parabel *d*-ter Ordnung dar (siehe Abbildung 11-3).

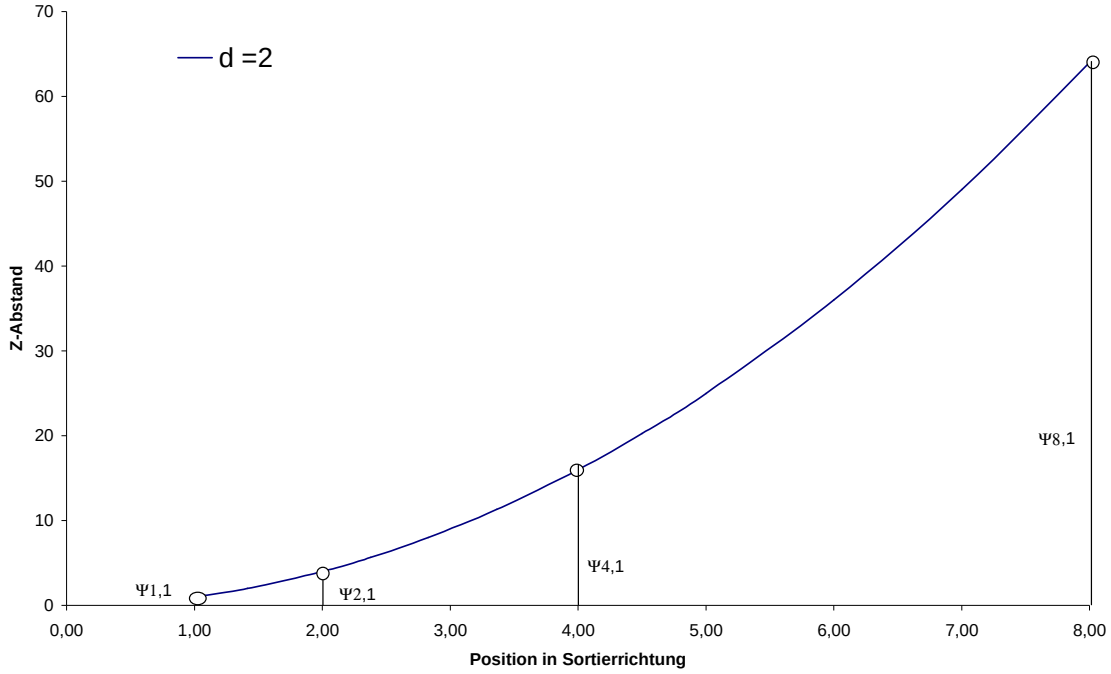


Abbildung 11-3: Z-Abstand von $\Psi_{0,1}$ zu $\Psi_{m,1}$

Geht man von gleichverteilten Daten aus und einer Kapazität κ einer Seite *p*, ergibt sich, dass mehr Regionen existieren, die die Sweep-Hyperebene $\Psi_{k,1}$ schneiden als $\Psi_{k,2}$. Somit ist die Z-Kurve bezüglich der T-Ordnung relativ lokal.

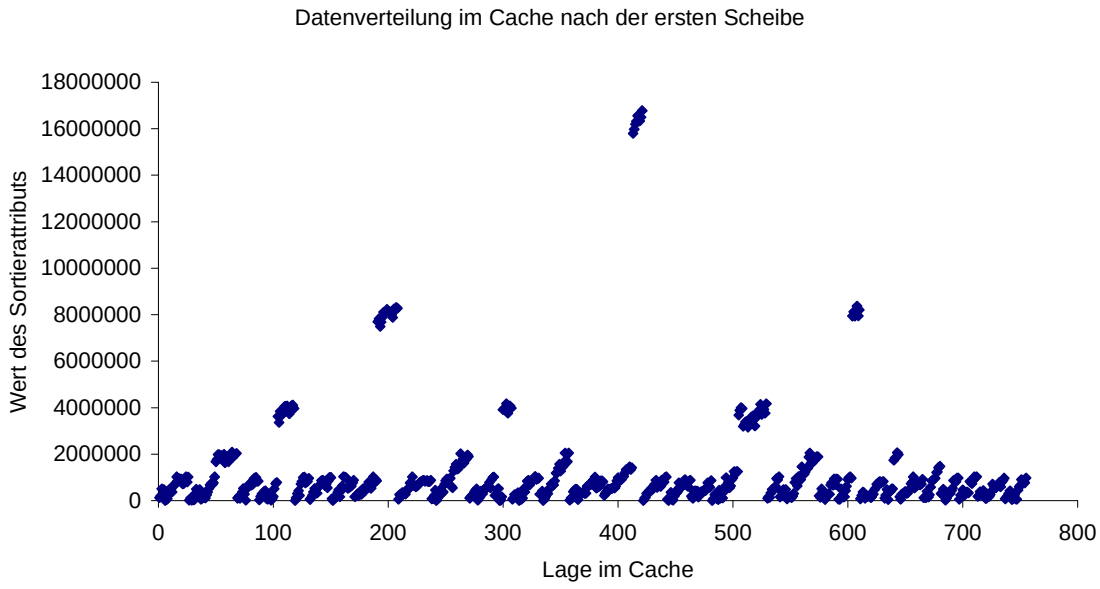


Abbildung 11-4: Datenverteilung des Caches nach der ersten Scheibe für den 2-dimensionalen Fall

Abbildung 11-4 zeigt die Datenverteilung des Caches nach dem Lesen der ersten Scheibe. Hierbei wurden die Tupel in eine Liste eingetragen. Der Graph zeigt eindeutig, dass die meisten Daten relativ nahe bei der Sweep-Hyperebene $\Psi_{1,0}$ liegen und somit obige Formel bestätigen.

Nach dem Monotoniegesetz bzgl. des Exponenten für Potenzen gilt:

$$d_1 < d_2 \Rightarrow x^{d_1} < x^{d_2} \qquad \text{Gl: 11-14}$$

Somit ist die Lokalität bei Basisräumen mit mehr Dimensionen größer als bei solchen mit wenig Dimensionen.

11.1.2 Sprünge aus der ersten Sweep-Hyperebene

Abbildung 11-4 zeigt noch ein weiteres Phänomen, das durch Sprünge der Z-Kurve verursacht wird, da die Z-Kurve nicht stetig ist. Dies sind die einzelnen Datencluster, wie zum Beispiel die zwei Datencluster, die ca. den Wert 8.000.000 in der Sortierdimension besitzen. Technisch gesehen kommt es immer dann zu einem Sprung in der Z-Kurve, sobald ein Überlauf entsteht. Ein Überlauf an Bit-Position *i* entsteht nur dann, wenn alle Bits von 0 bis *i* den Wert 1 haben und die Z-Adresse um 1 erhöht wird. Dann springen alle Bits von 0 bis *i* auf den Wert 0 um und die Bit-Position *i*+1 wird auf den Wert 1 gesetzt unter der Voraussetzung, dass Bit *i*+1 den Wert 0 hat. Sonst wird der Überlauf weiter propagiert, bis das erste Bit gefunden ist, dass den Wert 0 besitzt.

Beispiel: 11-1:
Gegeben sei ein 2-dimensionaler Basisraum Ω mit einer 8x8 Auflösung und der Z-Adresse $\alpha = 31_Z$. Die Z-Adresse hat folgende Binärdarstellung:

$$\alpha = 01.11.11$$

Addiert man auf die Z-Adresse α den Wert 1 kommt es zu einem Überlauf auf den Bit mit dem niedrigsten Wert. Der Überlauf wird bis auf Stufe 0 propagiert, da alle Bits in Stufe 2 und Stufe 1 den Wert 1 haben. Das Bit $a_{[0],1}$ hat auch den Wert 1, so dass das Bit $a_{[0],2}$ von 0 auf den Wert 1 gesetzt wird. Man erhält folgende binäre Darstellung:

10.00.00

Da das Bit $a_{[0],1}$ von 1 auf 0 gesetzt und das Bit $a_{[0],2}$ von 0 auf 1 gesetzt worden ist, haben wir einen Sprung über die Stufenlinie-1,0 und Stufenlinie-2,1. Der Sprung ist in Abbildung 11-5 links dargestellt.

Da es sich hier um den Tetris-Algorithmus handelt, wird das Verhalten der Z-Kurve bezgl. der Sweep-Hyperebene betrachtet. Zunächst wird der Begriff Sweepline-Sprung formal eingeführt. Ohne Beschränkung der Allgemeinheit betrachten wir hier nur die aufsteigende Sortierung.

Definition 11-3: Sweepline-Sprung

Gegeben sind eine Sweep-Hyperebene $\Psi_{k,c}$ und zwei Z-Adressen α und β. α und β sind ein *Sweepline-Sprung* $\langle \alpha, \beta \rangle$ genau dann, wenn gilt:

$$Z^{-1}(\alpha) \in \Psi_{k,c} \wedge Z^{-1}(\beta) \notin \Psi_{k,c} \wedge \text{extract}(\beta,k) > c \wedge |\alpha - \beta| = 1$$

Abbildung 11-5 zeigt zwei Sweepline Sprünge bezgl. der Sweep-Hyperebene $\Psi_{1,0}$. Hierbei handelt es sich um zwei Arten von Sprüngen. Der *Sweepline-Sprung* ⟨31:32⟩ ist ein Sprung von außerhalb der Sweep-Hyperebene in die Sweep-Hyperebene hinein, d.h.

extract(31,1) > extract(32,1).
⇔ 7>0

Das Entscheidende ist somit der Wert der Sortierdimension. Obwohl die Z-Adresse 31 der Vorgänger zur Z-Adresse 32 ist, besitzt die Z-Adresse 31 in der Sortierdimension einen höheren Wert als die Z-Adresse 32. Dieser Sprung wird als „ *Reinsprung*“ bezeichnet

Der zweite Fall ist in Abbildung 11-5 rechts dargestellt. Der Sprung ⟨8:9⟩ springt von der Sweep-Hyperebene heraus, d.h.

extract(8,1) < extract(9,1).
⇔ 0 < 1

Dieser Fall wird als „Raussprung“ bezeichnet.

Zu beachten ist jedoch, dass hier der Fall der aufsteigenden Sortierung betrachtet wird. Für die absteigende Sortierung müssen das Kleiner- und Größer-Zeichen ausgetauscht werden.

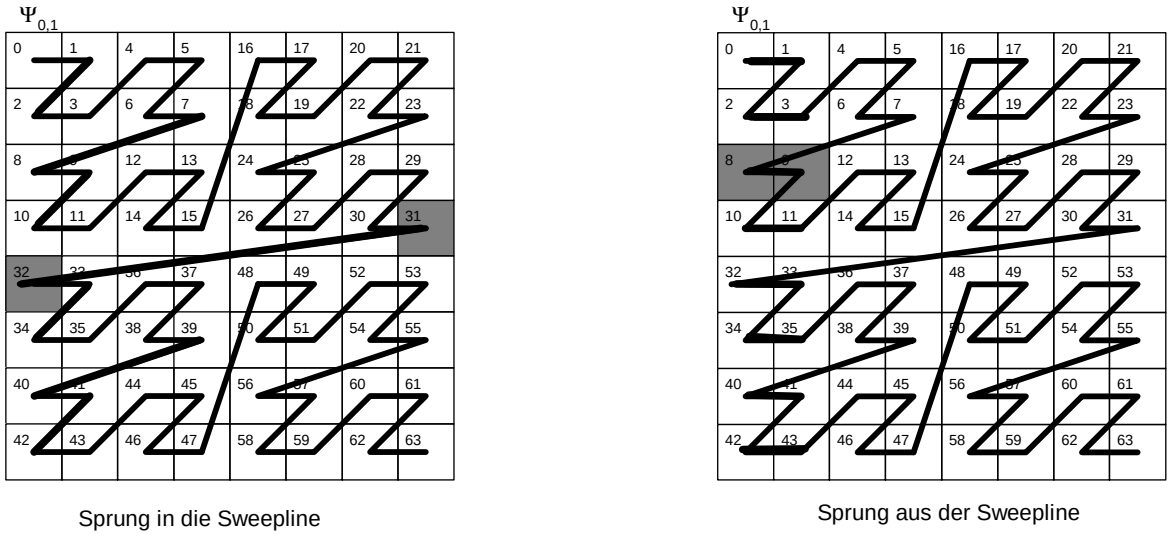


Abbildung 11-5

Es wird zunächst der Fall des „*Reinsprungs*“ betrachtet (siehe Abbildung 11-5).

Reinsprung

Ein *Reinsprung* $\langle \alpha:\beta \rangle$ entsteht genau dann, wenn der Wert der Sortierdimension der Z-Adresse α größer als der Wert der Sortierdimension β ist, wobei $Z^{-1}(\beta)$ Element der Sweep-Hyperebene $\Psi_{k,c}$ ist. Es wird wieder die $\Psi_{1,0}$ betrachtet.

Für β gilt somit:

$\beta_{[j],(i)} = 0$ falls $i = k$ und $i \in \{1, \ldots, d\}$

Um nun einen Sprung über die Stufenlinie-k,i zu erhalten muss ein Unterlauf auf der Stufe i erzeugt werden, der das Sortierdimensionsbit $\beta_{[i],(k)}$ von 0 auf 1 setzt. Ein Unterlauf entsteht genau dann, wenn alle Bits in der Z-Adresse, deren Wertigkeit kleiner als Bit $\beta_{[i],(k)}$ ist, den Wert 0 haben. Es wir zunächst der Unterlauf auf Stufe 0 betrachtet. Man erhält folgende Struktur:

$$\overbrace{\beta_{[0],d} \ldots \beta_{[0],(k+1)}}^{\text{Ein Bit ist} \neq 0} \quad \underbrace{0 \ldots 0}_{\beta_{[0],(k)} \text{bis } \beta_{[0],(1)}} \quad \underbrace{0 \ldots 0}_{\text{Stufe 1 bis s-1}} \qquad \text{Gl: 11-15}$$

Subtrahiert man jetzt den Wert 1 von der Z-Adresse β, werden alle Bits von $\beta_{[0],(k)}$ bis $\beta_{[s-1],(1)}$ durch den Unterlauf auf 1 gesetzt. Da nach Voraussetzung ein Bit von Bit $\beta_{[0],(d)}$ bis $\beta_{[0],(k+1)}$ ungleich 0 ist, wird der Unterlauf gestoppt und man erhält eine Z-Adresse α. Da das Bit $\alpha_{[0],(k)}$ den Wert 1 hat und $\beta_{[0],(k)}$ den Wert 0 hat, ist $\langle \alpha:\beta \rangle$ ein Sweepline-Sprung über die Stufenlinie-k,0. Die Z-Adresse α hat folgende Struktur:

$$\overbrace{\alpha_{[0],d} \ldots \alpha_{[0],(k+1)}}^{\text{Ein Bit ist von 1 auf 0 gesetzt worden}} \quad \underbrace{1 \ldots 1}_{\alpha_{[0],(k)} \text{bis } \alpha_{[0],(1)}} \quad \underbrace{1 \ldots 1}_{\text{Stufe 1 bis s-1}} \qquad \text{Gl: 11-16}$$

Um die Anzahl der Sprünge zu bestimmen, die die Stufenlinie-k,0 überschreiten, müssen die freien Bits in Gl: 11-16 gezählt werden. Somit ergibt sich für die Anzahl der Sprünge $SP_{\Psi(0,k),T(0,k)}$ von der Sweep-Hyperebene $\Psi_{0,k}$ über die Stufenlinie-k,0 folgende Formel:

$$SP_{\Psi(0,k),S(k,0)} = 2^{d-k} - 1 \qquad \text{Gl: 11-17}$$

S(k,0) ist eine Kurzschreibweise für die Stufenlinie-k,0 und $\Psi(k,0)$ für die Sweep-Hyperebene $\Psi_{k,0}$. Die Subtraktion um den Wert 1 ergibt sich daraus, dass mindestens ein Bit von $a_{[0],(k)}$ bis $a_{[s-1],(1)}$ den Wert 1 besitzen muss.

