TECHNISCHE UNIVERSITÄT MÜNCHEN
Fakultät für Informatik
Lehrstuhl III - Datenbanksysteme

Operational Business Intelligence as a Service in the Cloud

Diplom-Informatiker Univ.
Michael Seibold

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Martin Bichler

Prüfer der Dissertation:

1. Univ.-Prof. Alfons Kemper, Ph.D.
2. Univ.-Prof. Dr. Thomas Setzer,
   Karlsruher Institut für Technologie

Die Dissertation wurde am 06.09.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 21.01.2013 angenommen.

Abstract

Businesses typically employ operational database systems to service transaction-oriented applications that are vital to their day-to-day operations, and use data warehouse systems for analyzing large amounts of business data to support strategic decision making. The data warehouse is periodically updated with data that is extracted from the operational databases and transformed into a schema optimized for analytical processing. This data staging approach suffers from inherent drawbacks. On the one hand, two or more software and hardware systems must be purchased, administrated and maintained. On the other hand, analyses do not incorporate the latest data, but are processed on a stale snapshot in the data warehouse. Lately, the case has been made for so called Operational Business Intelligence, to overcome the disadvantages of this data staging approach. Advances in hardware architecture allow keeping large amounts of data in main-memory and make it possible to process some analytical queries directly on operational database systems without impeding the performance of mission-critical transaction processing too much. In this thesis, we analyze how business applications, like CRM, with Operational Business Intelligence features, like analytic dashboards, can be provided efficiently as a service in the cloud. Cloud providers typically employ a multi-tenant architecture in order to reduce costs by consolidating several customers onto the same hardware and software infrastructure. First, we discuss the challenges for multi-tenancy in the Cloud Computing context and propose to integrate multi-tenancy support into the database management systems in order to reduce administration and maintenance costs. We discuss what multi-tenancy features are required and propose a data model that supports extensibility, data sharing and evolution with branching. Second, we analyze the suitability of emerging cloud data management solutions for providing business applications as cloud services. Third, we focus on mixed workloads that result from Operational Business Intelligence. We propose a special-purpose approach that allows processing the mixed workload of our application scenario with low response times at high throughput rates while minimizing space overhead and adhering to maximal response times and minimal throughput guarantees. Fourth, we present a benchmark to analyze the suitability of database systems for mixed workloads and Operational Business Intelligence. Finally, we propose elastic workload management to improve the resource utilization of database servers running main-memory database systems with mixed workloads.

Contents

Chapter 1

Introduction

Lately, the case has been made for so-called Operational Business Intelligence or Real-time Business Intelligence. SAP's co-founder Hasso Plattner emphasizes the need to perform analytical queries on current data and compares the expected impact of real-time analysis on management of companies with the impact of Internet search engines on all of us [88]. In an Operational Business Intelligence system, analytical queries and business transactions have to be processed at the same time on the same data. This results in mixed workloads which are a big challenge for current database management systems.

Today, cloud computing has become a major industry trend. According to Gartner, "Cloud computing heralds an evolution of business - no less influential than the era of e-business"[1]. We focus on Software-as-a-Service (SaaS), where a service provider owns and operates a standardized application that is accessed over the Internet by many users who correspond to different customers. According to Gartner, "Users will be driven into cloud computing as business application services (e.g., SaaS) ... reach acceptable levels of maturity and offer new innovative technological and business model features that will become increasingly hard to resist" [2]. Today, a wide variety of business applications are provided according to the SaaS model. The best known examples are the Customer Relationship Management (CRM) system salesforce.com and Business ByDesign, the comprehensive business application system of SAP. With SaaS, the service provider and not the customer owns and operates the entire application software and hardware stack. By careful engineering,

[1] `http://www.gartner.com/it/page.jsp?id=1476715` (retrieved 08/28/2012)

[2] `http://www.gartner.com/it/page.jsp?id=1586114` (retrieved 08/28/2012)

it is possible to leverage economy of scale to reduce total cost of ownership relative to on-premise solutions. With regard to database systems, cloud computing may foster the development of data management systems that are optimized for specific application scenarios. Once a SaaS provider has identified an application scenario whose potential customer base is large enough, the entire software and hardware stack should be optimized based on the characteristics of the specific application scenario in order to gain competitive advantage relative to on-premise solutions and more general-purpose cloud offerings. Business applications, like CRM, with Operational Business Intelligence features may be such an application scenario, as CRM has the biggest share of the SaaS market today. Moreover, advances in hardware architecture allow keeping large amounts of data in main-memory which may be the enabling technology to process mixed workloads efficiently, that result from Operational Business Intelligence.

In this thesis, we analyze how business applications, like CRM, with Operational Business Intelligence features, like analytic dashboards, can be provided efficiently as a service in the cloud. We focus on database management systems which are a crucial component of most business applications. Providing an application as a service in the cloud corresponds to the Software-as-a-Service model, which represents a certain form of Cloud Computing. For providing a service efficiently, resource utilization has to be maximized while administration and maintenance costs have to be minimized. In order to maximize resource utilization on modern hardware architectures, an application has to be optimized for multi-core CPU architectures and large main-memory. In the following, we introduce Operational Business Intelligence, Cloud Computing and modern hardware architecture. Furthermore, we highlight our contributions and present the outline of this thesis. Moreover, we disclose previous publications that are relevant for this thesis.

1.1 Operational Business Intelligence

Today, businesses typically rely on transaction-oriented applications to manage business processes. Furthermore, business intelligence applications are used to support strategic decision making, e. g. computing sales revenue of a company by products across regions and time. These two kinds of business applications are vital to business operations and represent different workloads: Online transaction processing (OLTP), like order processing, and online analytical processing (OLAP), like sales analytics. These workloads are typically not executed on the same database system, because they have very different characteristics. On the one hand, OLTP workloads are typically processed by operational database systems and consist of many short business transactions, including update operations. On the other hand, OLAP workloads are typically processed by data warehouse systems and perform mostly read operations, as large amounts of business data are analyzed. If both workloads were performed on the same data in a single database, the resulting mixed workload may lead to resource contention and result in unacceptable transaction processing performance. Therefore, OLTP and OLAP workloads are typically processed by separate specialized database management systems (DBMSs) (see Figure 1.1 and chapter 17 in [62]).

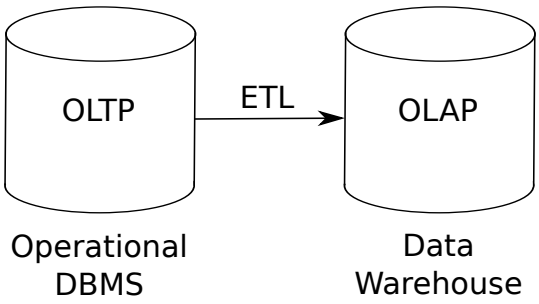


Figure 1.1: Traditional Separation of OLTP and OLAP

An Extraction, Transformation and Load process (ETL) is used to extract data from the operational databases periodically, perform transformations (e.g. aggregations) and load the transformed data into the data warehouse. As a consequence, the ETL process introduces a delay until updates become visible for analytical processing in the data warehouse. Modeling and optimizing the ETL process is an area of active research, e.g. see publications from researchers lead by Umeshwar Dayal

at HP Labs on modeling the ETL process [117] and ETL design for performance, fault-tolerance and freshness [100]. There is a trade-off between the additional load caused by extracting, transforming and loading the data and the business needs to analyze up-to-the-minute data, because the ETL process introduces high overheads on both systems. There is recent work on reducing the performance impact based on advanced change data capture and queuing techniques, see [86] for example. But, the separate system approach still incurs administration and maintenance costs for at least two systems and the complex ETL process. An advantage of this approach is that specific data representations can be used to suit the different workloads. Typically, normalized tables are used for OLTP and star-schema for OLAP (see chapter 17 in [62]).

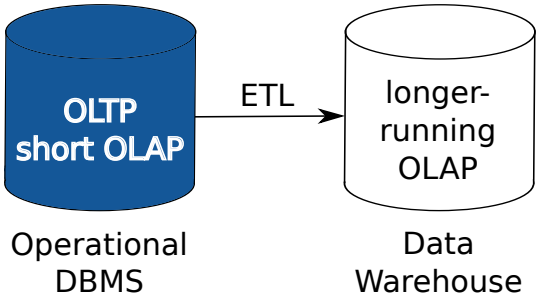


Figure 1.2: Operational Business Intelligence

Today, there is a trend towards Operational Business Intelligence, where some analytical queries are processed on the operational DBMS, in order to reduce the delay until updates become visible for analytical processing (see Figure 1.2). The goal is to process analytical queries directly on the operational database without impeding mission-critical transaction processing. This trend is driven by advances in hardware architecture. Hasso Plattner, the co-founder of SAP, advocates the "information ... at your fingertips"-goal [88] and Curt Monash discusses several use cases for so-called "low-latency analytics", including BI dashboards and interactive customer response for upselling and antifraud[3]. Modern business applications provide Operational Business Intelligence features, like analytic dashboards, in addition to traditional transaction processing, like order processing. Data warehouses will still be needed for very complex and long-running OLAP queries, which would cause too much resource contention on the operational databases. But by performing periodic

[3] `http://www.dbms2.com/2011/04/10/use-cases-for-low-latency-analytics/` (retrieved 08/28/2012)

and short-running queries on an Operational Business Intelligence system, the update cycles of data warehouses can be kept large enough for processing long-running queries efficiently.

Operational Business Intelligence results in mixed workloads (OLTP and OLAP) on the Operational DBMS. Managing these mixed workloads poses a big challenge for current disk-based DBMSs, as discussed by Krompass et al. in [70]. A DBMS that fulfills the ACID properties allows several users to work on the same data concurrently without interfering with each other, as described in chapter 9 of [62]. To make this feasible, the user has to specify begin and end of units of work, called database transactions. The highest isolation level, serializability, assures that a user does not see changes made by other users during a database transaction. For OLTP-only workloads, commercial DBMSs have support for transactions and serializability and can be configured to fulfill the ACID properties. For OLAP-only workloads, it is even easier to fulfill these properties, as data warehouses are optimized for read-mostly workloads and updates are typically applied during load windows when no queries are performed. But for mixed workloads it is a big challenge to fulfill these properties, as OLTP and OLAP workloads have very different characteristics. On the one hand, OLTP workloads consist of many short-running business transactions which read, insert, update and delete data in the database. On the other hand, OLAP workloads consist of longer-running read-only business queries. For mixed workloads, many short OLTP transactions, which make changes to the database, conflict with longer-running read-only OLAP queries. This incurs heavy synchronization overhead which negatively affects performance and results in low overall resource utilization. Stonebraker et al. stipulate the end of general-purpose DBMSs and postulate special-purpose DBMSs for specific application scenarios [105]. Business applications, like CRM, with Operational Business Intelligence features may be such an application scenario.