Betrachtet man nun die Anzahl der Sprünge $SP_{\Psi(k,0),S(1,k)}$ von der Sweep-Hyperebene $\Psi_{k,0}$ über die Stufenlinie-k,1, erhält man folgende Z-Adressen Struktur:

$$\underbrace{\overbrace{\alpha_{[0],d} \ldots \alpha_{[0],(k+1)}}^{d-k \text{ Bits}} \overbrace{0}^{k} \overbrace{\alpha_{[0],(k-1)} \ldots \alpha_{[0],(1)} \alpha_{[1],(d)} \ldots \alpha_{[1],(k+1)}}^{d\text{-}1 \text{ Bits}}}_{\text{Mindestens ein Bit muss den Wert 1 besitzen}} \quad \text{Gl: 11-18}$$

$$\underbrace{0 \ldots 0}_{\alpha_{[1],(k)} \text{bis } \alpha_{[1],(1)}} \quad \underbrace{0 \ldots 0}_{\text{Stufe 2bis s-1}}$$

Somit ergibt sich für die Anzahl der Sprünge $SP_{\Psi(k,0),S(1,k)}$ von der Sweep-Hyperebene $\Psi_{k,0}$ über die Stufenlinie-k,1 folgende Formel:

$$\begin{aligned} SP_{\Psi(k,0),S(k,1)} &= 2^{d-k} \cdot 2^{k-1} \cdot 2^{d-k-1} - 1 \\ &= 2^{(d-k)+(k-1)+(d-k)} - 1 \\ &= 2^{d-k+d-1} - 1 \end{aligned} \quad \text{Gl: 11-19}$$

In dieser Formel sind Sprünge über die Stufenlinie-k,0 und die Stufenlinie-k,1 enthalten.

Für den allgemeinen Fall ergibt sich somit:

$$SP_{\Psi(k,0),S(k,i)} = \overbrace{2^{d-k}}^{\text{Stufe 0}} \cdot \overbrace{2^{i(d-1)}}^{\text{Stufe 1 bis } i} \overbrace{-1}^{\text{Mindestens ein Bit muss den Wert 1 haben}} \quad \text{Gl: 11-20}$$

Die Anzahl der exklusiven Sprünge $SPE_{\Psi(k,0),S(1,k)}$ über die Stufenlinie-k,i ohne die Sprünge über die Stufenlinie-k,0 bis Stufenlinie-k,(i-1) ist:

$$SPE_{\Psi(k,0),S(i,k)} = \begin{cases} SP_{\Psi(k,0),S(k,0)} & i = 0 \\ SP_{\Psi(k,0),S(k,i)} - SP_{\Psi(k,0),S(k,i-1)} & i \in \{1,\ldots,s-1\} \end{cases} \quad \text{Gl: 11-21}$$

Für den 2-dimensionalen Fall ist in Tabelle 11-2 die Anzahl der exklusiven Sprünge aufgelistet.

	k =1	k = 2
$SPE_{\Psi(k,0),S(k,0)}$	1	0

$SPE_{\Psi(k,0),S(k,1)}$	2	1
$SPE_{\Psi(k,0),S(k,2)}$	4	2
$SPE_{\Psi(k,0),S(k,i)}$	2^i	2^{i-1}

Tabelle 11-2

Abbildung 11-6 zeigt für einen 2-dimensionalen Basisraum Ω mit einer Auflösung von 8x8 Punkten mit dem Ursprung rechts oben die „Reinsprünge“ aus der Sweep-Hyperebene $\Psi_{k,0}$

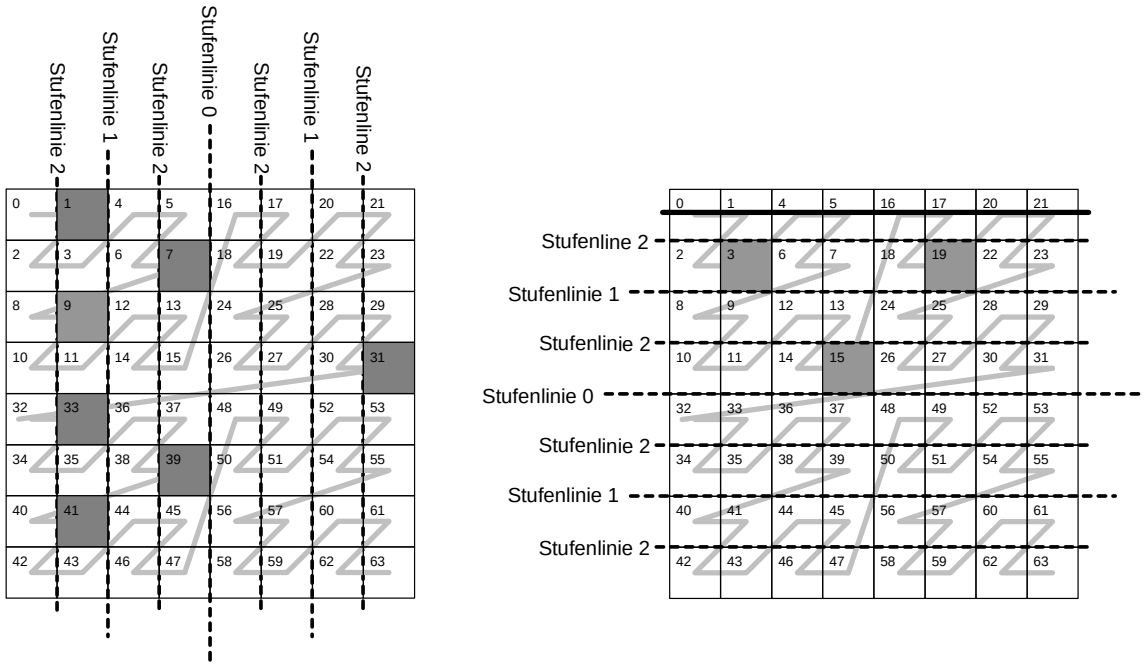


Abbildung 11-6

Wie man deutlich erkennen kann, existiert für die Sweep-Hyperebene $\Psi_{0,2}$ kein Sweepline-Sprung über die Stufenlinie-2,0. Dies entspricht genau der Formel in Gl: 11-21.

Die wesentliche Erkenntnis ist somit, dass die Anzahl der Sprünge über die Stufenlinie-i, mit kleiner werdendem i, exponentiell abnehmen. Somit existieren wenig Sprünge, die in der Sortierdimension einen großen Abstand haben und exponentiell viele, die einen kleinen Abstand haben.

Raussprung

Es wird nun der zweite Fall betrachtet, der Sprung aus der aktuellen Sweep-Hyperebene. Ein „Raussprung“ $\langle \alpha, \beta \rangle$ entsteht genau dann, wenn der Wert der Sortierdimension der Z-Adresse α kleiner als der Wert der Sortierdimension β ist, wobei $Z^{-1}(\alpha)$ Element der Sweep-Hyperebene $\Psi_{c,k}$ ist. Es wird wieder die Sweep-Hyperebene $\Psi_{0,1}$ betrachtet.

Für α gilt somit:

$\alpha_{[j],(i)} = 0$ falls $i = k$ und $i \in \{1, \ldots, d\}$

Damit ein Sweepline-Sprung entsteht muss das Bit $\alpha_{[s-i],(k)}$ vom Wert 0 auf den Wert 1 übergehen. Dies ist nur dann möglich, wenn die niederwertigen Bits $\alpha_{[s-i],(k-1)}$ bis $\alpha_{[s-i],(1)}$ den Wert 1 besitzen. Somit erhält man folgende Z-Adressen-Struktur:

$$\alpha_{[0],d} \ldots \alpha_{[0],(k+1)}\, 0\, \alpha_{[0],(k-1)} \ldots \alpha_{[0],(1)} \ldots \underbrace{\alpha_{[s-j],(d)} \ldots \alpha_{[s-j],(k+1)}\, \overbrace{0}^{k}\, 0 \ldots 0}_{\text{Stufe } s-1-j} \ldots \underbrace{0 \ldots 0\, \overbrace{0}^{k}\, 0 \ldots 0}_{\text{Stufe s-1}} \qquad \text{Gl: 11-22}$$

$$\underbrace{\alpha_{[0],d} \ldots \alpha_{[0],(k+1)}\, 0\, \alpha_{[0],(k-1)} \ldots \alpha_{[0],(1)}}_{\text{Stufe 0}} \ldots \underbrace{\alpha_{[s-j],(d)} \ldots \alpha_{[s-1],(k+1)}\, \overbrace{0}^{k}\, 1 \ldots 11}_{\text{Stufe s-1}}$$

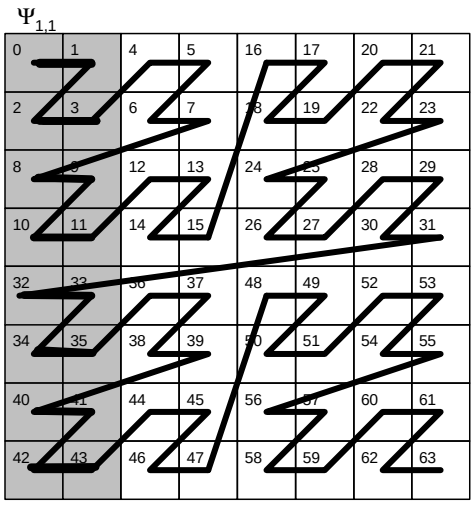
Die Anzahl der „Raussprünge“ *SPP* aus der Sweep-Hyperebene $\Psi_{0,1}$ ergeben sich aus der Anzahl der freien Bits. Pro Stufe existieren *d* Bits. Hiervon ist das Bit der Sortierdimension bereits gebunden. Somit hat man $d - 1$ Bits pro Stufe, die frei sind. Dies gilt für Stufe 0 bis $s - 2$. In Stufe $s - 1$ existieren noch $d - k$ freie Bits. Es existieren somit

$$SPP = 2^{(s-1)\cdot(d-1)+d-k} \qquad \text{Gl: 11-23}$$

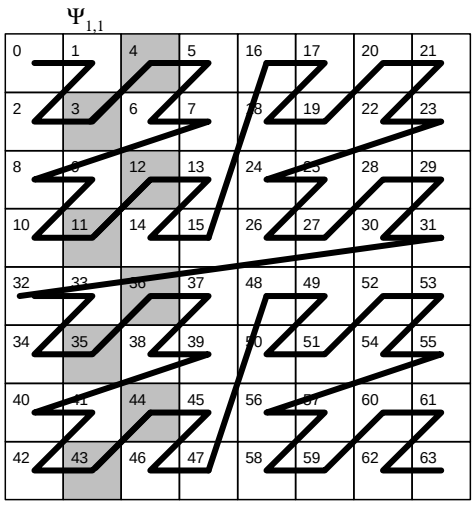
„Raussprünge“. Für einen 2-dimensionalen Basisraum Ω mit einer Auflösung $r = 8$ erhält man für die Anzahl der Stufen s genau 3 Stufen. Für die Sweep-Hyperebene $\Psi_{0,1}$ ergeben sich somit:

$$SPP = 2^{(s-1)\cdot(d-1)+d-k} = 2^{(3-1)\cdot(2-1)+(2-1)} = 2^{2+1} = 2^3 = 8 \qquad \text{Gl: 11-24}$$

„Raussprünge“. Die „Raussprünge“ sind in Abbildung 11-7.rechts zu sehen.



Sprung aus die Sweepline $\Psi_{1,0}$



Sprung aus die Sweepline $\Psi_{1,1}$

Abbildung 11-7

11.1.3 Sprünge aus der i-ten Sweep-Hyperebene

Zunächst wird wieder der Fall des *„Reinsprungs“* betrachtet und verallgemeinert.

Reinsprung

Verallgemeinert man die Formel aus Gl: 11-21 für beliebige Sweep-Hyperebenen $\Psi_{k,m}$, geht man wie folgt vor: Man sucht das erste Bit in der Sortierdimension, das den Wert 1 besitzt. Hierbei geht man von der höchsten Stufe s-1 los und steigt dann die Stufen herab bis man Stufe 0 erreicht. Die Stufe, die das erste Bit mit dem Wert 1 besitzt, sei *getOne*(*n*) = *m*. Da die Sprünge der Z-Kurve durch ihre rekursive Unterteilung entstehen, bestimmt *m* den Sprungraum, d.h. den Raum, der durch einen Sprung in Sortierrichtung erreicht wird. Ist *m* nicht definiert, d.h. alle Bits sind 0 in der Sortierdimension, giltGl: 11-21, da es sich um die Sweep-Hyperebene $\Psi_{k,0}$ handelt. Ist *m* = 0 wird der Basisraum um die Hälfte reduziert, so dass kein Sprung über die Stufenlinie-k,0 mehr existiert. Wegen der rekursiven Struktur existieren die übrigen Sprünge.

Ist *m* = 1 ist der Sprungraum ¼ des Basisraums. Somit existieren keine Sprünge über die Stufenlinie-k,0 und Stufenlinie-k,1. Wegen der rekursiven Struktur der Z-Kurve existieren die übrigen Sprünge. Somit ergibt sich für die allgemeine Formel:

$$SPE_{\psi(k,n)S(k,i)} = \begin{cases} 0 & i \le getStep(n) \\ SPE_{\psi(k,0)S(k,i)} & i > getStep(n) \end{cases} \qquad \text{Gl: 11-25}$$

Beispiel 11-2:

Gegeben sei ein 2-dimensionaler Basisraum Ω mit einem Ursprung links oben und einer Auflösung von 8 in jeder Dimension. Es werden die Sprünge der Sweep-Hyperebene $\Psi_{1,4}$ und $\Psi_{2,4}$ betrachtet. 4 hat die binäre Darstellung 100.

	k =1	k = 2
$SPE_{\Psi(k,4),S(k,0)}$	0	0
$SPE_{\Psi(k,4),S(k,1)}$	2	1
$SPE_{\Psi(k,4),S(k,2)}$	4	2

Tabelle 11-3

Man erhält *getStep*(4) = 0. Somit existiert kein Sprung über die Stufenlinie-k,0. Wendet man die Formel Gl: 11-26 an, erhält man das Ergebnis aus Tabelle 11-3. Die einzelnen Sprünge sind in Abbildung 11-8 dargestellt.