1.2 Cloud Computing

The buzz word *Cloud Computing* describes an on-going trend to provide more and more services over the Internet. The name comes from the cloud symbol that is often used in architecture diagrams to represent the Internet. According to the National Institute of Standards and Technology (NIST), "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. ... applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [78]. Cloud Computing attracts a lot of attention, because of its goal to reduce up-front and operational costs by enabling economies of scale.

1.2.1 Service Models

According to the NIST definition [78], Cloud Computing covers three service models which differ in the level of the provided service in the software and hardware stack of an application.

Infrastructure-as-a-Service (IaaS) provides fundamental computing resources, like processing, storage and networking, as a service over the Internet. The key enabling technology is virtualization, that allows to provide these resources in the form of virtual machines. On the one hand, virtualization allows to install and run arbitrary software on these virtual machines, including operating systems and applications. With virtualization based on binary translation or hardware virtualization support [2], neither operating systems nor application software need to be modified. Even legacy applications can be migrated to virtual machines. On the other hand, virtualization impedes customers from accessing the underlying physical infrastructure which is managed and controlled by the service provider. This is the major difference to traditional infrastructure hosting. A major advantage of IaaS is that physical resources can be shared between customers to improve resource utilization and to enable economies of scale. Furthermore, the provided infrastructure is elastic in the sense that additional infrastructure components, like virtual machines and virtual network components, can be added on demand with minimal service provider interaction, e.g. via a Web-Service interface. Thus, cloud applications do not have to be provisioned for peak loads, but can be scaled-out (and scaled-in) on demand. The risk of over-provisioning and under-provisioning is transferred to the service

provider, as only requested virtual resources are charged for. Typically, pricing is based on resource usage, e.g. runtime of virtual machines and data traffic, according to the pay-per-use principle. Resources can be added elastically on demand and only used resources have to be paid for. This approach is summarized by the marketing slogan pay as you go and turns utility computing into reality, as described by Armbrust et al. in [6]. Examples for IaaS are Amazon Elastic Compute Cloud[4] and Rackspace Cloud Hosting[5]. Today, many IaaS offerings provide virtual machine templates for various application scenarios which blur the differentiation between IaaS and other forms of Cloud Computing.

Platform-as-a-Service (PaaS) provides a platform for developing and deploying custom applications on the cloud infrastructure of the service provider. Any application may be built within the limitations of the provided platform. Customers do not have to manage or maintain the underlying infrastructure, including operating systems and software platform. Tedious tasks like applying patches to the operating systems and software platform are taken care of by the service provider. The major disadvantages of PaaS are that only tools and programming languages supported by the service provider can be used for creating these custom applications and that legacy applications may require significant changes in order to use PaaS, if it is possible at all. Typically the service provider provides proprietary frameworks and APIs that may enable applications to scale elastically. But there may be limitations depending on the used data store, e.g. Microsoft SQL Azure [23] or Microsoft Azure Tables[6]. Today, pricing for PaaS is often done similar to IaaS. This means that the customer has to pay for resources consumed by the application and the underlying software platform which may be difficult to predict and may change over time. Examples for PaaS are Microsoft Windows Azure[7], Google AppEngine[8] and Force.com[9].

Software-as-a-Service (SaaS) provides entire applications running on a cloud infrastructure. These cloud applications can be accessed over the Internet from various client devices typically using web interfaces, like web browsers. A service provider

[4] `http://aws.amazon.com/ec2` (retrieved 08/28/2012)
[5] `http://www.rackspace.com/cloud` (retrieved 08/28/2012)
[6] `http://www.windowsazure.com/en-us/home/features/data-management` (08/28/2012)
[7] `http://www.microsoft.com/windowsazure` (retrieved 08/28/2012)
[8] `https://appengine.google.com` (retrieved 08/28/2012)
[9] `http://www.salesforce.com/platform` (retrieved 08/28/2012)

owns and operates a standardized application that is accessed over the Internet by many users who correspond to different customers. The standardized application can be tailored according to customer needs within the limits of the provided customization features. By careful engineering, it is possible to leverage economy of scale to reduce TCO relative to on-premise solutions. In contrast to on-premise software, the service provider and not the customer owns and operates the entire application software and hardware stack (infrastructure, operating system, software platform and application). Customers only have to manage and control the client devices. In contrast to IaaS and PaaS, customers are typically charged a monthly service fee on a per-user basis. Today, a wide variety of business applications, including CRM are provided according to the SaaS model. Examples for SaaS are salesforce.com[10], SAP Business ByDesign [11], Google Docs[12] and Microsoft Office 365[13]. Today, CRM has the biggest share of the SaaS market. Therefore, we focus on this application scenario.

Our application scenario is standardized business applications, like CRM, with Operational Business Intelligence features. For this application scenario, SaaS seems to be the most promising approach, because many techniques for maximizing resource utilization while minimizing administration and maintenance costs can be applied when the entire application software and hardware stack is owned and operated by the service provider. On the one hand, the application, the software platform, the infrastructure and the management procedures can be highly optimized when focusing on one standardized application. On the other hand, SaaS providers employ multi-tenancy techniques to achieve economies of scale by consolidating several customers, referred to as tenants, onto a common software and hardware infrastructure. Consolidation may help to reduce administration and maintenance costs and improve resource utilization, but may cause resource contention between the workloads of different tenants.

[10] `http://www.salesforce.com` (retrieved 08/28/2012)
[11] `http://www.sap.com/solutions/technology/cloud/business-by-design` (08/28/2012)
[12] `http://docs.google.com` (retrieved 08/28/2012)
[13] `http://www.office365.com` (retrieved 08/28/2012)

1.2.2 Service Level Agreements

The consumer of a service typically requires a certain service level in order to be productive. For example, in telemarketing, there is a known number of agents that call prospective customers over the phone and have to enter gathered information into a CRM system during the call. In order to be productive in this scenario, end-to-end response times below one or few seconds are required for almost all of these business transactions and the required throughput depends on the number of agents. Furthermore, supervisors may use operational (or real-time) business intelligence features of the CRM product in order to monitor agent performance and to evaluate the success of marketing campaigns. For those analytical queries, the required throughput rate may be lower than for the business transactions. SaaS customers have to agree with the SaaS provider upon service level agreements (SLAs) which define the characteristics of the provided service including service level objectives (SLOs), like maximal response times and minimal throughput rates, and define penalties if these objectives are not met by the service provider. SaaS customers need to monitor SLA fulfillment and demand compensation if the promised service level is not met. Today, service providers already offer monitoring functionality, e.g. Amazon CloudWatch[14] and third-party vendors offer monitoring services, e.g. Hyperic Cloud-Status[15]. But in order to enable monitoring at the level of business transactions and analytic queries, application specific solutions may be required. Service providers need to offer the required interfaces, but the actual monitoring should be performed under the control of the customer or an independent service provider.

1.2.3 Deployment Models

According to the NIST definition [78], there are different deployment models for Cloud Computing. We focus on the public cloud model. According to this deployment model, the cloud service is owned by an organization selling cloud services which makes the cloud service available to the general public or a large industry group. The public cloud deployment model seems very suitable for the SaaS service model, as the same application may be provided to many different customers,

[14] `http://aws.amazon.com/cloudwatch` (retrieved 08/28/2012)

[15] `http://www.hyperic.com/products/cloud-status-monitoring` (retrieved 08/28/2012)

especially in the context of standardized business applications. There are concerns about the security of public clouds, as the service providers have to take care of information security for the provided service and customers have to trust them. For our application scenario this seems to be less of an issue, considering the success of services like salesforce.com.

1.3 Modern Hardware Architecture

In order to maximize resource utilization on modern hardware architectures, an application has to be optimized for multi-core CPU architectures and large main-memory. For business applications, this can be achieved by building them on-top of modern database systems. In the database community, there is an on-going trend towards database systems which keep most or even all data in main-memory. This trend is fueled by the increasing amount of main-memory available in off-the-shelf servers, which is increasing faster than the data volume that is required for storing all information contained in business applications of small and mid-sized businesses. For example, Intel announced already in 2011 a large multi-core processor which supports more than 1 TB of main-memory, as part of the so-called Tera-Scale initiative[16]. According to statistics published by salesforce.com [77], a single off-the-shelf server that is available today provides enough main-memory to accommodate all CRM data of several small to mid-sized businesses. Furthermore, emerging database technology can store large amounts of data in main-memory, in compressed form, and can process queries directly on compressed data [107].

Main-memory database architectures could be the right means to tackle the challenge that Operational Business Intelligence and mixed workloads pose for disk-based database systems. In main memory DBMSs, data can be accessed without disk I/O. Therefore requests can be processed at much smaller time scales and there is less variation in execution times. With a modern main-memory DBMS, typical business transactions like order entry or payment have execution times of less than 100 microseconds. This was shown by researchers led by Mike Stonebraker with the research prototype H-store [61, 105]. A commercial successor, called VoltDB[17], is now offered by a startup company. But main-memory is an expensive resource which has to be utilized efficiently. In our SaaS scenario, a space overhead of a factor of two could double hardware and energy costs as only half as many tenants can be accommodated on a given infrastructure if there is sufficient processing capacity and main-memory capacity is the bottleneck.

Today, a typical cloud computing infrastructure consists of a farm of commodity servers with multi-core architecture and large main-memory [12, 40]. The challenge

[16] `http://www.intel.com/go/terascale` (retrieved 08/19/2011)
[17] `http://www.voltdb.com` (retrieved 08/28/2012)

is to minimize maintenance and administration costs while maximizing resource utilization. A SaaS database system with automated administration procedures can help to reduce administration costs. Furthermore, maintenance costs can be reduced by building the SaaS application on-top of a SaaS database system which supports on-line application upgrades. Moreover, a SaaS database system should support multi-tenancy to allow for consolidation and should be optimized for large main-memory and CPUs with multi-core architecture to utilize such an infrastructure efficiently.

1.4 Contributions and Outline

For SaaS it is common practice to employ a multi-tenant architecture in order to reduce costs by consolidating several customers onto the same hardware and software infrastructure. In **Chapter 2**, we introduce multi-tenancy and give an overview on different implementation options. We point out that multi-tenant applications need a certain schema flexibility and that current techniques based on application owned schemata have severe drawbacks. We argue that these issues should be solved by integrating multi-tenancy support into the DBMS and propose an integrated model that allows capturing the evolution and extensibility of a SaaS application explicitly, including data sharing. Our major contribution is the proposed data model which supports branching in the evolution dimension and enables seamless upgrades that may help to reduce administration and maintenance costs significantly. The proposed data model eliminates redundancy by decomposition. Thus, model components have to be overlaid to derive "virtual" relations and partial overlays may be materialized to improve performance. We formulate the question which partial overlays should be materialized as an optimization problem.

In **Chapter 3**, we give an overview on emerging cloud data management solutions and analyze their suitability for SaaS business applications like CRM. These solutions have interesting features with regard to scalability and availability. Thus, we analyze how multi-tenancy can be realized with such systems and propose a multi-tenant schema mapping approach for one of these systems, namely Apache HBase. But, we assume that the resource requirements of any tenant can be met by a single server, which is the case for business applications of many small to mid-sized businesses according to published statistics. Therefore, we compare the single-server performance of an open-source cloud data management solution with a commercial DBMS. We conclude that many independent DBMS instances on a large server farm with automated administration procedures may be sufficient for multi-tenant SaaS business applications and our experimental results suggest that this approach may achieve better performance.

Operational Business Intelligence results in mixed workloads on the operational database. In **Chapter 4**, we discuss the characteristics of mixed workloads and the specifics of our application scenario. We point out the challenges posed by mixed workloads and give an overview on techniques for handling mixed workloads. Our

major contribution is a special purpose main-memory DBMS prototype that allows processing the mixed workload of our application scenario with low response times at high throughput rates while minimizing space overhead and adhering to maximal response time and minimal throughput guarantees. These guarantees enable strict SLAs and by minimizing space overhead, consolidation can be maximized in our multi-tenancy scenario. We present the underlying techniques: non-tree tier SaaS architecture, queuing approach, special-purpose logging approach, analytical model and resource reservation. Finally, our experimental evaluation shows that this approach is feasible for the mixed workload of our application scenario.

In **Chapter 5**, we present a benchmark for analyzing the suitability of database systems for mixed workloads and Operational Business Intelligence. CH-benCHmark combines transactional load based on TPC-C order processing with decision support load based on a TPC-H-like query suite run in parallel on the same tables in a single database system. Just as the data volume of actual enterprises tends to increase over time, an inherent characteristic of this mixed workload benchmark is that data volume increases during benchmark runs, which in turn may increase response times of analytic queries. Thus, the insert throughput metric for the transactional component interferes with the response-time metric for the analytic component of the mixed workload. We highlight this problem and discuss possible solutions including normalized metrics that account for data volume growth.

In **Chapter 6**, we present an approach for improving the resource utilization of emerging main-memory database systems that handle mixed workloads, by temporarily running other applications on the database server using virtual machines. In contrast to traditional multi-tenancy, this approach can be applied even if there are no complementary database workloads. We propose an elastic workload management approach and an analytical model to derive attractive service level objectives that can be met by a main-memory DBMS despite being co-located with arbitrary applications running in VMs. Furthermore, we propose an elastic scheduling approach called GOMA that tries to make spare resource better usable for co-locating VMs. We developed an elastic workload management extension for HyPer, an emerging main-memory DBMS that supports the mixed workloads of our operational business intelligence scenario. The experimental evaluation shows that GOMA may improve progress of co-located VMs significantly.

Chapter 7 concludes this thesis and gives an outlook on future challenges.

1.5 Previous Publications

SIGMOD 2009 Stefan Aulbach, Dean Jacobs, Alfons Kemper and Michael Seibold. *A Comparison of Flexible Schemas for Software as a Service.* In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), June 29 - July 2, 2009, Providence, Rhode Island, USA.

ICDE 2011 Stefan Aulbach, Michael Seibold, Dean Jacobs and Alfons Kemper. *Extensibility and Data Sharing in Evolving Multi-Tenant Databases.* In Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE), April 11 – 16, 2011, Hannover, Germany.

CLOUD 2011 Michael Seibold, Dean Jacobs and Alfons Kemper. *Strict SLAs for Operational Business Intelligence.* In Proceedings of the 4th IEEE International Conference on Cloud Computing (CLOUD), July 4 – 9, 2011, Washington DC, USA.

DBTest 2011 Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon and Florian Waas. *The mixed workload CH-benCHmark.* In Proceedings of the 4th International Workshop on Testing Database Systems (DBTest), June 13, 2011, Athens, Greece.

TPCTC 2011 Florian Funke, Alfons Kemper, Stefan Krompass, Harumi Kuno, Thomas Neumann, Anisoara Nica, Meikel Poess and Michael Seibold. *Metrics for Measuring the Performance of the Mixed Workload CH-benCHmark.* In Proceedings of the 3rd TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC), August 29, 2011, Seattle WA, USA.

DBSpektrum 2012 Michael Seibold and Alfons Kemper. *Database as a Service.* Datenbank-Spektrum 12(1): 59-62 (2012)

NOMS 2012 Sebastian Hagen, Michael Seibold and Alfons Kemper. *Efficient Verification of IT Change Operations or: How We Could Have Prevented Amazon's Cloud Outage.* In Proceedings of 13th IEEE/IFIP Network Operations and Management Symposium, Apr 16-20, 2012, Maui, Hawaii, USA.

CNSM 2012 Sebastian Hagen, Weverton Luis da Costa Cordeiro, Luciano Paschoal Gaspary, Lisandro Zambenedetti Granville, Michael Seibold and Alfons Kemper. *Planning in the Large: Efficient Generation of IT Change Plans on Large Infrastructures.* Accepted for publication in Proceedings of 8th International Conference on Network and Service Management (CNSM), October 22-26, 2012, Las Vegas, USA.

CLOUD 2012 Michael Seibold, Andreas Wolke, Martina Albutiu, Martin Bichler, Alfons Kemper and Thomas Setzer. *Efficient Deployment of Main-memory DBMS in Virtualized Data Centers.* To appear in Proceedings of 5th IEEE International Conference on Cloud Computing (CLOUD), June 24-29, 2012, Honolulu, Hawaii, USA.

IT Professional 2012 Michael Seibold, Dean Jacobs and Alfons Kemper. *Operational Business Intelligence: Meeting Strict Service Level Objectives for Mixed Workloads.* IT Professional (accepted for publication, May 2012)

ICDE 2013 Michael Seibold and Alfons Kemper. *GOMA: Elastic Workload Management for Emerging Main-memory DBMSs.* Submitted to 29th IEEE International Conference on Data Engineering (ICDE), April 8-11, 2013, Brisbane, Australia.

Chapter 2

Multi-Tenant DBMS for SaaS

For SaaS it is common practice to employ a multi-tenant architecture in order to reduce costs by consolidating several customers onto the same hardware and software infrastructure. Instead of provisioning for peak loads, several tenants with complementary workloads may be co-located onto the same machine or even database instance to improve consolidation and achieve better utilization of the common infrastructure. In this chapter, we introduce multi-tenancy and give an overview on different implementation options. We point out that multi-tenant applications need a certain schema flexibility and that current techniques based on application owned schemata have severe drawbacks. We argue that these issues should be solved by integrating multi-tenancy support into the DBMS and propose an integrated model that allows capturing the evolution and extensibility of a SaaS application explicitly, including data sharing. Furthermore, the proposed data model supports branching in the evolution dimension and enables seamless upgrades that may help to reduce administration and maintenance costs significantly. Moreover, the proposed data model eliminates redundancy by decomposition. Thus, model components have to be overlaid to derive "virtual" relations and partial overlays may be materialized to improve performance. We formulate the question which partial overlays should be materialized as an optimization problem and present a simple approach for reorganizing the proposed data model in order to derive an improved runtime representation.

2.1 Multi-Tenancy

In this section, we introduce multi-tenancy and give an overview on different implementation options. Furthermore, we discuss to what extent tenant migration and schema flexibility is supported by these different implementation options. Moreover, we discuss virtualization in the context of multi-tenancy and conclude that multi-tenancy support should be integrated into next generation DBMS to facilitate economies of scale and minimize total cost of ownership.

2.1.1 Challenges

A multi-tenant system provides an application to several tenants and consolidates several tenants onto the same operational system to allow for pooling of resources. Thereby resource utilization may be improved, as it eliminates the need to provision each tenant for their maximum load. For SaaS it is common practice to employ a multi-tenant architecture to reduce total cost of ownership, e.g. salesforce.com [116]. But multi-tenancy can also be used for on-premise deployments, e.g. to consolidate different departments of a large organization.

A multi-tenant system should allow for multiple specialized versions of the application, e.g. for particular vertical industries or geographic regions. We refer to this capability as application extensibility. An extension may be private to an individual tenant or shared between multiple tenants. Furthermore, the multi-tenant system should anticipate application evolution with support for on-line application upgrades. Moreover, even more tenants may be consolidated onto the same infrastructure, by sharing common data between tenants. In a multi-tenant system there is high potential for common data, like application code, application meta-data, master data, default configuration data and public information catalogs, such as area code data or industry best practices. But, application extensibility and evolution is even more challenging when data is shared between different tenants in favor of more consolidation.

The multi-tenant system comprises software and hardware. Depending on the application scenario, the data and workload of several tenants may fit on a single server. However, the multi-tenant software should support scale out across a farm of shared-nothing servers (as defined in [40]), because it is not cost effective to increase the capacity of a single server indefinitely [12]. The distribution of data

across the server farm should be tenant-aware to benefit from data locality. For example, a tenant might be placed on a server that manages trial accounts initially, be moved to a small production server later and once the capacity of this server is not sufficient anymore, the tenant may be moved to a larger production server or split across several servers. A uniform framework for system administration can improve management efficiencies and thereby reduce operational expenditures as fewer personnel may be required. The administration framework should allow for automation of operations like adding tenants, removing tenants and moving tenants within the server farm.

Consolidation can reduce capital expenditures for hardware and software licenses, but may cause resource contention and introduce additional security risks. The multi-tenant system has to provide tenant isolation with regard to security and performance. On the one hand, access control mechanisms have to span application and database layer to ensure that data is processed according to adequate security policies, as described in [118]. On the other hand, contention for shared resources has to be controlled by the multi-tenant system to ensure that the workloads of co-located tenants do not interfere with each other or with administrative operations.