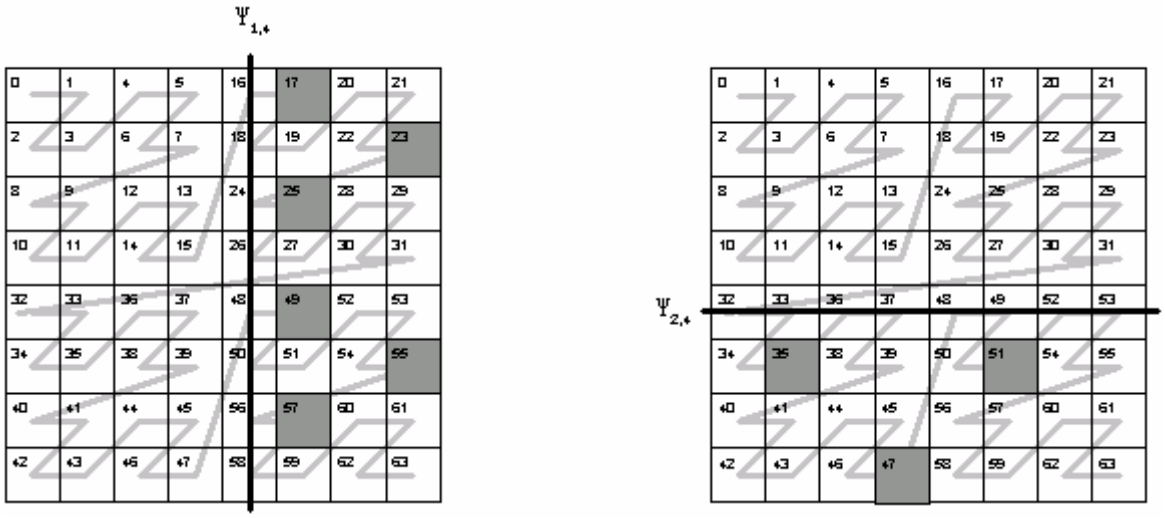


Abbildung 11-8: Reinsprünge in $\Psi_{1,4}$ und $\Psi_{2,4}$

Raussprung

In diesem Abschnitt wird nun die Formel aus Gl: 11-23 für den allgemeinen Fall präsentiert. Für beliebige Sweep-Hyperebenen erhält man folgende Formel:

$$SPP = 2^{(i)\cdot(d-1)+d-k} \qquad \text{Gl: 11-26}$$

i bezeichnet die höchste Stufe in der Z-Adresse, in der das Bit der Sortierdimension den Wert 0 besitzt. Betrachten wir zum Beispiel die Sweep-Hyperebene $\Psi_{1,1}$ für den 2-dimensionalen Fall mit einer Stufentiefe von s = 3. Die Sortierdimension hat den Wert 1. Binär ergibt sich

$$x_k = 001 \qquad \text{Gl: 11-27}$$

Somit ergibt sich für *i* der Wert 1 und man erhält:

$$SPP_i = 2^{i(d-1)+d-k} = 2^{1+1} = 4 \qquad \text{Gl: 11-28}$$

Die Sprünge sind in Abbildung 11-7 rechts zu sehen. Wie man aus Abbildung 11-7 erkennt, sind es „Raussprünge“ in die nächste Sweep-Hyperebene, das heißt es wird von der Sweep-Hyperebene $\Psi_{k,i}$ in die Sweep-Hyperebene $\Psi_{k,i+1}$ gesprungen. Diese Sprünge bezeichnen wir als lokal, da die Punkte bereits in der nächsten Sweep-Hyperebene verarbeitet werden.

11.1.4 Messung und Bewertung

Um das Modell aus den vorangegangenen Abschnitten durch Messungen zu bestätigen, wurden verschiedene Messungen durchgeführt, die die Datenverteilung der Caches des Tetris-Algorithmus darstellen.

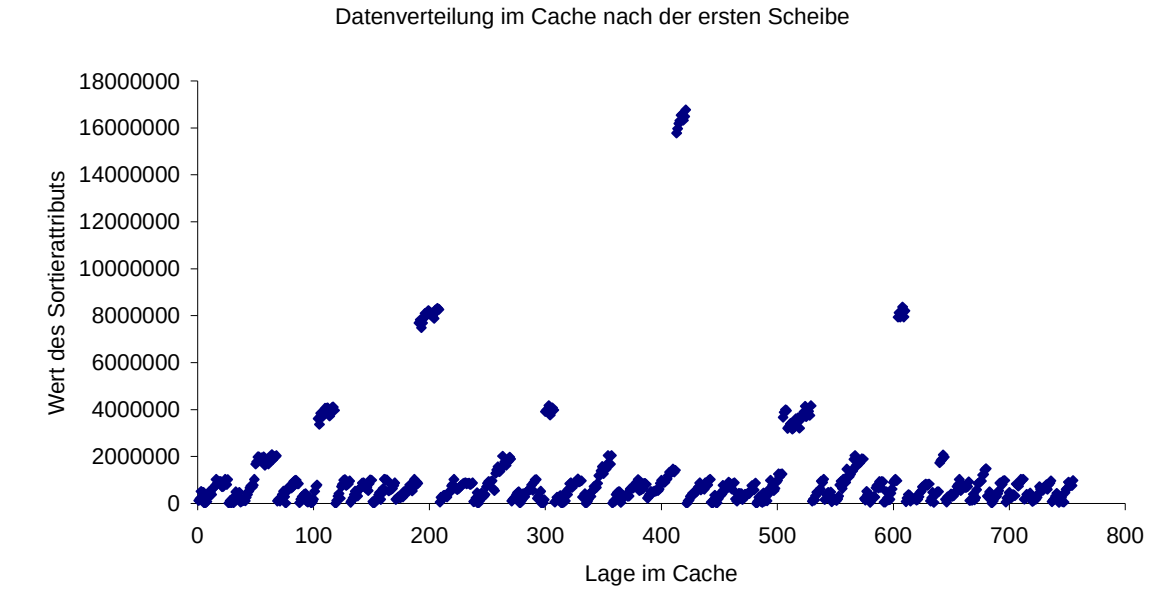


Abbildung 11-9

Für die Messung wird eine 2-dimensionale gleichverteilte Datenbank verwendet. Sie enthält 10.000 Tupel, die auf 494 Regionen verteilt sind. Der Wertebereich in jeder Dimension ist $[0,2^{24}\text{-}1]$. Eine Beschreibung der Datenbank ist in Tabelle 11-4 zu finden.

Name	Dimensionen	Wertebereich	Datenverteilung	Tupel	Seiten	Size MB
DB1	2	$[0,2^{24}\text{-}1]$	gleichverteilt	10000	494	1

Tabelle 11-4

Der Cache ist als Array organisiert und wird mit dem Quicksort Algorithmus sortiert. Es wurde eine Bereichsanfrage auf den gesamten Basisraum Ω ausgeführt. Die Bereichsanfrage wird mit dem Tetris-Algorithmus abgearbeitet. Die Sortierdimension ist x_1. Der Algorithmus erzeugt 59 Scheiben. Die erste Scheibe umfasst 36 Seiten. Abbildung 11-9 zeigt die Datenverteilung des Caches nach dem Einfügen der ersten Scheibe. Die Position im Cache reflektiert die Lesereihenfolge der Tupel aus dem Sekundärspeicher. Man erkennt, dass genau eine Region existiert, die über die Stufenlinie-1,0 springt. Hierbei handelt es sich um das Cluster, das nahe bei dem Wert $2^{24} - 1 = 166.777.215$ liegt. Dies ist der maximale Wert in Sortierrichtung. Diese Tupel bleiben somit bis zum Ende des Algorithmus im Cache. Des weiteren existieren zwei Sprünge über die Stufenlini-1,1. Sie besitzen den Wert $2^{23}\text{-}1 = 8388607$. Für die Stufenlinie-1,2 existieren nur 3 Cluster. Nach Gl: 11-26 würden maximal 4 Cluster erwartet. Die Formel gibt die maximale Anzahl von Sprüngen an, die möglich sind. Die maximale Anzahl von Sprüngen wird genau dann nicht erreicht, wenn der Split-Algorithmus die Sprungregion in zwei Regionen aufteilt oder in diesem Bereich keine Daten existieren. Diese Messung zeigt, dass Gl: 11-25 eine obere

Grenze für die Anzahl der Sprünge darstellt. Man erkennt auch deutlich, dass die Anzahl der Sprünge exponentiell zunimmt je näher man der aktuellen Sweep-Hyperebene kommt.

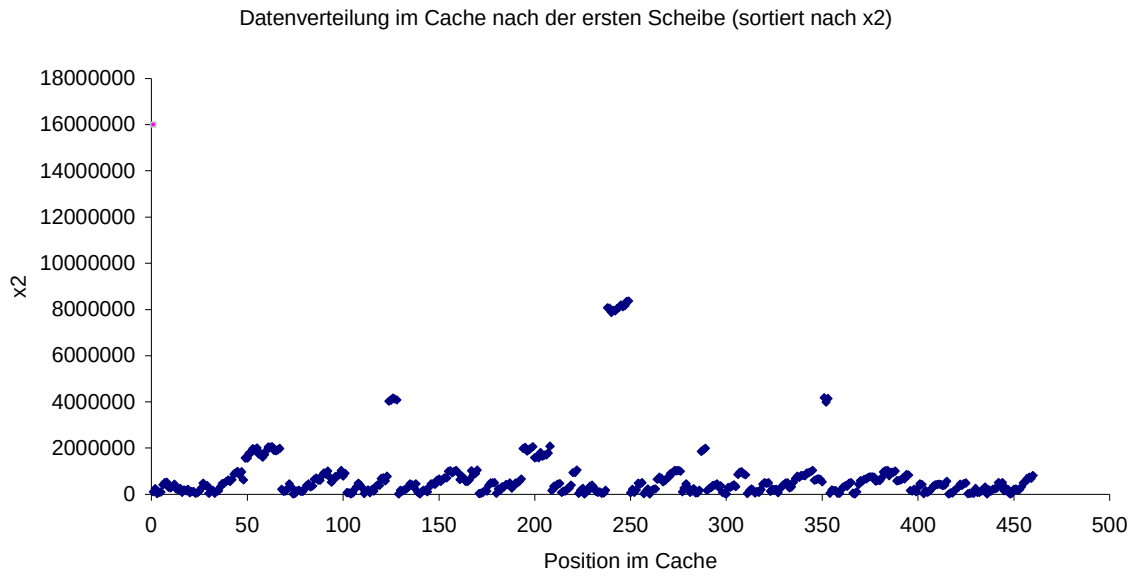


Abbildung 11-10

Abbildung 11-10 zeigt die entsprechenden Messungen für Sortierdimension x_2. Allgemein erkennt man, dass es weniger weite Sprünge gibt, als für die Sortierdimension x_1. Dies entspricht Gl: 11-25. Somit existiert zum Beispiel kein Sprung über die Stufenlinie-2,0.

Abbildung 11-11 zeigt die Datenverteilung nachdem die Daten der zweiten Scheibe aus dem UB-Baum geladen, jedoch noch nicht sortiert worden sind. Es können zwei Bereiche unterschieden werden. Der Bereich der alten Tupel n_{old}, die bereits durch die erste Scheibe in den Cache geladen worden sind, aber noch nicht aus dem Cache entfernt worden sind, da sie die aktuelle Sweep-Hyperebene schneiden. Sie liegen auf Position 1 bis 760. Der andere Bereich besteht aus den neuen Tupeln der zweiten Sweep-Hyperebene. Sie liegen auf Position 761 bis 780. Im Bereich 1 bis 760 erkennt man, dass die Anzahl der Tupel mit einem großen Abstand zur aktuellen Sweep-Hyperebene exponentiell abnimmt. Um die 494 Seiten in einem 2-dimensionalen Basisraum Ω zu verwalten, wird eine Stufentiefe von 5 benötigt. Da die Tupel aus der ersten Sweep-Hyperebene $\Psi_{0,1}$ stammen, können „Reinsprünge“, die die Stufenlinie-1,0, Stufenlinie-1,1, Stufenlinie-1,2 überspringen, existieren. Die Tupel auf Position 0 bis 10 gehören zu der Sprungregion, die die Stufenlienie-1,0 überspringt. Diese Tupel werden erst in der 59. Scheibe verarbeitet.

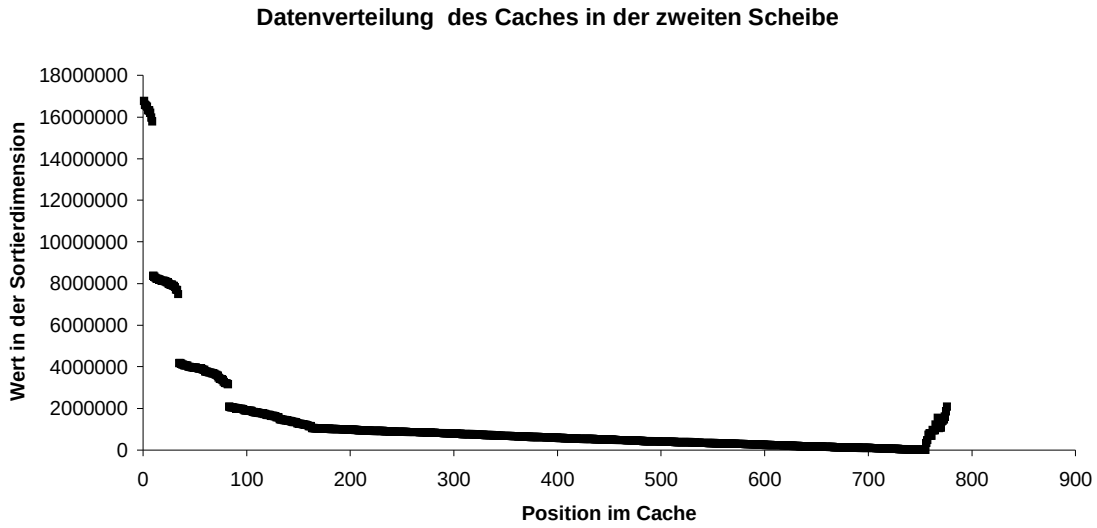


Abbildung 11-11

Die Tupel von Position 11 bis 34 gehören zu den Sprungregionen, die durch einen Sprung über die Stufenlinie-1,1 entstehen. Die Tupel werden spätestens in der 32. Scheibe verarbeitet, da die 33. Scheibe, die erste Scheibe nach der Stufenlinie-1,0 ist. Die Anzahl der existierenden Sprungregionen entspricht der Gl: 11-25, wodurch das Modell bestätigt wird.