2.1.2 Implementation Options

Today, multi-tenancy features are usually provided by a middleware between the application layer and a standard DBMS. Basically, there are four different approaches for implementing multi-tenancy with a middleware and a standard DBMS: dedicated machine approach, shared machine approach, shared process approach and shared table approach. These approaches increasingly improve consolidation but make tenant isolation more challenging.[1]

Dedicated Machine Approach The dedicated machine approach represents the most basic form of multi-tenancy. Each tenant gets its own physical server which runs a single DBMS instance. Figure 2.1 introduces our running multi-tenancy example with two tenants (Tenant1 and Tenant2). According to the dedicated machine approach, there are two servers, one per tenant, and on these servers run indepen-

[1]The following overview of multi-tenancy is based on a paper by Aulbach et al. at SIGMOD 2008 [7] which formed the basis of a conjoint follow-up paper at SIGMOD 2009 [8]. A short overview on multi-tenancy is also included in our Datenbank-Spektrum journal article [93].

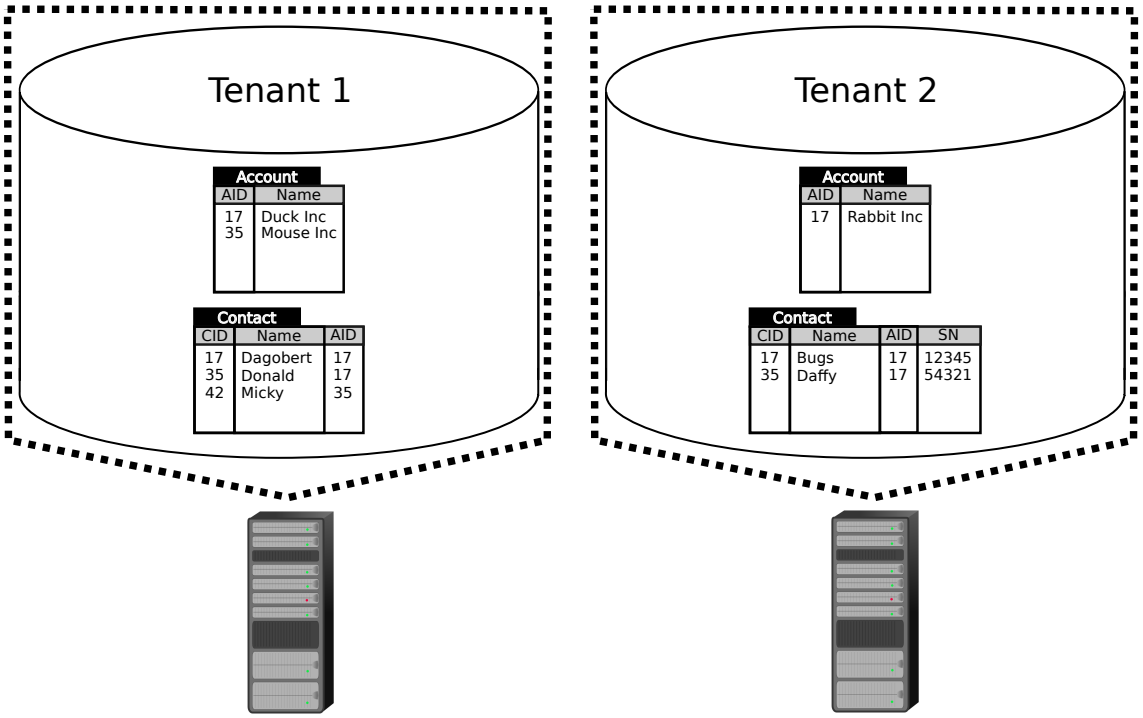


Figure 2.1: Dedicated Machine Approach

dent database instances. In our example, each tenant has two tables (Account and Contact), containing different data. Furthermore, Tenant2 has an additional column *SN* in the Contact table. The additional column represents a tenant-specific extension for managing information about the social network ID of contacts. We refer to this as the "social" extension. There is no consolidation and resource contention between tenants is avoided, as each tenant is isolated on its own physical machine. This approach seems appropriate for applications, like Enterprise Resource Planning that manage very sensitive data. Furthermore, a uniform administration framework for the entire server farm can improve management efficiencies and reduce operational expenditures. Moreover, if a single server is not sufficient for a given tenant, a dedicated database cluster may be used instead.

Shared Machine Approach With the shared machine approach, each tenant still gets its own DBMS instance, but several DBMS instances of different tenants run on the same physical server. Figure 2.2 shows a multi-tenancy layout of our running example according to the shared machine approach. In this case, there is only one server and both database instances run on the same server. As separate database processes are used for each tenant, tenants are isolated from each other

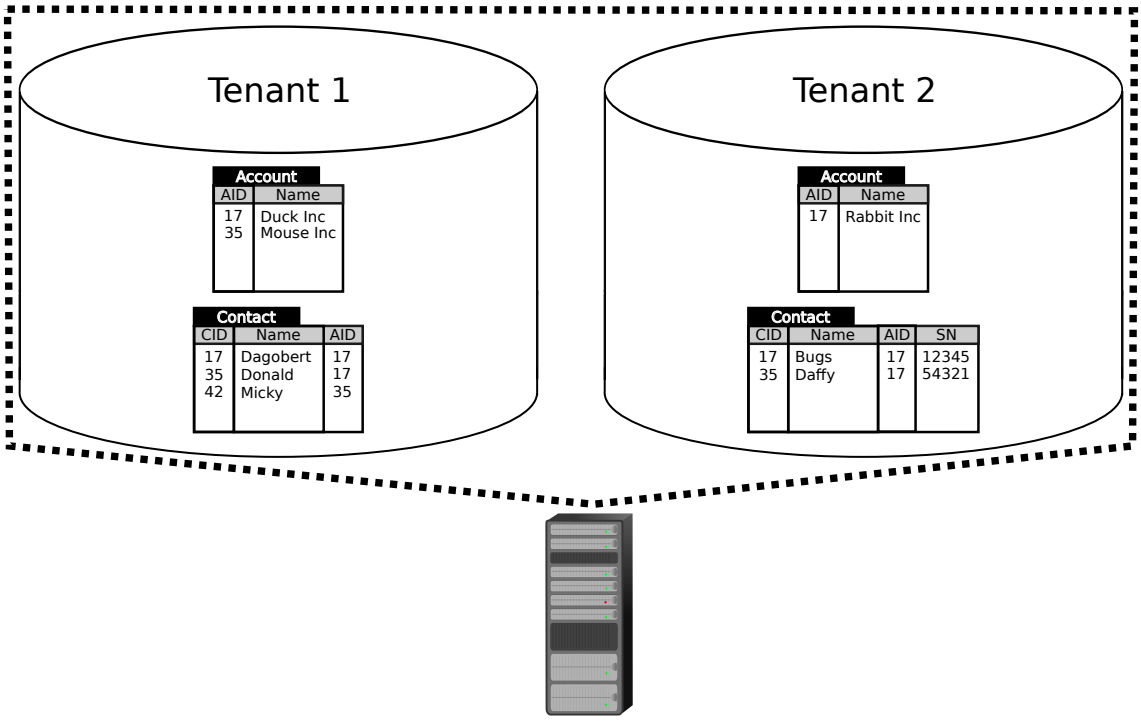


Figure 2.2: Shared Machine Approach

by using separate address spaces. But sharing the same machine between multiple tenants may introduce additional security risks, as security issues of one tenant may affect co-located tenants. Furthermore, resource contention between tenants may occur and has to be controlled by scheduling mechanisms of the operating system which executes the different database processes. Only low consolidation can be achieved with the shared machine approach, because separate DBMS instances do not pool resources between each other, e.g. memory is not pooled as each DBMS instance reserves its own buffer pools. The amount of consolidation that can be achieved with the shared machine approach depends on the required resources for a single tenant, e.g. data volume, and the capacity of a given server. For large tenants who require almost all resources of a given server almost all the time, the dedicated machine approach is more suitable. A good example for the shared machine approach is CasJobs, which creates private database instances — called MyDB — to allow researchers to analyze astrophysical data using a full-blown SQL interface and without interfering with each other. This approach allows researchers to work with a private database at the server-side without transferring high-volume data over Internet connections [83].

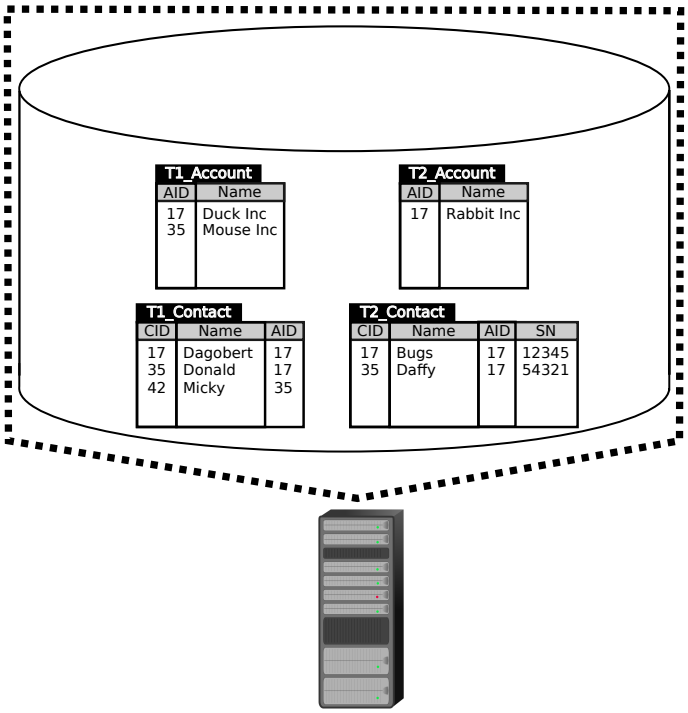


Figure 2.3: Shared Process Approach (= Private Table Layout)

Shared Process Approach Consolidation can be further improved with the shared process approach by sharing the same database process between multiple tenants and by assigning each tenant its own set of tables. To identify the tables of a given tenant, the table name may be prefixed with the tenant ID or a separate schema per tenant may be used. We also refer to this approach as the **Private Table Layout**, as tables are not shared between tenants. Figure 2.3 shows a multi-tenancy layout of our running example according to the shared process approach, also called private table layout. In this case, there is only one database instance running on one server and the table names are prefixed with a tenant ID. Isolation between tenants is lower than in the shared machine approach, as co-located tenants run in the same address space, because they use the same database processes. Instead of operating system mechanisms, standard DBMS mechanisms can be employed to isolate tenants from each other, e.g. separate user profiles. The shared process approach allows for more consolidation as resource pooling is improved by using only one DBMS instance per server. Memory and disk resources can be utilized more efficiently, e.g. by sharing buffer-pools between tenants and using a common write-ahead-log. However, the data of individual customers (or tenants) is still strictly separated into separate tables or even schemata, which causes a certain overhead

related to meta-data and index structures. The level of consolidation that can be achieved depends on the number of tables the DBMS can handle. On the one hand, DBMS performance degrades if there are too many tables. This effect is due to the large amount of memory needed to hold the associated meta-data, as well as the inability to keep index pages in the buffer pool. On the other hand, standard DBMS have certain limitations, e.g. allocate a certain amount of memory for each table up front.