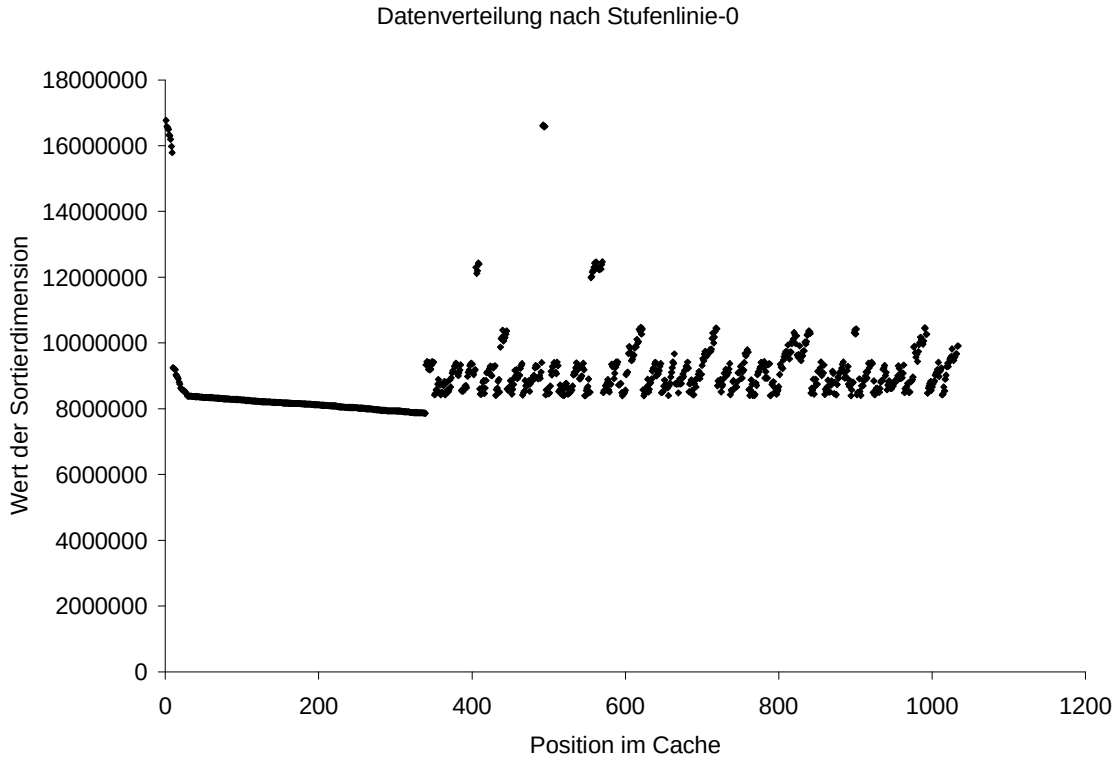


Abbildung 11-12

Abbildung 11-12 zeigt die Datenverteilung nach der Stufenlinie-0. Der Cache unterteilt sich hierbei in zwei Teile. Der erste Teil besteht aus Tupeln, die bereits im Cache lagen (Position 0 bis 370) und der zweite Teil aus den Tupeln, die neu in den Cache eingefügt werden. Der erste Teil besteht aus zwei Clustern, dem Cluster, das durch die Verarbeitung der ersten Scheibe erzeugt worden ist (Sprung über die Stufenlinie-0, „Reinsprung“) und dem Cluster, das durch die Raussprünge über die Stufenlinie-0 entsteht. Der Teil mit den neu eingefügten Tupeln zeigt das erwartete Bild. Es existieren Sprünge über die Stufenlinie-1,1 sowie über die Stufenlinie-1,2 usw. Dies entspricht genau der Gleichung Gl: 11-25.

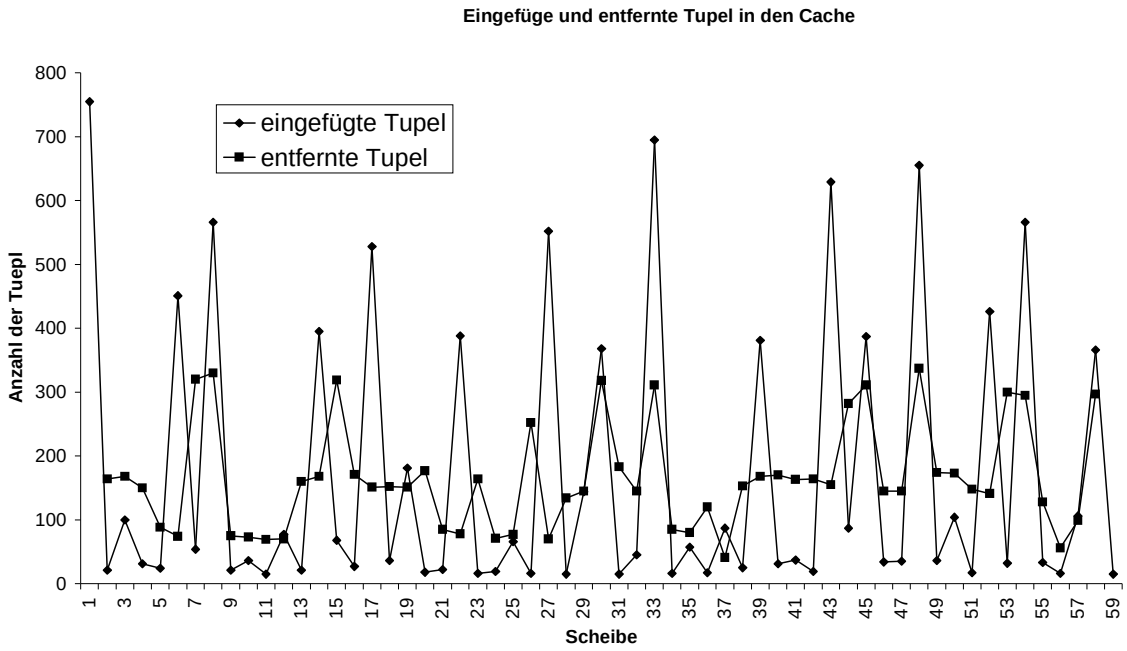


Abbildung 11-13: Einfügen und Entfernen von Tupeln aus dem Cache

Abbildung 11-13 zeigt die Anzahl der Tupel, die pro Scheibe in den Cache eingefügt bzw. entfernt werden. Grundsätzlich erkennt man einen alternierenden Verlauf zwischen wenigen und vielen Tupeln. Man erkennt eindeutig, dass die erste Scheibe die meisten Tupel in den Cache einfügt. Dies liegt zum einen daran, dass für die erste Scheibe, die durch $\Psi_{1,0}$ bestimmt wird, Sprünge über alle Stufenlinien existieren. Daneben existieren für $\Psi_{1,0}$ nach Gl: 11-26 die maximale Anzahl von Raussprüngen. Scheibe 33 ist die Scheibe nach der Stufenlinie-1,0. Somit existiert kein Sprung für Stufenlinie-1,0. Alle anderen Sprünge sind möglich. Deshalb werden aus dieser Scheibe fast so viele Tupel in den Cache eingefügt wie in Scheibe 1. Allgemein kann man feststellen, dass mehr Tupel aus einer Scheibe in den Cache eingefügt werden, wenn die Scheibe nach einer Stufenlinie mit kleiner Stufennummer gelesen wird. Das Entfernen der Tupeln aus dem Cache entspricht im Wesentlichen dem Einfügen.

11.1.5 Zusammenfassung

In diesem Kapitel wurde der Aufbau des Cache untersucht, der durch den Tetris-Algorithmus erzeugt wird. Ziel der Untersuchung war es, herauszufinden, wie lange ein Tupel im Cache verbleibt, bis es wieder entfernt werden kann und wie wahrscheinlich es ist, dass dieser Fall eintritt. Ein Tupel kann aus dem Cache entfernt werden genau dann, wenn das Tupel die aktuelle Sweep-Hyperebene schneidet. Als Metrik wurde der Z-Abstand eingeführt. Für diese Metrik wurde ein Modell entwickelt. Das Modell zeigt, dass der Z-Abstand wie eine Parabel für jede Stufenlinie in Sortierrichtung wächst. Da die Z-Kurve nicht stetig ist, kann es jedoch auch zu Sprüngen kommen. Dies wiederum führt dazu, dass zwei Punkte in Sortierrichtung den maximalen Abstand besitzen, jedoch auf der Z-Kurve Nachbarn sind. Diese Regionen sind somit Problemregionen, da die Daten sehr lange im Speicher gehalten werden müssen. Die Anzahl der Sprünge, die niedrige Stufen überspringen, nimmt exponentiell ab. Somit ist die Z-Kurve relativ lokal und es existieren nur wenig Regionen, deren Tupel lange im Arbeitsspeicher gehalten werden müssen. Es konnte auch gezeigt werden, dass das Sortierattribut einen signifikanten Einfluss auf die Datenverteilung des Caches hat. Allgemein kann man sagen, dass es weniger große als kleine Sprünge existieren. Das Modell wurde durch Messungen bestätigt.

11.2 Internes Sortieren

Für das Sortieren des Caches werden jetzt zwei Sortierverfahren verglichen. Quicksort [Knu98v3] und Heapsort. [Knu98v3]. Für eine genaue Beschreibung wird auf [Knu98v3] verwiesen. Es wird hier nur die Anzahl der Vergleichsoperationen betrachtet. Der wesentliche Unterschied zwischen den beiden Verfahren besteht darin, dass Quicksort den Cache vollständig neu sortiert und der Heapsort durch seine Heap-Struktur die neuen Tupel in den Cache, der normalerweise nicht leer ist, einfügt und so Vergleichs-Operationen spart.

11.2.1 Quicksort

Quicksort basiert auf dem Prinzip “divide-and-conquer”. Hierbei wird die zu sortierende Menge bezüglich eines Partitionselements aufgeteilt. Die eine Menge enthält alle Elemente, die kleiner als das Partitionselement sind, die andere Menge alle Elemente, die größer oder gleich sind. Der Algorithmus wird rekursiv wieder auf die Teilmengen angewendet, bis die Menge vollständig sortiert ist. Da alle Vergleiche gegen dasselbe Partitionselement ausgeführt werden, kann der Wert im Register bleiben und somit die innere Schleife des Quicksort-Algorithmus extrem effizient ausgeführt werden. Quicksort benötigt im Durchschnitt:

$$2 \cdot n \cdot \ln(n) \approx 1.39 \cdot n \log_2(n) \qquad \text{Gl: 11-29}$$

Vergleiche um eine Menge von *n* Elementen zu sortieren. Quicksort hat jedoch einen erheblichen Nachteil. Ist die Menge bereits sortiert, tritt der Worst-Case ein und Quicksort benötigt:

$$\frac{n^2}{2}$$ Gl: 11-30

Vergleichsoperationen[Knu98v3].

Um den Worst-Case zu verhindern, wird als Partitionselement der Median von drei Elementen berechnet. Diese Methode führt, neben der Reduzierung der Wahrscheinlichkeit des Worst-Case, zu einer Reduzierung der durchschnittlichen Laufzeit von ca. 5 % [Knu98v3]. Eine weitere Reduzierung der Laufzeit um 10% kann durch die Anwendung von „Sortieren durch direktes Einfügen“ auf kleine Teildateien erreicht werden [Knu98v3].

Es werden somit ca .

$$c_{quick} = 1.18 \cdot n \log_2(n)$$ Gl: 11-31

benötigt.

11.2.2 Messung

Für die Messungen wurde die in Tabelle 5-4 beschriebene Datenbank verwendet. Abbildung 11-14 zeigt die Sortierkosten mit Quicksort in Abhängigkeit von der Anzahl der Tupel. Es wurde eine *n lg (n)* Trennlinie eingetragen. Sie repräsentiert die durchschnittlichen Kosten für den Quicksort-Algorithmus. Wie man leicht aus Abbildung 11-14 erkennt, liegen die Sortierkosten relativ nahe an dem Durchschnittswert für Quicksort und somit nicht beim schlechtesten Fall von $O(n^2)$.

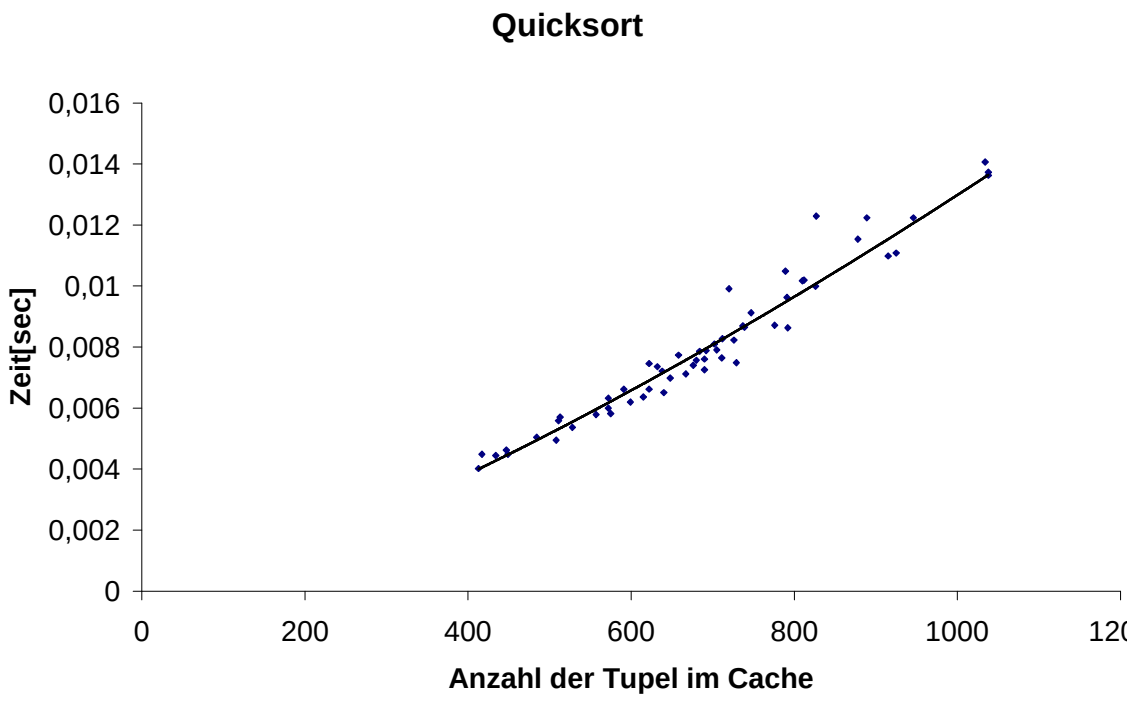


Abbildung 11-14

Somit zeigt die Verwendung der 3-Median-Strategie eindeutig seine Wirkung. Nach Gl: 11-31 benötigt man für das Sortieren mit Quicksort $1.18\, n \log_2 (n)$ Vergleichsoperationen. Die Zeit, die eine Vergleichoperation benötigt, bezeichnen wir als c-Faktor. Somit erhält man:

$$t_{quick} = c_{c-quick} \cdot 1.18 \cdot n \log_2 (n) \qquad \text{Gl: 11-32}$$

Abbildung 11-15 zeigt den *c*-Faktor $c_{c\text{-}quick}$ für unterschiedliche Cachegrößen. Sie entsprechen den jeweiligen Scheibengrößen. Im Durchschnitt liegt der c-Faktor bei ca. $1 * 10^{-6}$ s.