Shared Table Approach Consolidation can be further improved with the shared table approach by storing the data from many tenants in the same tables of the same schema. Thereby the overhead related to meta-data and index structures is reduced. Instead of the number of tables, the limiting factor is now the number of rows the database can hold which is sufficiently large for most standard DBMS. But there are new challenges. With regard to performance, the interleaving of tenants' data breaks down the natural partitioning of the data and may compromise query optimization. First, table scans go across the data of all tenants. Second, optimization statistics aggregate across all tenants. Third, if one tenant requires an index on a column, then all tenants have to have that index. With regard to security, the interleaving of tenants' data complicates access control mechanisms which have to occur at the row level rather than the table level. Moreover, the shared table approach requires schema mapping techniques which map the logical schemata from multiple tenants into one physical schema in the database. The database only contains tables for the physical schema and these tables contain meta-data information apart from the data itself, in order to reconstruct the tables of the logical schemata. To make this approach transparent to application programmers, a middleware transforms queries against the logical schemata into queries against the physical schema. In the following we give an overview on the most important schema mapping techniques.

Basic Layout The basic layout allows tenants to share tables and adds a tenant ID column to each table to identify the owner of each row. To allow customers to extend the base schema, each table is given a fixed set of additional generic columns which may be of type VARCHAR or follow a certain type mix. Figure 2.4 shows our running example according to the shared table approach with basic layout. In this case, there are only two tables (Account and Contact) and the tables contain data

Account

Tenant	AID	Name
1	17	Duck Inc
1	35	Mouse Inc
2	17	Rabbit Inc

Contact

Tenant	CID	Name	AID	Ext1
1	17	Dagobert	17	
1	35	Donald	17	
1	42	Micky	35	
2	17	Bugs	17	54321
2	35	Daffy	17	54321

Figure 2.4: Shared Table Approach: Basic Layout

of both tenants. The Contact table contains three rows with NULL values in the *Ext1* column, because Tenant1 does not have the "social" extension. This schema mapping technique only requires a small, fixed number of tables which can provide performance benefits in comparison to private table layout, because a huge number of tables may impact DBMS performance negatively as discussed above. But the generic columns may contain many NULL values and special provisions may be required as commercial DBMS do not handle NULL values efficiently by default.

Account

Tenant	Row	AID	Name
1	1	17	Duck Inc
1	2	35	Mouse Inc
2	1	17	Rabbit Inc

Contact

Tenant	Row	CID	Name	AID
1	1	17	Dagobert	17
1	2	35	Donald	17
1	3	42	Micky	35
2	1	17	Bugs	17
2	2	35	Daffy	17

Social

Tenant	Row	SN
2	1	54321
2	2	54321

Figure 2.5: Shared Table Approach: Extension Table Layout

Extension Table Layout The extension table layout vertically partitions extensions into separate tables that are joined to the base tables using an additional *Row* column. Figure 2.5 shows our running example according to the shared table approach with extension table layout. In this case, there is one extension table for the "social" extension. This layout may require additional join operations which may

cause a certain performance overhead. But vertical partitioning also has its advantages, as extension tables only have to be read if they are really required by a query. Copeland et al. discuss the pros and cons of vertical partitioning based on their Decomposed Storage Model [31]. Nowadays, vertical partitioning has been adopted by column-oriented databases like C-Store [107] and its commercial successor Vertica[2] to improve the performance of analytical queries.

Tenant	Table	Row	Column1	Column2	Column3	Column4
1	Account	1	17	Duck Inc		
1	Account	2	35	Mouse Inc		
1	Contact	1	17	Dagobert	17	
1	Contact	2	35	Donald	17	
1	Contact	3	42	Mickey	35	
2	Account	1	17	Rabbit Inc		
2	Contact	1	17	Bugs	17	12345
2	Contact	2	35	Daffy	17	54321

Figure 2.6: Shared Table Approach: Universal Table Layout

Universal Table Layout The universal table layout uses a single physical table with a *Tenant* column, a *Table* column, a *Row* column and a large number of data columns with generic type, e.g. VARCHAR. As there is only one physical table, all physical rows have the same width and this may result in many NULL values if the width of logical tables varies. Figure 2.6 shows our running example according to the shared table approach with universal table layout. In this case, there is only one table, that contains all data, and there are several NULL values in *Column3* and *Column4*. This approach originates from the theoretical concept of the Universal Relation [75].

Pivot Table Layout The pivot table layout also uses a single physical table, but stores each value of a logical column in a separate physical row. Apart from *Tenant*, *Table*, and *Row* columns, the pivot table has an additional *Col* column that specifies which logical column a row represents and a single *Data* column. This layout does not require additional NULL values and furthermore NULL values of logical tables don't have to be stored explicitly. Figure 2.7 shows our running example according to the shared table approach with pivot table layout. In this case, there is only one table with only one *Data* column and there are no NULL values. Apart from eliminating

[2] `http://www.vertica.com` (retrieved 08/28/2012)

Tenant	Table	Row	Col	Data
1	Account	1	1	17
1	Account	1	2	Duck Inc
1	Account	2	1	35
1	Account	2	2	Mouse Inc
1	Contact	1	1	17
1	Contact	1	2	Dagobert
1	Contact	1	3	17
1	Contact	2	1	35
1	Contact	2	2	Donald
1	Contact	2	3	17
1	Contact	3	1	42
1	Contact	3	2	Micky
1	Contact	3	3	35
2	Account	1	1	17
2	Account	1	2	Rabbit Inc
2	Contact	1	1	17
2	Contact	1	2	Bugs
2	Contact	1	3	17
2	Contact	1	4	12345
2	Contact	2	1	35
2	Contact	2	2	Daffy
2	Contact	2	3	17
2	Contact	2	4	54321

Figure 2.7: Shared Table Approach: Pivot Table Layout

NULL values, the pivot table layout has the benefits of vertical partitioning, but causes space and performance overhead. The space overhead is due to the large amount of meta-data, as each row of the physical table contains four meta-data values and only one actual data value of a logical table. The performance overhead results from additional join operations that are required to reconstruct the logical tables. Each logical column apart from the first one requires an additional join.

This approach is based on a proposal by Agrawal et al. to "represent objects in a vertical format storing an object as a set of tuples. Each tuple consists of an object identifier and attribute name-value pair" [3]. Furthermore, they recommend to use a logical horizontal view of the vertical representation to hide complexity. Cunningham et al. [32] propose to implement Pivot and Unpivot as first-class RDBMS operations for better performance.

Chunk Table Layout and Chunk Folding Pivot table layout and universal table layout both use a single physical table, but the number of data columns differs greatly. Pivot table layout uses a single data column and the universal table layout uses a large number of generic data columns. Aulbach et al. [7] propose a compromise between these two extremes called chunk table layout. A chunk table

is similar to a pivot table but has several data columns of various types and uses a *Chunk* column instead of the *Col* column. Furthermore, Aulbach et al. [7] propose a technique called chunk folding that vertically partitions logical tables. With the goal to exploit the entire "meta-data budget" of the database, conventional tables are used for those parts of the logical schema that are most often accessed and chunk tables for the remaining parts. Recently, Grund et al. [54] published related work and propose a main memory hybrid database system that automatically partitions tables into vertical partitions of varying width depending on access patterns.

2.1.3 Tenant Migration

The resource requirements of tenants may change over time. Therefore, it may become necessary to migrate a tenant from one server of the server farm to a different server that has more free resources available. The presented implementation options differ with regard to tenant migration support.

With the dedicated machine approach and the shared machine approach, each tenant has its own database instance that can be migrated within the server farm by moving database files. To minimize downtime, a snapshot of the database files may be used in combination with log shipping. Similarly, the shared process approach allows to use a separate table space per tenant and to store different table spaces in separate database files. But a migration may affect performance for other tenants that share the same database instance. With the Shared Table Approach, it is more difficult to keep the data of different tenants separate, as several tenants share the same tables. Of course, it is always possible to export a tenant's data by querying the DBMS, but this may have a large impact on co-located tenants and may be much slower than moving files, especially if the data is not clustered by tenant ID.

There is recent work on database live migration that is similar to virtual machine live migration, but avoids virtualization overhead. Das et al. [38] propose a database live migration technique for scenarios where multiple tenants share the same database process — this corresponds to the shared process approach — and the persistent database image is stored on network attached storage. Elmore et al. [43] lift the latter restriction. Barker et al. [10] propose a throttling technique to minimize the performance impact of database live migration on other tenants.

2.1.4 Schema Flexibility

To support application extensibility and on-line application upgrades, a multi-tenant DBMS has to provide a certain schema flexibility. With the dedicated machine approach, the shared machine approach and the shared process approach, the schema is explicitly defined in the database. But commercial DBMSs only offer limited support for schema modifications on existing data and these operations may deteriorate performance, if they are supported at all. The impact of physical schema modifications on throughput for MS SQL Server has been studied in [8]. In contrast, the shared table approach requires a schema mapping technique and only the physical schema is defined in the database. The mapping between logical schemata and physical schema is done by the middleware. The major advantage of these "application owned schemata" is that logical schema changes can occur while the database is on-line, as no heavy-weight reorganization operations are required. The major disadvantage is that the underlying DBMS degenerates to a "dumb data repository" that only stores data rather than managing it. Experimental results, published in [8], show that application owned schemata cause a significant decrease in performance unless characteristic DBMS features, such as query optimization, are re-implemented in the middleware. But such a complex middleware may cause high maintenance efforts as functionality is duplicated in the standard DBMS and the middleware.