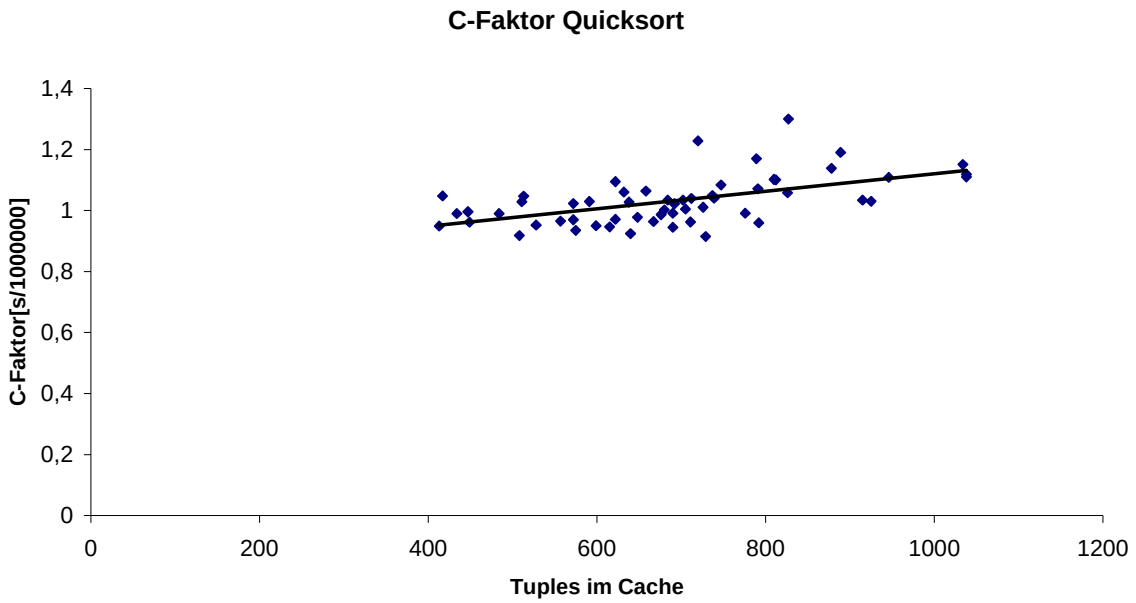


Abbildung 11-15

Abbildung 11-16 zeigt die CPU-Kosten der Cachverwaltung für die Verarbeitung der 59 Scheiben. Die CPU-Kosten wachsen linear zu der Anzahl der Scheiben. Da es sich hier um gleichverteilte Daten handelt, liegt ungefähr dieselbe Anzahl von Daten in jeder Scheibe, so dass der Aufwand pro Scheibe für das Einfügen, Entfernen und Sortieren konstant ist.

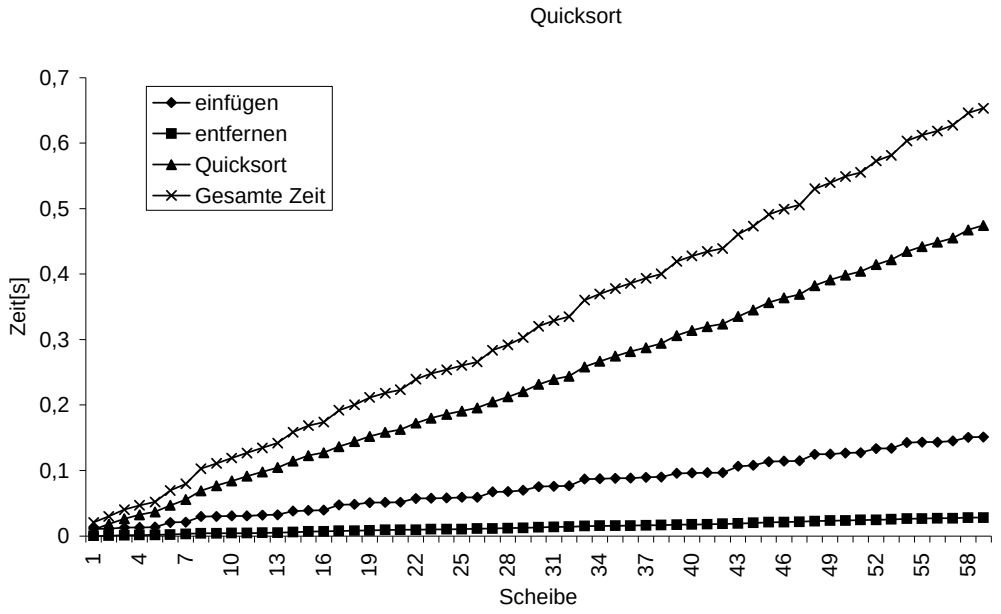


Abbildung 11-16

Das Sortieren erzeugt den größten Teil der Kosten. Das Einfügen in die Liste benötigt mehr CPU-Kosten als das Entfernen, da beim Einfügen noch getestet werden muss, ob das Tupel zum Anfragebereich *Q* gehört.

Die nächste Messung betrachtet die Anzahl der Vergleichsoperationen, die benötigt werden, um die Ergebnismenge zu sortieren. Das Messergebnis wird mit dem Idealfall verglichen. Der Idealfall tritt genau dann ein, wenn der Cache nach der Verarbeitung einer Scheibe leer ist.

	Scheiben	Tupel pro Scheibe	Vergleiche pro Scheibe	Vergleiche insgesamt
Theorie	59	170	1487	87694
Messung	59	690	7681	453125

Tabelle 11-5

Tabelle 11-5 zeigt das Ergebnis der Messung. Im Durchschnitt beinhaltet eine Scheibe 690. Die Anzahl der Tupel in einer Scheibe schwankt zwischen 413 bis 1038. Im Idealfall muss bei 59 Scheiben, um 10000 Tupel zu sortieren, eine Scheibe nur 170 Tupel enthalten. Somit erhöht sich die Menge durch n_{old} Tupel, d.h. Tupel, die durch eine frühere Scheibe bereits in den Cache geladen worden sind, um Faktor 4,06. Somit müssen mehr Vergleichsoperationen durchgeführt werden. Im Idealfall benötigt man 87694 Vergleichsoperationen. Für die Messung benötigt der Quicksort-Algorithmus insgesamt 453125 Vergleichsoperationen. Somit erhält man einen Faktor von 5,17, d.h. man erhält somit eine Erhöhung der CPU-Kosten um einen Faktor von 5,17.

11.2.3 Bewertung

Der Einsatz des Quicksort-Algorithmus, um die Tupel im Cache nach der Sortierdimension zu sortieren, hat im Wesentlichen zwei Nachteile:

Im Cache liegen bereits n_{old} Tupel, die nach der Sortierdimension sortiert sind. Da dieser Teil der Tupel bereits sortiert ist und nun die neuen Tupel n_{new} eingefügt werden, besteht die Gefahr, dass die Laufzeit des Quicksort-Algorithmus ca. $O(n^2)$ ist. Durch den Einsatz der 3-Median-Strategie kann die Wahrscheinlichkeit gemindert werden. Messungen haben gezeigt, dass eine durchschnittliche Laufzeit von $O(n \lg n)$ erreicht wird.

Der zweite Nachteil besteht darin, dass nicht nur die neuen Tupel n_{new}, sondern auch n_{old} im Cache sortiert werden. Die Tupel werden jedes Mal wieder neu sortiert, bis sie Element der aktuellen Sweep-Hyperebene sind. Da durch die Reinsprünge Tupel im Cache eingefügt werden, die bereits die Stufenlinie-1,0 überspringen, können Tupel existieren, die erst aus dem Cache entfernt werden, wenn die letzte Scheibe verarbeitet wird. Da die Anzahl der Sprünge über die Stufenlinie-k,0 gegenüber der Anzahl der Sprünge über Stufenlinie-k,i exponentiell abnimmt, existieren exponentiell mehr Sprünge über die Stufenlinie-i,k als über die Stufenlinie-k,(i-1). Somit existieren die wenigsten Sprünge über die Stufenlinie-k,0. Somit nimmt die Anzahl der Tupel, die lange im Cache verweilen, exponentiell ab. Für die oben vorgestellten Messungen mussten ca. 4 Mal so viele Vergleiche beim Sortieren durchgeführt werden als im Idealfall. Somit erhöhen sich die CPU-Kosten für diese Messung um Faktor 4.

11.2.4 Heapsort

Eine Alternative zum Quick-Sort stellt der Heapsort dar. Da der Heapsort für den Tetris-Algorithmus eingesetzt wird und der Einfügephase die Sortierphase folgt, ohne dass der Cache vollständig leer ist, wird eine „button-up“ Heap-Erzeugungsphase nicht verwendet. Dies könnte bei der ersten Scheibe verwendet werden. Dies wird jedoch für die theoretische Untersuchung nicht berücksichtigt. Es werden wie beim Quick-Sort auch nur die Vergleichsoperationen betrachtet.

Kosten für das Einfügen

Die Kosten, um einen Tupel in einen Heap einzufügen, betragen im schlechtesten Fall:

$$c_{heap-insert} = \lfloor \log_2(n+1) \rfloor \qquad \text{Gl: 11-33}$$

Dieser Fall tritt genau dann ein, wenn das Tupel bis zur Wurzel sickert, d.h. das neue Element hat den kleinsten Wert. n bezeichnet die Anzahl der Tupel, die vor dem Einfügen im Heap enthalten sind. Um nun ein n_{new} Tupel in eine Heap mit n_{old} Tupel einzufügen erhält man folgende obere Grenze:

$$\sum_{i=n_{old}}^{n_{new}+n_{old}-1} c_{heap-insert} = \sum_{i=n_{old}}^{n_{new}+n_{old}-1} \lfloor \log_2(i+1) \rfloor \leq \sum_{i=n_{old}}^{n_{new}+n_{old}-1} \lfloor \log_2(n_{new}+n_{old}-1+1) \rfloor = n_{new} \log_2 \lfloor n_{new}+n_o \qquad \text{Gl: 11-34}$$

Kosten für das Löschen

Um das kleinste Element aus dem Heap zu finden, benötigt man keine Vergleichsoperation, da das kleinste Element an der Wurzel des Heap liegt. Ist das kleinste Element aus dem Heap entfernt worden, muss die Heap-Ordnung wieder hergestellt werden. Hierzu werden

$$2 \cdot \left(\left\lceil \log_2 (n+1) \right\rceil - 1 \right) = 2 \cdot \left\lfloor \log_2 n \right\rfloor \qquad \text{Gl: 11-35}$$

Vergleichs-Operationen im schlechtesten Fall benötigt, falls das Element bis auf ein Blatt nach unten bewegt werden muss.

Um nun n_{out} Tupel aus einem Heap zu entfernen, erhält man folgende obere Grenze:

$$\sum_{i=n_{new}+n_{old}-n_{out}+1}^{n_{new}+n_{old}} 2 \cdot \left\lfloor \log_2 (i) \right\rfloor \leq 2 \sum_{i=n_{new}+n_{old}-n_{out}+1}^{n_{new}+n_{old}} \left\lfloor \log_2 (n_{new}+n_{old}) \right\rfloor = 2 \cdot n_{out} \left\lfloor \log_2 (n_{new}+n_{old}) \right\rfloor \qquad \text{Gl: 11-36}$$

11.2.5 Messungen

Die hier präsentierten Messungen basieren auf der 2-dimensionalen Datenbank mit gleichverteilten Daten. Eine genaue Beschreibung ist in Tabelle 11-4 zu finden. Abbildung 11-17 zeigt die Einfüge-Kosten des Heap-Sort-Algorithmus in Abhängigkeit der eingefügten Tupel. Es wurde eine *lineare* Trendlinie eingetragen. Sie repräsentiert die durchschnittlichen Kosten für das Einfügen in den Heap. Wie man leicht erkennt sind die Einfügekosten relativ nahe bei der Trendlinie, so dass die Einfügekosten in den Heap als linear angenommen werden können.

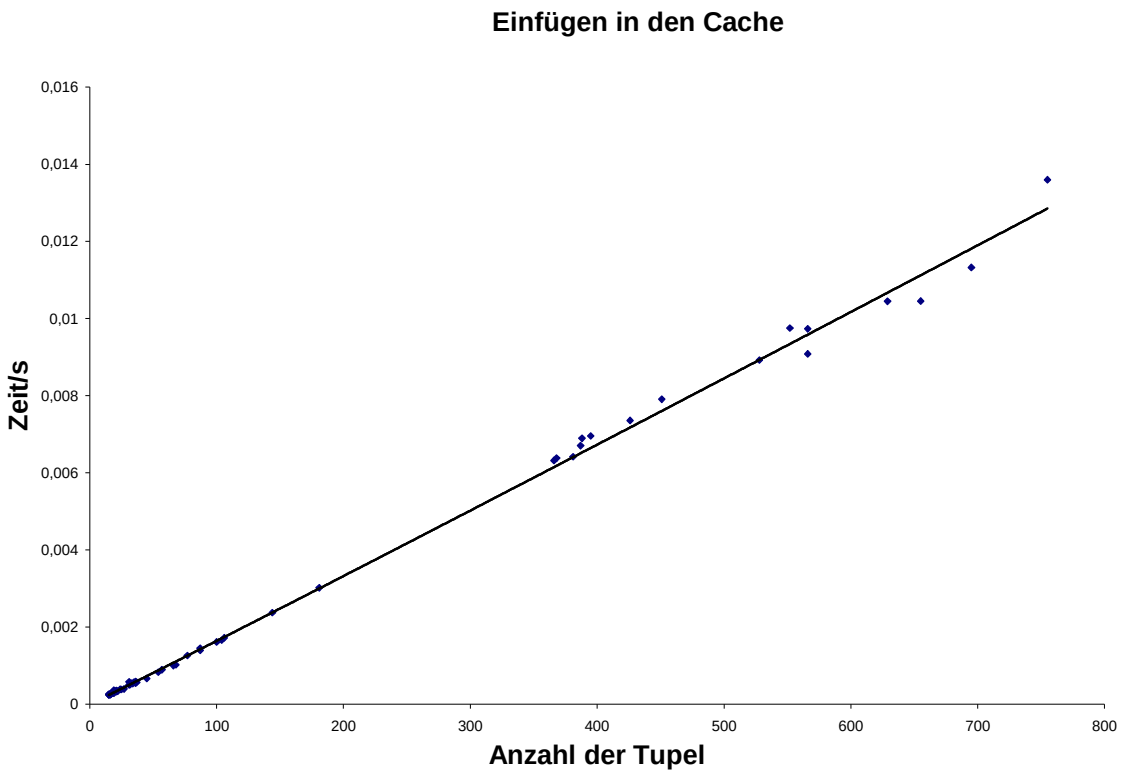


Abbildung 11-17

Der Grund liegt darin, dass der Cache nach der ersten Scheibe nicht leer ist, und somit bereits n_{old} Tupel im Cache enthalten sind. Bei dieser Messung liegen im Durchschnitt 525 Tupel im Cache und es werden im Durchschnitt 165 Tupel pro Scheibe eingefügt. Nachdem alle Tupel einer Scheibe in den Heap eingefügt worden sind, enthält der Heap im Durchschnitt 690 Tupel. Die Höhe des Heaps beträgt in beiden Fallen 10, so dass im schlechtesten Fall noch jeweils 10 Vergleichoperationen benötigt werden, um das Tupel in den Heap einzufügen. Eine Scheibe benötigt somit im Durchschnitt 1650 Vergleichsoperationen.