2.1.5 Virtualization and Multi-Tenancy

Virtualization ranges somewhere between the dedicated machine approach and the shared machine approach. By using a separate virtual machine for each tenant, tenant isolation is higher than in the shared machine approach. But it is lower than in the dedicated machine approach, as it may be possible to break out of a virtual machine with current virtualization technology. A recent example is US-CERT Vulnerability Note VU#649219[3], which describes a vulnerability that may be exploited for local privilege escalation or a guest-to-host virtual machine escape. Ray and Schultz [89] give an overview of risks associated with virtualization. Having said that, virtualization also has its benefits, like mature support for virtual machine live migration. Live migration allows migrating operating system instances

[3] `http://www.kb.cert.org/vuls/id/649219` (retrieved 08/28/2012)

across distinct physical hosts, including the applications running within an operating system instance and without remote clients having to reconnect [29]. When a separate virtual machine is used for each tenant, individual tenants can be moved between physical servers using live migration. Moreover, consolidation is higher than in the dedicated machine approach and may be further improved by advanced virtualization techniques, like transparent page sharing. Transparent page sharing allows sharing virtual memory pages that have the same content between virtual machines [115]. But according to Curino et al. [35], advanced multi-tenancy techniques can achieve more than twice the amount of consolidation than current virtualization technology. The major advantage of virtualization is that legacy applications can be consolidated without modification and that a good level of isolation can be achieved. But virtualization typically results in a complex software stack and may cause higher maintenance costs, as there are more possible sources of errors: each virtual machine runs its own operating system and there is the additional virtualization layer. In the following chapters, we therefore focus on a light-weight form of virtualization: a multi-tenant DBMS. Multi-tenant DBMSs help to improve resource utilization by consolidating databases with complementary workloads. In Chapter 6, we propose an approach to improve resource utilization of database servers even if there are no complementary database workloads, by temporarily running other applications on the database server using virtual machines.

2.1.6 Conclusions

A multi-tenant system faces the challenge to support application extensibility and evolution while employing consolidation and data sharing to minimize total cost of ownership. To achieve this, schema modifications over existing data have to be supported in an online fashion without affecting co-located tenants. Today, this kind of schema flexibility can be achieved with application owned schemata and the shared table approach. Several major SaaS vendors have developed mapping techniques in which the application owns the schema. This approach has become a design principle called "meta-data driven architecture", whereby application components are generated from meta-data at runtime [116]. But application owned schemata with generic structures in the DBMS break down the natural partitioning of the data. The problem is, that application owned schemata may cause a significant decrease in performance unless characteristic DBMS features, such as query optimization, are

re-implemented in the middleware. But such a complex middleware may cause high maintenance efforts as functionality is duplicated in the standard DBMS and the middleware. These issues could be solved by a new DBMS design with integrated multi-tenancy support. In our opinion, such a DBMS design should be based on the shared process approach and a "virtual" private table layout. As we propose a new DBMS design, the shared table approach and techniques like chunk table layout or chunk folding are not applied. But these techniques remain relevant for current standard DBMSs with limited support for online-schema modification and limited "meta-data budget".

2.2 Multi-Tenancy and SaaS

For SaaS it is common practice to employ a multi-tenant architecture in order to reduce total cost of ownership, e.g. salesforce.com [116]. According to Steve Bobrowski, multi-tenancy is a core cloud computing technology that controls how computing resources are shared among applications [17]. Figure 2.8 shows how multi-tenant business applications can be provided according to the SaaS model. The customers of the SaaS provider represent different tenants. Users who belong to different tenants use their web-browsers to access the SaaS application over the Internet. Business applications, like CRM, typically have a 3-tier architecture consisting of a web-server, an application-server and a DBMS. The SaaS provider consolidates several tenants into the same multi-tenant DBMS. There may be different application server instances, as different tenants may use different versions of the same standardized business application. The goal is to reduce total cost of ownership relative to on-premise solutions by consolidation and data sharing while allowing for extensibility and evolution.

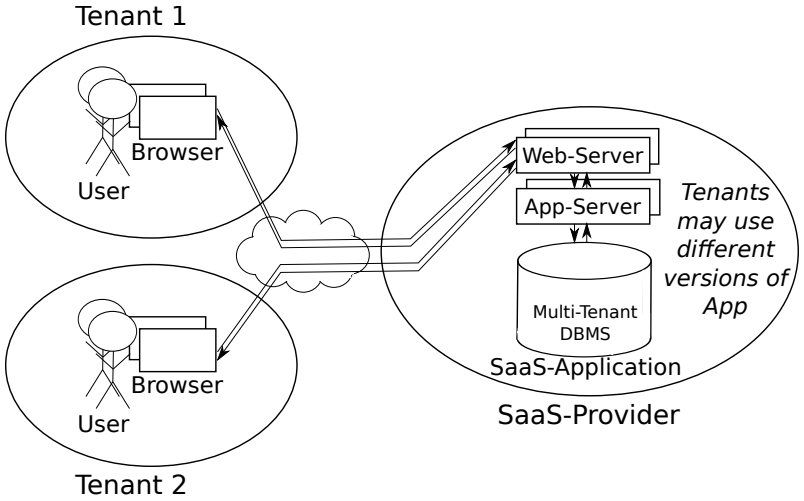


Figure 2.8: Multi-Tenant SaaS-Application

The following analysis of multi-tenancy in the SaaS context is based on research done in cooperation with Stefan Aulbach. Parts of this work have been published in a conjoint paper at ICDE 2011 [9].

2.2.1 Extensibility

For being competitive, a service provider has to leverage economies of scale by providing the same service to as many customers as possible. One prerequisite for increasing the customer base is to satisfy the needs of individual customers and customer groups. This prerequisite can be achieved by allowing customers to tailor the SaaS application according to their individual business needs. The extensibility dimension is made up of extensions to the common base application. These extensions may require database-schema changes by adapting predefined entities (e.g. adding attributes or changing types of attributes) and the possibility to add new entities. Extensions may be developed individually by customers themselves or by Independent Software Vendors (ISVs), and thus be shared between customers. Today, SaaS applications offer platforms for building and sharing such extensions, like force.com [116].

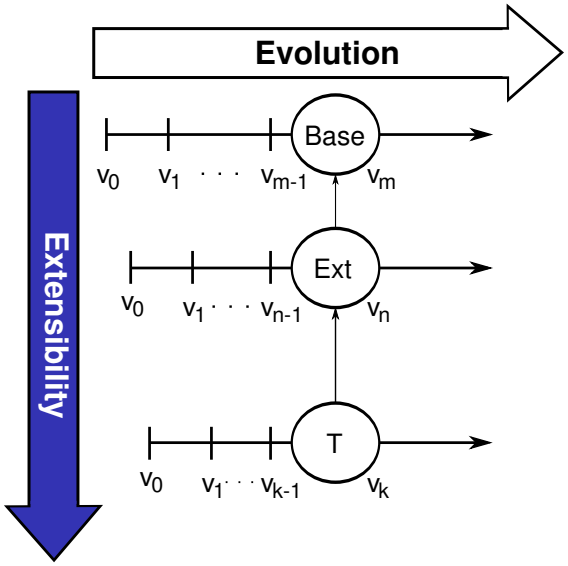


Figure 2.9: Extensibility

Figure 2.9 shows the SaaS application for a single tenant which is made up of the following components: the base application *Base* in version v_m, an extension *Ext* of an ISV in version v_n, and the tenant-specific extension T in version v_k. The components are developed and maintained separately from each other. Therefore, the releases of new versions are not synchronized. There may be dependencies between these components, as an extension may depend on a specific version of the base application or another extension. In the example, T in version v_k depends on *Ext* in version v_n and *Ext* in version v_n depends on *Base* in version v_m.

When the customers of a SaaS provider customize the SaaS application according to their individual business needs, the SaaS provider has to ensure that these extensions do not interfere with each other. Although most of the customizations may be small, managing the huge number of variants is a big challenge. It is not feasible for the service provider to manage each application instance separately, as the administration and maintenance costs would be similar to on-premise solutions multiplied by the number of customers of the service provider. A straight forward approach to improve manageability of many different application instances is to separate the base application and the extensions from the customer specific data and to maintain a master copy for each combination of base application version and extension versions used by any customer. But the number of different combinations which may be used by customers depends on the number of extensions used by a single customer and the number of available extensions, which may be pretty high. The best example is the SaaS CRM product salesforce.com, for which there are more than 1000 extensions, called "apps", available on the AppExchange marketplace[4].

2.2.2 Data Sharing

There is high potential for data redundancy between different instances of the same SaaS application. Even with a high degree of customization, big parts of the base application and ISV extensions can be shared across tenants: application code, application meta-data, master data, default configuration data and public information catalogs, such as area code data or industry best practices. SaaS providers are forced to minimize total cost of ownership and therefore already co-locate several customers on a single operational system, which lowers administration and maintenance efforts. Data sharing techniques may reduce space overhead and thus allow for more tenants to be consolidated onto the same hardware and software infrastructure.

In Figure 2.10, tenants T_1 and T_2 share version v_m of the base application $Base$ and version v_n of the ISV extension Ext. If the application would be managed separately for each tenant, the version v_m of the base application $Base$ and the version v_n of the ISV extension Ext would require twice the amount of space, as it would be part of the application instance of tenant T_1 and part of the application instance of tenant T_2.

[4] `http://appexchange.salesforce.com` (retrieved 08/28/2012)

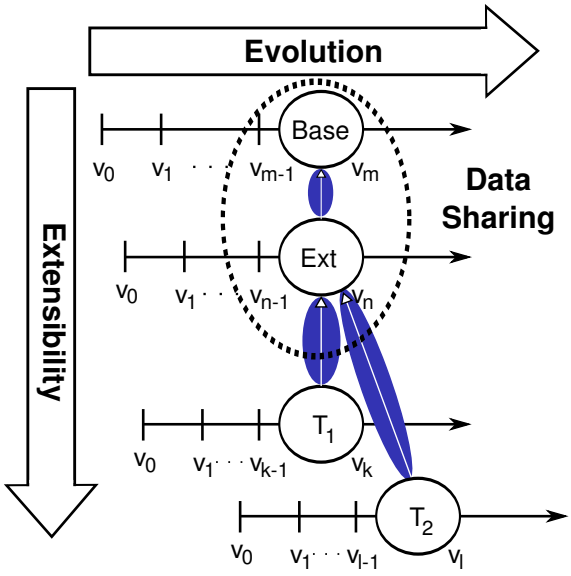


Figure 2.10: Data Sharing

SaaS enables optimizations which are not possible with on-premise solutions. For example, today many businesses use standardized software products for their on-premise business applications like CRM and ERP. The application instance of one business is very similar to the application instance of many other businesses that use the same version of the standardized software product, e.g. the executable code and the default configuration data are the same. Furthermore, master data may be very similar for businesses in the same economic sector. This data redundancy becomes more apparent, when these application instances are deployed on the same infrastructure as businesses adopt cloud computing. This data redundancy can be eliminated for SaaS applications by applying data sharing techniques.

For enabling data sharing in our multi-tenancy scenario, the ability to override common data with tenant-specific data is required. Shared data needs to be updatable, as already existing entries may have to be overwritten by tenants. Instead of replicating all data for each tenant, a small delta per tenant can be used if only a low percentage of the shared data is modified by a particular tenant. Current DBMSs do not support the functionality of data overriding in a transparent fashion.

2.2.3 Evolution

Another prerequisite to increase the customer base is to provide all functionality expected by prospective customers. This prerequisite can be achieved by incor-

porating new functionality into the provided service and by ensuring the security of the provided service. This may also require changes to the existing database schema. Therefore, SaaS applications are constantly evolving. The Evolution dimension tracks changes to the SaaS application which are necessary to either fix issues with the application or to integrate new features. Evolution is not only required for the base application itself, but also for extensions. The base application and its extensions may be developed and maintained separately from each other. Therefore, the releases of new versions are not synchronized.

For SaaS applications which allow extensions to a common base application, a major issue arises with regard to application upgrades. It may not be feasible for customers to always use the latest version of the provided common base application, because customer specific extensions and third-party extensions have to be checked for compatibility with the new version of the common base application and may require changes before the migration can take place. For a tenant, it is important to always have a consistent snapshot of the shared data. If shared data could change over time without the knowledge of the tenants, they would get unpredictable results. For many customers it may not be acceptable to upgrade their extensions according to the release schedule of the service provider. They may accept to pay a higher service fee if they don't upgrade right away, but they need the possibility to stay on a given version of the base application with which all their extensions work.

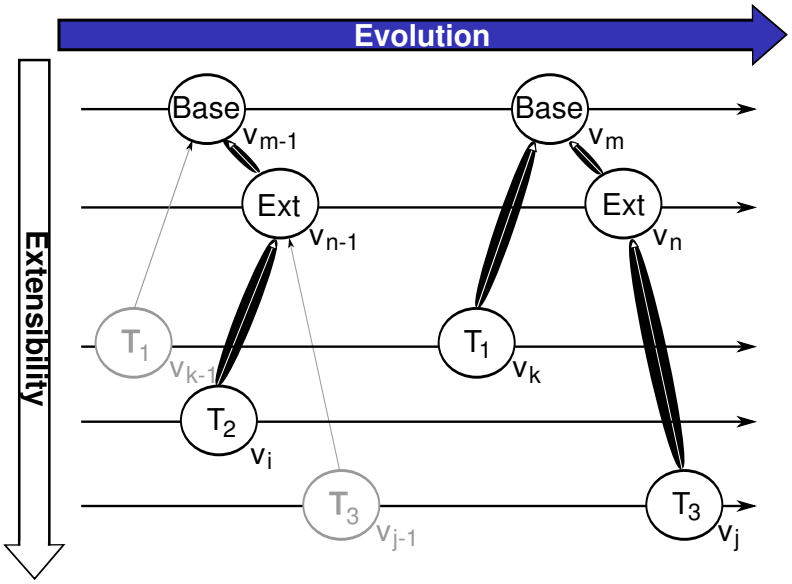


Figure 2.11: Evolution

Figure 2.11 depicts a SaaS application used by three tenants. It shows how the SaaS application and its extensions develop in the Evolution dimension. In this example, the base application has evolved from version $v_m - 1$ to version v_m and the extension Ext has evolved from version $v_n - 1$ to version v_n. Tenants T_1 and T_3 have already migrated to the new version of the base application and the extension, but Tenant T_2 is still using the old version of the base application and the extension, because the tenant-specific extension of T_2 has not been checked for compatibility with the new version of the extension and the base application yet. Once the tenant-specific extensions of Tenant T_2 have been checked for compatibility with the new version of the base application and the extension, Tenant T_2 can migrate to the new version of the base application and the extension. After this migration has happened, there is even more potential for data sharing, as all three tenants share the common base application in version v_m and tenants T_2 and T_3 share the common third-party extension Ext in version v_n.

2.2.4 Conclusions

The way applications develop according to the dimensions Extensibility and Evolution is important in the SaaS scenario, because different customers may be using different versions of the application with different extensions and the service provider has to provide all these different application instances at the same time. Furthermore, data has to be shared between tenants to allow for more consolidation. This is a big challenge for service providers, as administration and maintenance costs have to be minimized for being competitive.

Today, many service providers try to avoid these issues by the following measures. First, customers are forced to always use the latest version of the provided application. Second, only a very limited form of customization is provided. A typical limited form of customization is to provide a generic application which incorporates functionality for all customer groups and allow customization only in the form of configuration switches. This approach may be acceptable for some potential customers, but probably not for all. Today, many large and mid-size businesses already use standardized software products for their on-premise business applications like CRM and ERP. These standardized software products also provide a generic application which incorporates functionality for all customer groups and allow customization in the form of configuration switches. But apart from this simple form of customiza-

tion, many businesses tailor these standardized software products to their needs by changing the software themselves or by using third-party extensions. Often, there is a market around standardized software products for third-party extensions developed by ISVs. This extended form of customization is quite popular for on-premise software solutions and achieves levels of customization which cannot be done by configuration switches only. To enable these businesses to move from on-premise software to the SaaS model, SaaS applications have to provide more flexibility with regard to customization. Database-schema changes by adapting predefined entities (e.g. adding attributes or changing types of attributes) and the possibility to add new entities are required. Those modifications result in a high number of application variants, each individually customized for a particular customer.

A better approach would be to model the evolution and extensibility of a SaaS application explicitly. SaaS applications need to support Extensibility, Evolution and Data Sharing. These features can be characterized as multi-tenancy features and should be provided by a multi-tenant DBMS, as business applications are typically built on top of a standard DBMS. A multi-tenant DBMS needs an integrated model to capture these features. We propose such a model, which is described in the next section (Section 2.3).

2.3 Data-Model for Multi-Tenancy

A multi-tenant DBMS should have inherent support for the following multi-tenancy features: Extensibility, Data Sharing and Evolution. These capabilities are highly related. Therefore an integrated model is required to capture these features[5].

There already exist models which capture Extensibility, like the object-oriented concept of inheritance [65] and there are models for capturing the Evolution of an application, like the PRISM project for database-centric applications [36]. But to our knowledge there is no integrated model that captures both dimensions and Data Sharing together. Therefore, we propose such a data model for managing meta-data and shared data in a multi-tenant DBMS.[6]

Multi-tenant applications can be represented by a hierarchy of extensions to a common base application. In the proposed data model, Extensibility is captured by modeling schema hierarchies of multi-tenant applications explicitly. From these schema hierarchies a "virtual" private table layout can be derived for each tenant. The knowledge about the different schemata of a multi-tenant application can be used to improve consolidation by applying data sharing at all levels of the multi-tenant DBMS. The hierarchical schema representation can be used to share meta-data between tenants and shared data can be attached to this hierarchy. Different versions of this schema hierarchy can be used to model the evolution dimension. We propose a hierarchical data model for meta-data and shared data which captures extensibility, evolution and data sharing in one integrated model. This model can be used to automate administration tasks, like evolving a tenant from one application version to a different application version and compatible versions of the used exten-

[5] A similar approach has been proposed by Schiller et al. [91] at EDBT 2011. This research was done independent to our work and in parallel, as our ICDE 2011 paper submission deadline (July 23, 2010) was before the EDBT 2011 paper submission deadline (September 15, 2010). To our understanding, they only consider extensibility, but not evolution. Furthermore, the paper only mentions meta-data sharing, but not sharing of data (like default configuration or master data). Moreover, branching is not mentioned in the paper.

[6] An earlier version of this data model has been published in [9]. Polymorphic relations of the same relation history were condensed into one polymorphic relation whose instances contained fragment sequences and segment sequences to capture versioning of schema and shared data. This enabled a very compact representation, but required that dependencies between instance versions had to be managed separately on a per-tenant basis and branching in the evolution dimension was not supported.

sions. Thereby, application upgrades can be performed by the tenants themselves with minimal management effort and without service provider interaction. This can help to lower administration and maintenance efforts and thus decrease total cost of ownership. Furthermore, it can help to ease the maintenance of the application, as dependencies between extensions and application versions are modeled explicitly.

The proposed data model for managing meta-data and shared data in a multi-tenant DBMS adheres to the traditional relational data model, as the "virtual" private table layout of each tenant represents a traditional relational layout. We chose the traditional relational data model (without object-relational extensions), because it is the predominant data model as relational database systems are most widely used today. There is related work by Curino et al. [37] to support legacy applications when the database schema evolves, including support for legacy updates under schema and integrity constraint evolution. Schema evolution is relevant in our context, but updates on older versions of the application are only required at the level of individual tenants and only for the latest schema used by a given tenant. Evolution of integrity constraints is an interesting direction for future work.

2.3.1 Model Components

The proposed data model represents multi-tenant applications as a hierarchy of extensions to a common base application. Each relation from the databases of any tenant is decomposed into Instances according to this hierarchy and for a given relation the Instances of all tenants are combined to form a Polymorphic Relation. We use the term "polymorphic", because it contains the information of all tenants and the "virtual" relation of each tenant can be derived from a Polymorphic Relation. A Polymorphic Relation may change over time due to evolution. A Relation History comprises the different versions of a Polymorphic Relation. Branching, as known from revision control systems, is supported to capture the evolution of a Polymorphic Relation explicitly along the development history of the corresponding application.

Polymorphic Database A Polymorphic Database consists of a set of Relation Histories.

Relation History A Relation History consists of a set of Polymorphic Relations.