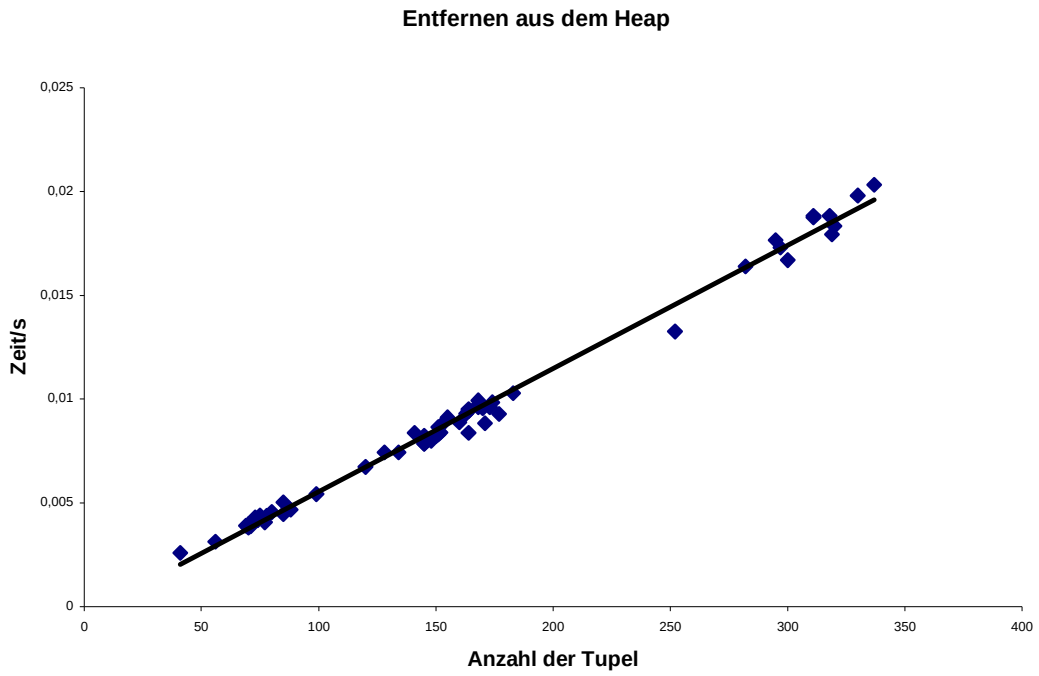


Abbildung 11-18

Abbildung 11-18 zeigt die Löschkosten des Heapsort-Algorithmus in Abhängigkeit der gelöschten Tupel. Es wurde eine *lineare* Trendlinie eingetragen. Sie repräsentiert die durchschnittlichen Kosten für das Löschen im Heap. Wie man leicht erkennt sind die Löschkosten relativ nahe bei der Trendlinie, so dass die Löschkosten aus dem Heap als linear angesehen werden können. Der Grund ist identisch zum Einfügen.

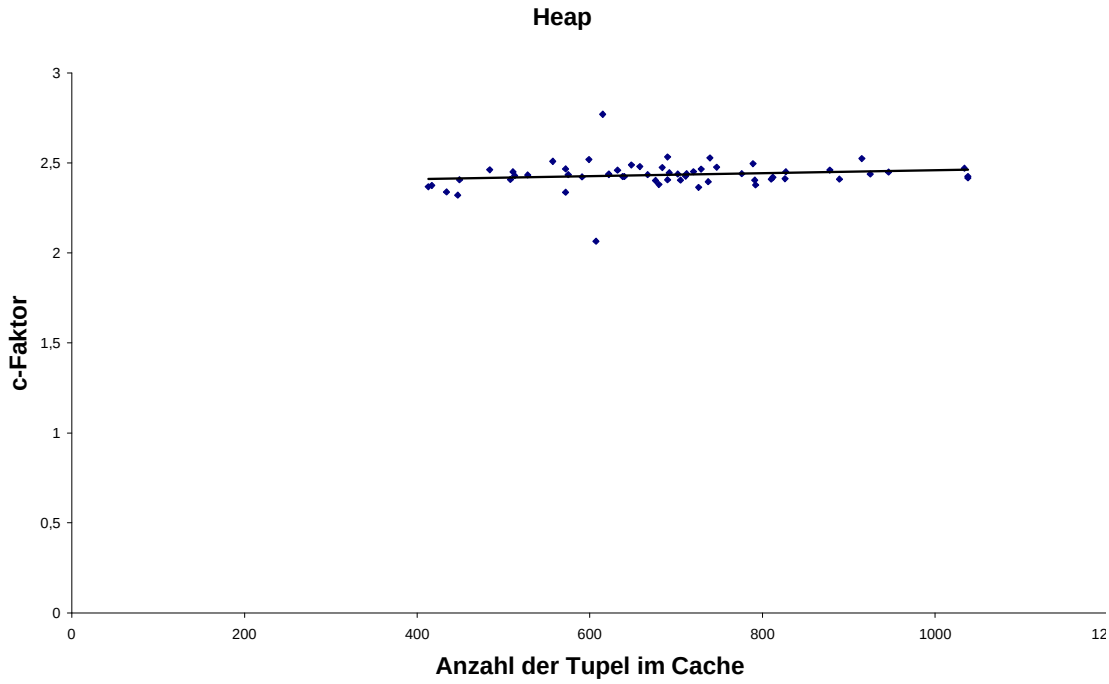


Abbildung 11-19:

Nach Gl: 11-36 ergibt sich für den *c*-Faktor $c_{c\text{-}heap}$ für das Löschen aus dem Heap:

$$t_{heap-out} = c_{c-heap} \cdot 2 \cdot n_{out} \lfloor \log_2 (n_{new} + n_{old}) \rfloor \qquad \text{Gl: 11-37}$$

Abbildung 11-19 zeigt den c-Faktor für verschiedene Cachegrößen, die durch die Verarbeitung der einzelnen Scheiben entstanden sind. Der *c*-Faktor liegt zwischen 2,3 und 2,5 ms. Es existiert ein Messwert, der ein c-Faktor von nur 2,0 ms hat. Dieser c-Faktor entsteht bei der Verarbeitung der letzten Scheibe, da der ganze Heap aufgelöst wird.

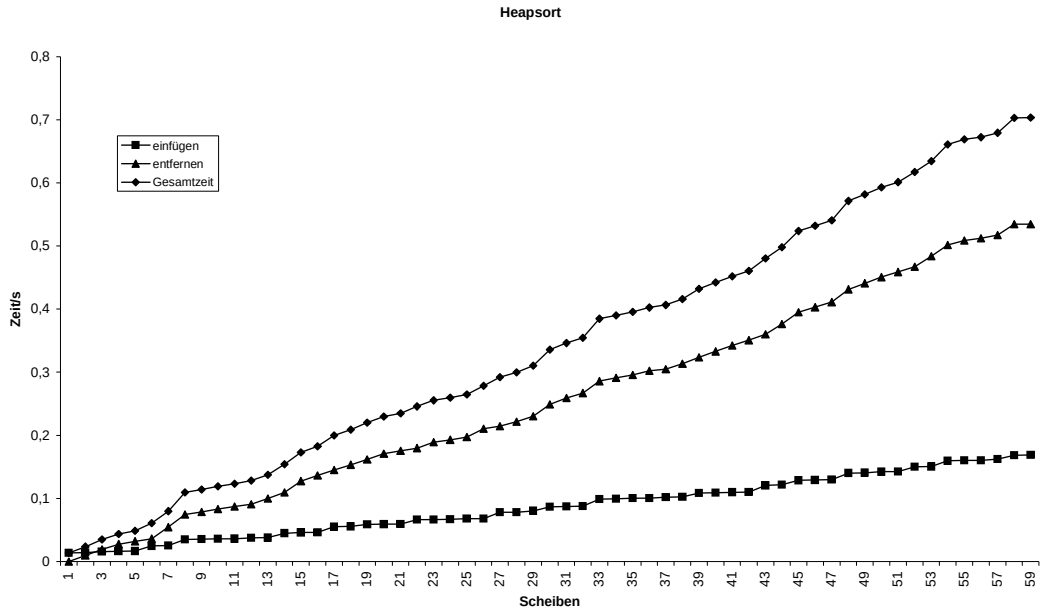


Abbildung 11-20

Gl: 11-20 zeigt die Gesamtkosten kumuliert über die 59 Scheiben. Nach dem Kostenmodell aus Kapitel 11.2.4 werden für das Löschen im Worst-Case ca. zwei Mal soviel CPU-Kosten benötigt als für das Einfügen. Die Messung zeigt jedoch einen Faktor von 3. Das zeigt, dass beim Löschen das Element, das die Wurzel ersetzt, im Durchschnitt fast bis zur Blattebene sickert. Dieser Fall tritt beim Einfügen nicht so häufig auf.

11.2.6 Vergleich Quicksort und Heap

In diesem Abschnitt wird nun ein theoretischer Vergleich zwischen Quicksort und Heapsort für den Tetris-Algorithmus durchgeführt. Es wird zunächst der Heap betrachtet. Um n_{new} Tupel einer neuen Scheibe in den Cache einzufügen, müssen

$$n_{neu} \lfloor \log_2 (n_{new} + n_{old} + 1) \rfloor \qquad \text{Gl: 11-38}$$

Vergleiche im schlechtesten Fall durchgeführt werden. Um n_{out} Tupel aus dem Cache zu entfernen, werden im schlechtesten Fall

$$2 \cdot n_{out} \lfloor \log_2 (n_{new} + n_{old} - 1) \rfloor \quad \text{Gl: 11-39}$$

Vergleiche benötigt. Somit erhält man für die Verarbeitung einer Scheibe folgende Kosten:

$$c_{heap} = n_{neu} \lfloor \log_2 (n_{new} + n_{old} + 1) \rfloor + 2 \cdot n_{out} \lfloor \log_2 (n_{new} + n_{old} - 1) \rfloor \quad \text{Gl: 11-40}$$

Unter der Annahme, dass pro Scheibe genauso viel Tupel eingefügt und entfernt werden, (n_{new}= n_{old}) erhält man

$$c_{heap} \approx 3 \cdot n_{out} \log_2 (n_{new} + n_{old} - 1) \quad \text{Gl: 11-41}$$

Für Quicksort erhält man

$$c_{quicksort} \approx 1.18 \cdot (n_{new} + n_{old}) \log_2 (n_{new} + n_{old}) \quad \text{Gl: 11-42}$$

Der Verhältnis der Kosten für Heapsort und Quicksort für das Sortieren einer Scheibe ist somit:

$$\frac{c_{heapsort}}{c_{quicksort}} \approx \frac{3 \cdot n_{new} \log_2 (n_{new} + n_{old})}{1.18 \cdot (n_{new} + n_{old}) \log_2 (n_{new} + n_{old})} = \frac{3 \cdot n_{new}}{1.18 \cdot (n_{new} + n_{old})} \quad \text{Gl: 11-43}$$

Basierend auf diesen Annahmen benötigt der Heapsort Algorithmus weniger Vergleiche als der Quicksort-Algorithmus, wenn:

$$\frac{n_{new}}{n_{old}} < 0.64 \quad \text{Gl: 11-44}$$

gilt.

Handelt es sich beim UB-Baum um eine perfekte idealisierte gleichverteilte Partitionierung (siehe Definition 4-8) entstehen keine Fransen und Sprünge. Somit gehören alle Daten im Cache zur aktuellen Scheibe. Der Cache wird somit sortiert und dann vollständig gelehrt. Man erhält somit

$$\frac{c_{heapsort}}{c_{quicksort}} \approx \frac{3 \cdot n_{new}}{1.18 \cdot n_{new}} = 2{,}54 \quad \text{Gl: 11-45}$$

Somit benötigt der Heap-Sort-Algorithmus 2,54 Mal mehr Vergleiche als der Quicksort. Da jedoch jeweils nach jeder Scheibe der Cache leer ist, kann der Heap vollständig neu aufgebaut werden. Dadurch wird die Zahl der Vergleiche für den Heapsort weiter reduziert [Knu98v3] und man erhält

$$\frac{c_{heapsort}}{c_{quicksort}} \approx \frac{2 \cdot n_{new} \log_2 (n_{new})}{1.18 \cdot n_{new} \log_2 (n_{new})} = \frac{2 \cdot n_{new}}{1.18 \cdot n_{new}} = 1.69 \quad \text{Gl: 11-46}$$

Messungen zeigen, dass der Heapsort in diesem Fall um Faktor 2 langsamer ist als der Quicksort, da der Quicksort eine sehr effiziente innere Schleife besitzt [Knu98v3].

11.2.7 Messung

In diesem Abschnitt werden einige Messergebnisse vorgestellt, die den Heap-Sort-Algorithmus mit den Quicksort-Algorithmus in der Cacheverarbeitung des Tetris-Algorithmus miteinander vergleichen. Weitere Messergebnisse sind in [KleZ00] zu finden.

Für die Messung wurde die 2 dimensionale Datenbank DB1 verwendet. Eine genaue Beschreibung der Datenbank ist in Tabelle 11-4 zu finden. Abbildung 11-21 zeigt die Sortierkosten kumulativ in Abhängigkeit der verarbeiteten Scheiben. Die Messung liest den Anfragebereich *Q* sortiert nach x_1 .

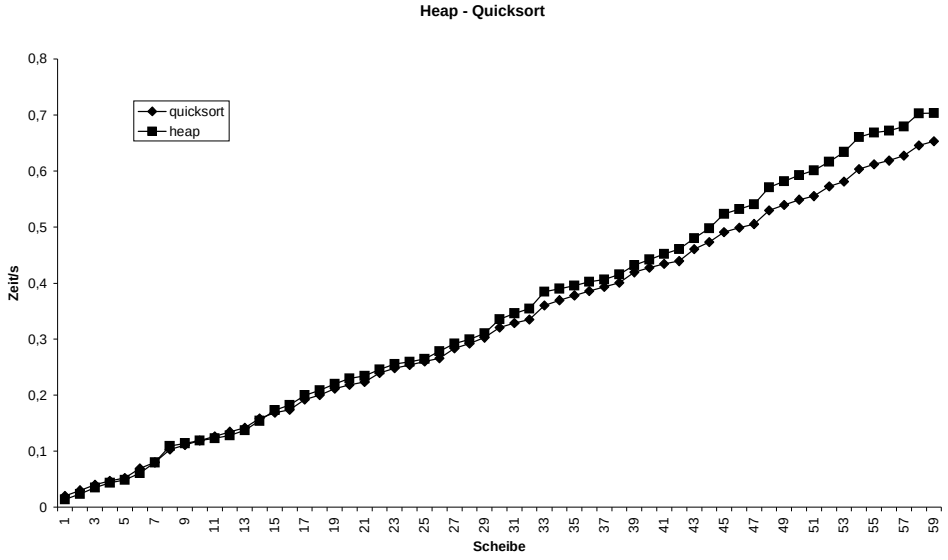


Abbildung 11-21

Beide Algorithmen Quicksort und Heapsort zeigen ein lineares Verhalten. Der Quicksort-Algorithmus ist dem Heapsort-Algorithmus leicht überlegen. Der Heapsort-Algorithmus benötigt 0,70 s um die 10.000 Tupel zu sortierten, der Quicksort-Algorithmus benötigt 0,65 s. Somit benötigt der Quicksort 7% weniger CPU-Zeit. Dies entspricht dem Kostenmodell aus Kapitel 11.2.1 und 11.2.4. Nach Gl: 11-31 benötigt der Quicksort-Algorithmus:

$$t_{quick} = c_{c-quick} \cdot 1.18 \cdot (n_{old} + n_{new}) \cdot \log_2(n_{old} + n_{new}) = \qquad \text{Gl: 11-47}$$
$$= c_{c-quick} \cdot 1.18 \cdot (690) \cdot \log_2(690) = c_{c-quick}\, 7681$$

Der Heapsort-Algorithmus benötigt nach Gl: 7-34 undGl: 11-37 im schlechtesten Fall:

$$t_{heap} \approx c_{c-heap} \cdot 2 \cdot n_{out} \cdot \log_2(n_{old} + n_{new}) + c_{c-heap} \cdot n_{new} \cdot \log_2(n_{old} + n_{new}) = \qquad \text{Gl: 11-48}$$
$$\overset{n_{new}=n_{out}}{=} \; c_{c-heap} \cdot 3 \cdot n_{new} \cdot \log_2(n_{old} + n_{new}) = c_{c-heap} \cdot 3 \cdot 169 \cdot \log_2(690) = c_{c-heap}\, 4781$$

Setzt man die empirischen *c*-Faktoren in die jeweiligen Gleichungen erhält man für t_{heap} = *2,45 µs* * 4781* 59 = 0,69 *s* und für Quicksort erhält man t_{quick} = *1,0 µs* * 7681 * 59 = 0,45 *s*. Zusätzlich muss beim Quicksort-Algorithmus noch die Zeit für die Verwaltung der Liste berücksichtigt werden. Für diese Messung betrug diese 0,18 s. Somit erhält man eine Zeit von 0,63 *s*. Dies entspricht in etwa den gemessenen Werten.

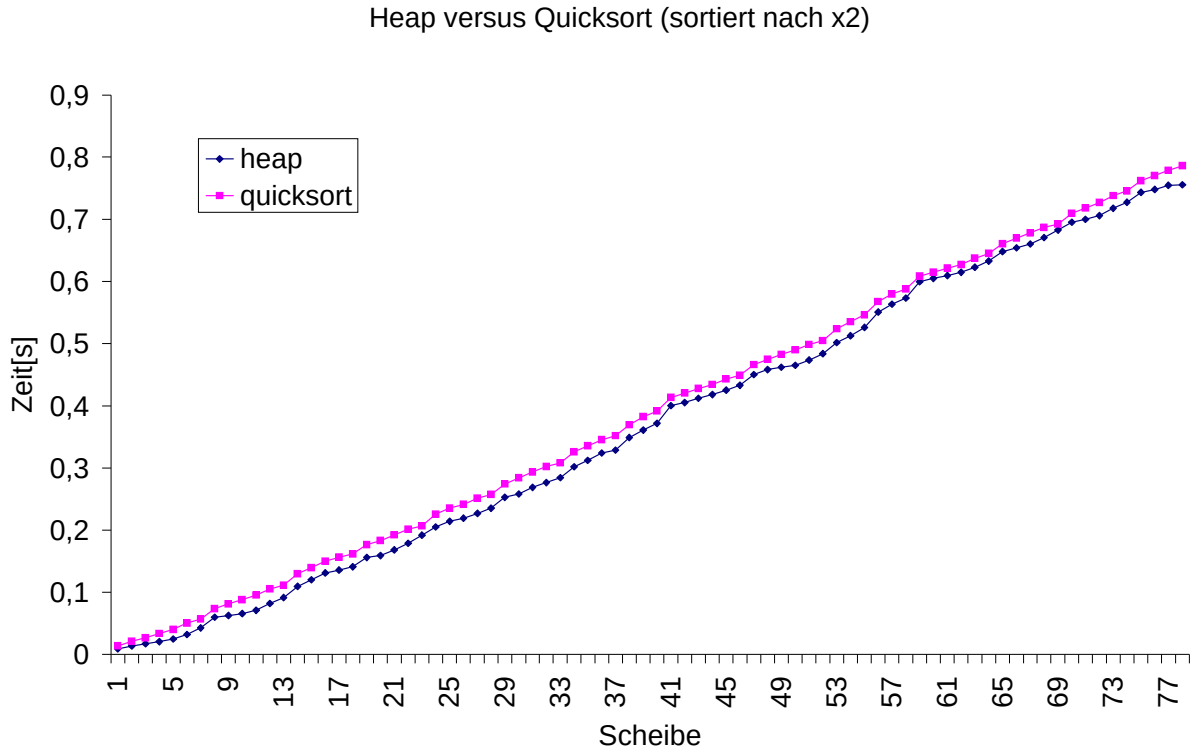


Abbildung 11-22

Abbildung 11-22 zeigt eine Messung auf der Datenbank DB1. Hier wird das Ergebnis sortiert nach x_2 ausgegeben. Die Sortierkosten werden kumulativ in Abhängigkeit der verarbeiteten Scheiben dargestellt. Beide Algorithmen zeigen das erwartete lineare Verhalten. Diesmal ist der Heapsort-Algorithmus dem Quicksort-Algorithmus überlegen. Der Quicksort-Algorithmus benötigt für die Sortierung 0,76 *s*, der Quicksort-Algorithmus 0,78 *s*.

Datenbank	Tupel/ Seite	Anzahl der Scheiben	Anzahl der Tupel im Cache	Anzahl gelöschten Tupel pro Scheibe	Vergleich Heap pro Scheibe	Vergleich Quicksort pro Scheibe	Zeit [s] Heap	Zeit [s] Quicksort
DB1 (sort by x1)	20	59	689	169	4780	7665	0.70	0.65
DB1 (sort by x2)	20	78	513	128	3457	5450	0.76	0.78

Tabelle 11-6

Somit ist der Heap um 2,7 % schneller als der Quicksort-Algorithmus. Nach dem Kostenmodell benötigt der Heapsort-Algorithmus 10% mehr Zeit als der Quicksort-Algorithmus. Das Kostenmodell geht jedoch beim Quicksort-Algorithmus vom Durchschnitt aus, der bei dieser Datenverteilung nicht eintritt. Eine Zusammenfassung der Messergebnisse ist in Tabelle 11-6 zu finden.

11.2.8 Bewertung

Es wurden zwei Sortierverfahren für den Einsatz im Cache für den Tetris-Algorithmus untersucht, Heapsort und Quicksort. Zunächst wurde auf Basis der Vergleichsoperation jeweils ein Kostenmodell aufgestellt, das durch Messungen belegt worden ist. Im Allgemeinen benötigt der Quicksort-Algorithmus nur 56% der Vergleichoperationen des Heap-Sort-Algorithmus. Bei der Cacheverwaltung des Tetris-Algorithmus ergibt sich jedoch ein völlig anderes Bild, das durch die Struktur der Z-Kurve, nach der der UB-Baum die Daten auf den Sekundärspeicher clustert, verursacht wird. Im Cache sind bereits n_{old} Tupel, die bereits durch die Verarbeitung vorangegangener Scheiben in den Cache geladen worden sind. Es werden pro Scheibe n_{new} hinzugefügt. Somit sortiert der Quicksort-Algorithmus n_{old} Tupel erneut. Ist das Verhältnis $n_{new}/n_{old} < 0{,}64$ benötigt der Heapsort-Algorithmus weniger Vergleiche als der Quicksort-Algorithmus und stellt somit die bessere Alternaive für die Sortierung des Caches dar. Da der Heap-Sort-Algorithmus nicht ganz so effizient wie der Quicksort-Algorithmus ist, kostet eine Vergleich-Operation in beiden Algorithmen unterschiedlich viel Zeit. Bei der hier verwendeten Protoimplementierung benötigt der Heap-Sort-Algorithmus 2,4 Mal mehr Zeit als der Quicksort-Algorithmus um eine Vergleichoperation auszuführen. Des Weiteren wurde festgestellt, dass die Löschoperation des Heaps der Kostentreiber ist. Da der Heap-Sort-Algorithmus $O(n \log n)$ garantiert, stellt er die bevorzugte Methode für den Tetris-Algorithmus dar.

Literaturverzeichnis

[AdeL62] G. M. Adelson-Velskii and Y. M. Landis, "An Algorithm for the organisation of information," *Doklady Akademia Nauk SSSR, 146:263-266 ;English Translation: Soviet Math. 3, 1259-1263*, 1962.

[AgaAD+96] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi, "On the Computation of Multidimensional Aggregates," presented at VLDB, 1996.

[AhoHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithm*. Reading Massachusetts: Addison-Wesley, 1974.

[AraS89] C. R. Aragon and R. G. Seidel, "Randomized search trees," presented at 30th IEEE Symposium on Foundations of Computer Science, 1989.

[Bae89] R. A. Baeza-Yates., "Expected behaviour of B + -trees under random insertions.," *Acta Informatica,*, vol. 26, pp. 439-472, 1989.

[BarGN*98] T. Barclay, R. Eberl, J. Gray, J. Nordlinger, G. Raghavendran, D. Slutz, G. Smith, and P. Smoot, "The Microsoft TerraServer," Microsoft Research, Redmond, USA MSR-TR-98-17, 1998.

[Bay71] R. Bayer, presented at ACM-SIGFIDET Workshop, 1971.

[Bay96] R. Bayer, "*The universal B-Tree for multidimensional Indexing*," Institut für Informatik, TU München, München TUM-I9637, 1996.

[Bay97b] R. Bayer, "*UB-Trees and UB-Cache - A new Processing Paradigm for Database Systems*," Institut für Informatik, TU München, München TUM-I9722, 1997.

[Bay97a] R. Bayer, "The universal B-Tree for multidimensional Indexing: General Concepts," presented at World-Wide Computing and Its Applications WWCA, 1997.

[Bay02] R. Bayer, "Datawarehousing," Forschungs- und Lehreinheit Informatik III, Fakultät für Informatik , Technische Universität München, München, Vorlesung SS 2002.

[BayM98] R. Bayer and V. Markl, "The UB-Tree: Performance of Multidimensional Range Queries," Technische Universität München, Informatik, München Technical Report TUM-I9814, 1998.

[BayM72] R. Bayer and E. M. McCreight, "Organization and Maintenance of Large Ordered Indices," *Acta Informatica*, vol. 1, pp. 173-189, 1972.

[BayU77] R. Bayer and K. Unterauer, "Prefix B-Tree," *ACM Transaction on Database Systems*, vol. 2, pp. 11-26, 1977.

[BecKS+90] N. Beckmann, H. P. Kriegel, R. Schneider, and S. B., "*The R*-Tree. An efficient and robust Access Method for Points and Rectangles*," presented at Proc. of ACM SIGMOD Conf, 1990.

[BitW83] D. Bitton and D. J. DeWitt, "Duplicate record elimination in large data files," *ACM Transactions on Database Systems*, vol. 8, 1983.

[BlaCE77] M. W. Blasgen, R. G. Casey, and K. P. Eswaran, "An Encoding Method for Multifield Sorting and Indexing," *Comm. ACM*, vol. 20, pp. 874-878, 1977.

[ChaF81] J.-M. Chang and K.-s. Fu, "Extended K-d Tree Database Organization: A Dynamic Multiattribute Clustering Method," *IEEE Transactions on Software Engineering (TSE),*, vol. 7, 1981.

[ChaD97] S. Chaudhuri and U. Dayal, "*An Overview of Data Warehousing and OLAP Technologies*," *ACM SIGMOD Record*, vol. 26, 1997.

[ChaS94] S. Chaudhuri and K. Shim, "Including Group-By in Query Optimization of Queries with Aggregates," presented at VLDB, Santiago de Chile, Chile, 1994.

[CheHH+91] J. Cheng, D. Haderle, R. Hedges, B. R. Iyer, T. Mess-inger, C. Mohan, and Y. Wang, " *An effi-cient hybrid hash join algorithm: a DB2 proto-type*," presented at Proc. of the 7th ICDE, Kobe, 1991.

[BöhK99] H.-P. K. Christian Böhm, "Efficient Bulk Loading of Large High-Dimensional Indexes," , DaWaK, 1999.

[Cod70] E. F. Codd, "*A Relational Model of Data for Large Shared Databanks*," *Comm. of ACM*, vol. 13, pp. 377-387, 1970.

[CodCS93] E. F. Codd, S. B. Codd, and C.T.Salley, "Providing OLAP (On-line Anaytical Processing) to User-Analysts: An IT Mandate," Arbor Essbase 1993.

[Com79] D. Comer, "*The ubiquitous B-Tree*," *ACM Computing Surveys*, vol. 11, pp. 121-138, 1979.

[CorLR90] T. H. Cormen, C. E. Leiserson, and R. E. Riverst, *Introduction to Algorithms*. New York: McGraw-Hill Book Company, 1990.

[Date95] C. J. Date, *An Introduction to Database Systems*, sixth ed. New York: Addison-Wesley, 1995.

[DewKO+84] D. J. DeWitt, R. H. Katz, F. Olken, S. L. D., S. M.R., and W. D., "*Implementa-tion Techniques for Main Memory Database Systems*," presented at Proc. of ACM SIGMOD Conf, 1984.

[ElmR94] R. Elmasri and S. B. Navathe, *Fundamentals of database systems*. Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1994.

[EstW92] V. Estivill-Castro and D. Wood, "A Survey of Adaptive Sorting Algorithms," *ACM Computing Surveys*, vol. 24, pp. 441-476, 1992.

[FagN+79] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible Hashing - A Fast Access Method for Dynamic Files," *TODS*, vol. 4, pp. 315-344, 1979.

[Fal88] C. Faloutsos, "Gray-Codes for Partial Match and Range Queries," *IEEE Trans. Software Eng. 14*, 1988.

[FenKM00] R. Fenk, A. Kawakami, V. Markl, R. Bayer, and S. Osaki, "Bulk loading a Data Warehouse built upon a UB-Tree," presented at IDEAS, Yokohama, Japan, 2000.

[FreMV94] J. C. Freytag, D. Maier, and G. Vossen, *Query Processing for Advanced Database Systems*. San Matea (CA): Morgan Kaufmann Publishers, Inc., 1994.

[FriM97] N. Frielinghaus, "Evaluierung der Einsatzfähigkeit des UB-Baums für das SAP-System R/3," , V. Markl, Ed. München: Technische Universität München, 1997.

[GaeG98] V. Gaede and O. Günther, "*Multidimensional Access Methods*," *ACM Computing Surveys*, vol. 30, pp. 170 - 231, 1998.

[GräKK+01] A. Gärtner, A. Kemper, D. Kossmann, and B. Z. 1, "Efficient Bulk Deletes in Relational Databases," presented at ICDE, Heidelberg, Germany, 2001.

[Gel93] A. V. Gelder, "Multiple Join Size Estimation by Virtual Domains," presented at POS, 1993.

[Gra93] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, vol. 25, pp. 73-170, 1993.

[Gra78] J. Gray, "Notes on Data Base Operating Systems," presented at Advanced Course: Operating Systems, 1978.

[GraCB+97] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total," *Data Mining and Knowledge Discovery*, vol. 1, pp. 29-53, 1997.

[GraG97] J. Gray and G. Graefe, "The Five-Minute Rule Ten Years Later,
and Other Computer Storage Rules of Thumb," *SIGMOD Record*, vol. 26, pp. 63-68, 1997.

[GraR97] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, California USA: Morgan Kaufmann Publishers, 1997.

[Gre89] D. Greene, "*An Implementation and Performance Analysis of Spatial Data Access Methods*," presented at Proc. of 5th ICDE, 1989.

[Gün93] O. Günther, "Efficient Computations of Spatial Joins," presented at Proc. of 9th ICDE, Vienna, 1993.

[GünB00] H. Günzel and A. Bauer, *Data-Warehouse-Systeme; Architektur, Entwicklung, Anwendung*. Heidelberg: dpunkt Verlag, 2000.

[GupHR+97] H. Gupta, V. Harinarayan, A. Rajaraman, and D. Ullman, "*Index Selection for OLAP*," presented at Proc. of ICDE, 1997.

[Gut84] A. Guttman, "*R-Trees: A dynamic Index Structure for spatial Searching*," presented at Proc. of ACM SIGMOD Conf, 1984.

[HarNK+90] L. Harada, M. Nakano, M. Kitsuregawa, and M. Takagi, " *Query Processing Methods for Multi-At-tribute Clustered Relations*," presented at Proc. of 16th VLDB Conf, 1990.

[Här77] T. Härder, "A Scan-Driven Sort Facility for a Relational Database System," presented at VLDB, Tokyo, 1977.

[HärR99] T. Härder and E. Rahm, *Datenbanksysteme, Konzepte und Techniken der Implementierung*. Berlin, Heidelberg, New York: Springer, 1999.

[HarRU96] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing Data Cubes Efficiently," presented at SIGMOD, Montreal, Canada, 1996.

[HarR96] E. P. Harris and K. Ramamohanarao, "Join algorithm costs revisited," *VLDB Journal*, vol. 5, 1996.

[HickZ00] S. Hick, "Optimierung und Generalisierung des Tetris- und TempTris-Algorithmus," in *Institut für Informatik*. München: Technische Universität München, 2000.

[HicZ00] S. Hick and B. M. Zirkel, "Analysis and Implementation of the Sorted-Writing Operation on UB-Trees," in *Fakultät für Informatik III*. München: Technische Universität München, 2000, pp. 20.

[HjaSS97] G. R. Hjaltason, H. Samet, and Y. Sussmann., "Speeding up Bulk-Loading of Quadtrees," presented at ACM International Workshop on Advances in Geographic Information Systems, 1997.

[Inf97] I. S. Inc, "A New Generation of Decision Support Indexing for Enterprisewide Data Warehouses," , 1997.

[Int01] Intel, “Intel Pentium 4 Processor in the 478-pin Package at 1.50 GHz, 1.60 GHz, 1.70 GHz, 1.80 GHz, 1.90 GHz and 2 GHz, Datasheet,” , vol. 2001, 2001.

[Jag90] H. V. Jagadish, “*Linear Clustering of Objects with multiple Attributes*,” presented at Proc. of ACM SIGMOD Conf, 1990.

[Jar84] M. Jarke and J. Koch, “*Query Optimization in Database Systems*,” *ACM Computing Surveys*, vol. 16, pp. 111-152, 1984.

[KamF93] I. Kamel and C. Faloutsos, “On Packing R-trees,” presented at CIKM, 1993.

[EmE96] A. Kemper and E. Eickler, *Datenbanksysteme: eine Einführung*. München: R. Oldenbourg Verlag GmbH, 1996.

[Kim96] R. Kimball, *The Data Warehouse Toolkit*: John Wiley & Sonc, Inc., 1996.

[KleZ00] W. Klein, “Performance of Sort Operations in a Relational Database System,” in *Fakultät für Informatik III*. München: Technische Universität München, 2000, pp. 51.

[Klu82] A. Klug, “Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Funktions.,” *Journal of the ACM*, vol. 29, pp. 699-717, 1982.

[Knu98v1] D. E. Knuth, *Fundamental Algorithms*, vol. 1. Reading Massachusetts: Addison-Wesley, 1998.

[Knu98v3] D. E. Knuth, *Sorting and Searching*, vol. 3. Reading Massachusetts: Addison-Wesley, 1998.

[Kur99] A. Kurz, *Data Warehousing, Enabling Technology*. Bonn: MITP-Verlag, 1999.

[Küs83] K. Küspert., “Storage utilization in B*-trees with a generalized over ow technique.,” *Acta Informatica,*, vol. 19, pp. 35-55, 1983.

[LanZ99] J. Lanzinger, “Desing and Implementation of a Middleware for Maintaining Multidimensional Indexes in the RDBMS Oracle,” in *Fakultät für Informatik III*. München: Technische Universität München, 1999.

[Lar83] Larson, “Dynamische Hashverfahren,” *Informatik-Spektrum*, vol. 6, pp. 7-19, 1983.

[LarG98] P. Larson and G. Graefe, “Memory Management During Run Generation in External Sorting,” presented at SIGMOD,International Conference on Management of Data, Seattle, Washington, USA, 1998.

[Lar88] P.-Å. Larson, “Linear Hashing with Separators - A Dynamic Hashing Scheme Achieving One-Access Retrieval,” *TODS*, vol. 13, pp. 366-288, 1988.

[LarK84] P.-Å. Larson and A. Kajla, “File Organization: Implementation of a Method Guaranteeing Retrieval in One Access,” *Comm. ACM*, vol. 27, 1984.

[Leh98] W. Lehner, “Aggregatverarbeitung in Multidimensionalen Datenbanksystemen.,” . Erlangen: Friedrich-Alexander Universität, 1998.

[LeuEL97] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez, “Proceedings of the Thirteenth International Conference on Data Engineering,” presented at ICDE: Proceedings of the Thirteenth International Conference on Data Engineering, Birmingham U.K, 1997.

[LeuN97] S. T. Leutenegger and D. M. Nicol, “Efficient Bulk-Loading of Gridfiles.,” *IEEE Transactions on Knowledge and Data Engineering,*, vol. 9, pp. 410-420, 1997.

[Let80] W. Litwin, “Linear Hashing: A New Tool for File and Table Addressing,” presented at VLDB, 1980.

[LoR95] M.-L. Lo and C. V. Ravishankar, “Generating Seeded Trees from Data Sets,” presented at Symposium on Large Spatial Databases, 1995.

[LomS90] D. Lomet and B. Salzberg, “The hB-Tree: A Multiattribute Indexing Method with good guaranteed Performance,” *ACM Transactions on Database Systems*, vol. 15, pp. 625 - 658, 1990.

[LueZ99] Y. Lue, “Investigation of the Applicability of the UB-Tree and the Tetris Algorithm for Relational Query Processing,” in *Fakultät für Informatik III*. Müchen: Technische Universität München, 1999, pp. 84.

[Mar99] V. Markl, *MISTRAL: Processing Relational Queries using a Multidimensional Access Technique*, vol. 59. Sankt Augustin: infix, 1999.

[MarB98] V. Markl and R. Bayer, “*The Tetris-Algorithm for Sorted Reading from UB-Trees*,” presented at "Grundlagen von Datenbanken", 10th GI Workshop, Konstanz, Germany, 1998.

[MarB00b] V. Markl and R. Bayer, “On Analyzing the Cost of Queries with Multi-Attribute Restrictions and Sort Operations,” presented at IDEAS, Yokohama, Japan, 2000.

[MarRB99] V. Markl, F. Ramsak, and R. Bayer, “Improving OLAP Performance by Multidimensional Hierarchical Clustering,” presented at IDEAS, Montreal, Canada, 1999.

[MarZB99] V. Markl, M. Zirkel, and R. Bayer, “Processing Operations with Restrictions in Relational Database Management Systems without external Sorting,” presented at ICDE, Sydney, Australia, 1999.

[Meh84] K. Mehlhorn, *Data Structures and Algorithms Vol1*. *Sorting and Searching*. Berlin/Heidelberg: Springer-Verlag, 1984.

[Mer99] S. Merkel, “Evaluation of the UB-Tree for a Market Research Data Warehouse,” in *Forschungs- und Lehreinheit Informatik III Datenbanksysteme und Wissenbasen*. Tutor: F. Ramsak, München, Germany: Technische Universität München, 1999.

[Mer83] T. H. Merret, “Why sort-merge gives the best implementation of then natural join,” *SIGMOD Record*, vol. 13, pp. 38-51, 1983.

[MisE92] P. Mishra and M. H. Eich, “Join Processing in Relational Databases,” *Computing Surveys*, vol. 24, 1992.

[Mit95] B. Mitschang, *Anfrageverarbeitung in Datenbanksystemen: Entwurfs- und Implementierungskonzepte*: Vieweg, 1995.

[NieHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, “The Grid-File,An Adaptable, Symmetric Multikey File Structure,” *ACM Transactions on Database Systems*, vol. 9, pp. 38-71, 1984.

[NeiQ97] P. O´Neill and D. Quass, “*Improved Query Performance with Variant Indexes*,” presented at Proc. of ACM SIGMOD Conf, Tucson, Arizona, 1997.

[NeiG95] P. O'Neil and G. Graefe, “Multi-Table Join Through Bitmapped Join Indices,” *SIGMOD Record*, vol. 24, 1995.

[OreM84] J. A. Orenstein and T. H. Merret, “A Class of Data Structures for Associate Searching,” presented at Proc. of ACM SIGMOD-PODS Conf, Portland, Oregon, 1984.

[OttW96] T. Ottmann and P. Widmayer, *Algorithmen und Datenstrukturen*. Heidelberg, Berlin, Oxford: Spektrum Akadimischer Verlag, 1996.

[PelZ00] S. A. Pelizzari, “Implementation and Analysis of Randomized Search Trees for the Tetris-Algorithm,” in *Fakultät für Informatik III*. München: Technische Universität München, 2000, pp. 18.

[PreS85] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. New York: Springer-Verlag, 1985.

[Ram02] F. Ramsak, "Towards a general-purpose, multidimensional index: Integration, Optimization and Enhancement of UB-Trees," in *Institut für Informatik*. München: Technische Universität München, 2002, pp. 198.

[RamMF*00] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer, "Integration the UB-Tree into a Database System Kernel," presented at the 26 International Conference on Very Large Databases, Cairo, Egypt, 2000.

[Rob81] J. T. Robinson, "*The K-D-B-Tree: A Search Structure for large multidimensional dynamic indexes*," presented at Proc. of ACM SIGMOD Conf, 1981.

[Rot91] D. Rotem, "*Spatial Join Indices*," presented at Proc. of ICDE, 1991.

[RouKR97] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos, "Cubetree: Organization of and Bulk Incremental Updates on the Data Cube," presented at SIGMOD, 1997.

[RouL85] N. Roussopoulos and D. Leifker., "Direct Spatial Search on Pictorial Databases Using Packed R-trees," presented at ACM SIGMOD International Conference on Management of Data,, Austin, Texas, 1985.

[Sag94] H. Sagan, *Space-Filing Curves*. New York, Inc.: Springer Verlag, 1994.

[Sal89] B. Salzberg, "Merging Sorted Runs Using Large Main Memory," *Acta Informatica*, vol. 27, pp. 195-215, 1989.

[Sam90] H. Samet, *The Design and Analysis of Spatial Data Structures*: Addison Wesley, 1990.

[SAP01a] SAP, "SAP Bibliothek Business Information Warehouse," , 2001.

[Sar97] S. Sarawagi, "Indexing OLAP Data," *IEEE Data Engineering Bulletin*, vol. 20, 1997.

[Sato81] Sato, , 1981.

[SelAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," presented at ACM SIGMOD Conf. on Management of Data, Boston, USA, 1979.

[SelRF87] T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects," presented at VLDB, 1987.

[Stu95] W. Stucky, *Automaten, Sprachen, Berechanbarkeit*. Stuttgart, Germany: Teubner, 1995.

[SwaS94] A. N. Swami and K. B. Schiefer, "On the Estimation of Join Result Sizes," presented at EDBT, 1994.

[TCP01] TCP-H, "TCP-H," , 2001.

[TroH81] H. Tropf and H.-L. Herzog, "Multidimensional range search in dynamically balancde trees," *Angewandte Informatik 2*, 1981.

[Ull88] J. D. Ullman, *Database and Knowledge Based Systems Volume I*. Rockville, MD: Computer Science Press, 1988.

[Wed74] H. Wedekind, "On the Selection of Access Paths in a Data Base System," presented at IFIP Working Conference Data Base Management, Cargèse, Corsica, France, 1974.

[WedZ86] H. Wedekind and G. Zoerntlein, "Prefetching in Realtime Database Applications," presented at SIGMOD, Washington, D.C., USA, 1986.

[Wil64] J. W. J. Williams, "Algorithm 232 (heapsort)," *Communicatioins of the ACM*, vol. 7, pp. 347-348, 1964.

[WonY76] E. Wong and K. Youssefi, “Decomposition - DA Strategy for Query Processing,” *ACM Trans. on Database Systems*, vol. 1, pp. 223-241, 1976.

[YanL94] W. P. Yan and P. A. Larson, “Performing Group-By before Join,” presented at ICDE, Housten, USA, 1994.

[YanL95] W. P. Yan and P. A. Larson, “Eager Aggreation and Lazy Aggreagtion,” presented at VLDB, Zürich, Schweiz, 1995.

[Yao78] A. C. Yao, “On random 2-3 trees,” *Acta Informatica,*, vol. 9, pp. 159-170, 1978.

[YuM98] C. T. Yu and W. Meng, *Principles of database query processing for advanced applications*. San Fancisco, USA: Morgan Kaufmann Publischers, Inc., 1998.

[Zir98] C. Zirkel, “Hinweise für eine eigenständige Existenz der Subgruppe IV humaner Immunglobulin Leichtketten vom Lambda-Typ Aminosäuresequenzanalyse,” in *Institut für klinische Molekularbiologie und Tumorgenetik, GSF München*. München: Universität München, 1998, pp. 119.

[Zir00b] M. Zirkel, “Efficient Calculation of Tetris Adress,” Technische Universität München, München 2000.

[Zir02] M. Zirkel, “Prinzip Arbeitnehmer Urheberrecht und Europarecht. - Dargestellt am Beispiel des Softwareentwicklers,” . München: Bundeswehr Universität München, 2002.

[ZirMB00a] M. Zirkel, V. Markl, and R. Bayer, “Efficient Processing of the Cube Operator,” presented at EDBT, Konstanz, Germany, 2000.

[ZirMB01] M. Zirkel, V. Markl, and R. Bayer, “Expoitation of Pre-sortedness for Sorting in Query Processing , The TempTris-Algorithm for UB-Trees,” presented at IDEAS, Grenoble, 2001.

Definitionsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Index

B

C

D

E

F

G

H

I

K

L

S

T

U

V

W

X

Y

Z

□

□

□

□

□

Gleichungsverzeichnis