Polymorphic Relation A Polymorphic Relation consists of Instances and Inheritance Relationships between these Instances. An artificial primary key is used to identify tuples within a Polymorphic Relation.

Instance An Instance is made up of a local Fragment, which contains schema information, and a list of local Segments, which contain data items.

Inheritance Relationship An Instance has zero or one parent Instance and zero or more child Instances.

Predecessor Relationship An Instance has zero or one preceding Instance and zero or more succeeding Instances.

Fragment A Fragment contains a Transform Sequence and represents a list of N typed attributes.

Transform Sequence A Transform Sequence is a list of M Transforms.

Transform A Transform consists of a schema modification operator, as described in [36], and a mechanism to migrate data accordingly.

Segment A Segment contains data items that may be shared or private. The schema of a Segment consists of exactly one Fragment and the artificial primary key of the Polymorphic Relation. The Fragment may be a local Fragment or an inherited Fragment.

Figure 2.12 shows a Polymorphic Relation, that is used by three tenants (T_1, T_2 and T_3). T_1 extends the original relation from the *Base* application with a tenant-specific extension. T_2 and T_3 both extend an extension *Ext* of the relation with tenant-specific extensions.

Figure 2.13 shows a simplified Polymorphic Relation that is only used by tenant T_3 and corresponds to the path from the root to tenant T_3 in Figure 2.12. In the following figure (Figure 2.14), we use the simplified Polymorphic Relation to make it easier to understand how Relation Histories work.

Figure 2.14 shows a relation history that contains the simplified Polymorphic Relation from Figure 2.13 and another Polymorphic Relation that represents a newer

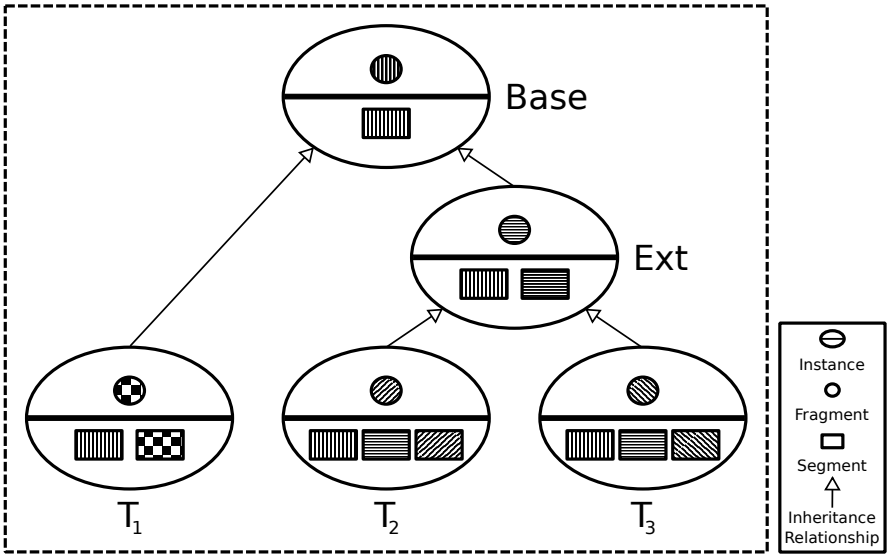


Figure 2.12: Polymorphic Relation

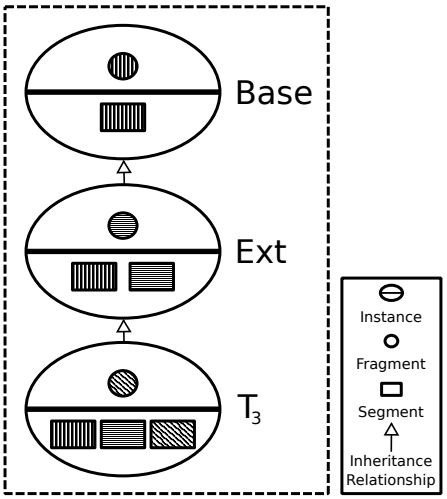


Figure 2.13: Polymorphic Relation Simple

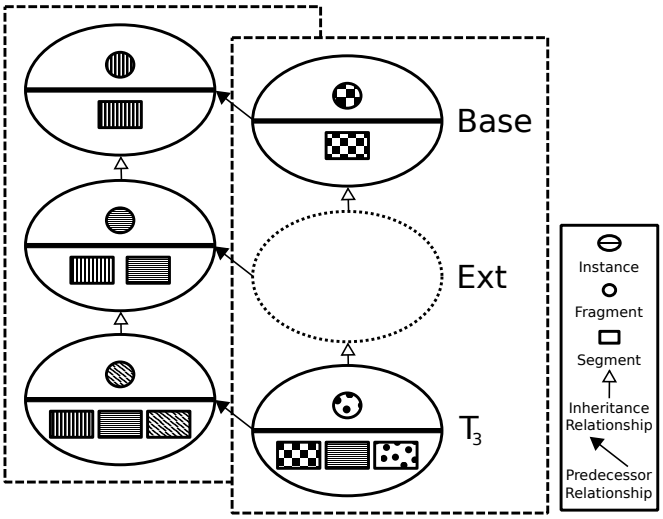


Figure 2.14: Relation History

version of the former Polymorphic Relation. In the latter Polymorphic Relation, the relation has changed at the level of the *Base* application and of tenant T_3. At the level of the extension *Ext*, the relation has not changed, therefore a place holder points to the earlier version of the *Ext* Instance.

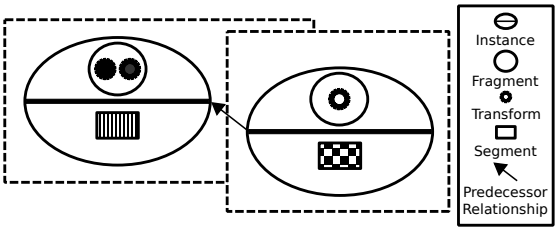


Figure 2.15: Fragment

Figure 2.15 shows a Relation History containing two versions of a Polymorphic Relation that consist of only one Instance each. The Predecessor Relationship between the two instances determines which of the two Instances is newer. The Instances contain one Fragment and one Segment each. Furthermore, the Transform Sequence of the older Fragment contains two Transforms and the Transform Sequence of the newer Fragment contains one Transform.

2.3.2 Data and Meta-Data Overlay

The proposed data model eliminates redundancy by decomposition. An Instance of a given Polymorphic Relation corresponds to a "virtual" relation of the common base application, a common extension or a tenant. In order to derive a "virtual" relation of a given tenant from a Polymorphic Relation, the components of the model have to be overlaid according to the Inheritance Relationships and the Predecessor Relationships. The Inheritance Relationship defines the Instance hierarchy. The model does not support multiple inheritance to avoid the associated complexity. Inheritance Relationships determine inheritance of schema and shared data, as Fragments and Segments are inherited according to the Inheritance Relationships. The Predecessor Relationship captures the changes between different Polymorphic Relations of the same Relation History.

Meta-Data Overlay Meta-data in the form of schema information is represented by Fragments. For a given Instance of a given Polymorphic Relation, the schema of the corresponding "virtual" relation can be derived by concatenating the attribute lists of inherited Fragments and the local Fragment. But Fragments do not store attribute lists explicitly. Instead, the attribute list of a Fragment is defined by Transform Sequences. The attribute list of a Fragment can be derived as follows. First, the Transform Sequences of preceding Fragments and the local Transform Sequence have to be concatenated (Fragment f_x of Instance x precedes Fragment f_y of Instance y if Instance x precedes Instance y according to predecessor relationships.) Second, the resulting list of Transforms has to be evaluated to determine the attribute list.

Data Overlay Data is represented by Segments. An Instance contains exactly one local Segment for each local or inherited Fragment. For a given Fragment there may be multiple local or inherited Segments. For a given Instance of a given Polymorphic Relation, the data of the corresponding "virtual" relation can be derived by generating virtual Segments and aligning these virtual Segments. First, partial Segments are generated by overlaying[7] each local Segment of the given Instance and all ancestor Instances with the corresponding local Segments of their preceding Instances. Second, the partial Segments (local or inherited) of each Fragment (local

[7] A similar overlay operator has been formally defined in [9].

or inherited) are overlaid. Both overlay steps have to adhere to the Data Overriding Precedence. The Data Overriding Precedence defines that data contained in Segments whose originating Instance is lower down in the instance hierarchy overwrites data contained in Segments whose originating Instance is higher up in the instance hierarchy. Furthermore, data contained in succeeding Segments overwrites data contained in preceding Segments. For proper alignment of the resulting virtual Segments, each data item in any Segment has to replicate the artificial primary key value of the corresponding tuple.

Shared and private data are handled similarly with regard to data overlay. Data overlay allows tenants to overwrite shared data without affecting other tenants, as the changes are stored as part of the tenants' private data. Furthermore, an extension can overwrite shared data and the changes only affect tenants or other extensions which inherit from the given extension. But there is a big difference between shared and private data. Changes to shared data have to be released as part of a new version of the base application or extension which is added to the Relation History as a new Polymorphic Relation. After release, shared data is read-only. The space overhead for changes to shared data is small, because the proposed data model stores only changes due to decomposition. But still we assume that changes to shared data are less frequent than changes to private data. Changes to private data of a tenant are immediately visible to the owning tenant and private data segments are writable until the tenant upgrades to a new version of the base application and/or extensions. Private data Segments have to be made read-only as part of an upgrade, as changes are tracked in succeeding Segments. This approach may cause space overhead for high update rates, but enables a low overhead roll-back mechanism which can be used by tenants to test-drive upgrades in separate branches.

2.3.3 Data Overriding

In the proposed data model, data overriding allows tenants or extensions to override shared data. Data Overriding is a new concept which is not supported by the original data models from which the proposed data model is derived.

For comparison to the object-oriented data model, an Instance can be seen as an Abstract Data Type, which only has attributes, and the corresponding set of objects. The instance hierarchy is similar to a class hierarchy, but the semantics are different:

in the object-oriented model, the inclusion polymorphism is specified by subtyping, i.e. each sub-type can be treated as its super-type. However, in the proposed data model this is not the case: The inclusion polymorphism goes from the leaf node to the root node. There is some similarity to the object-oriented concept of Method Overriding, but currently neither Object-Oriented DBMSs (OODBMSs) nor object-oriented programming languages support the functionality of Data Overriding. The reason may be that Data Overriding conflicts with Object Identity, as a tuple of a tenant overrides a tuple of the base relation or an extension by specifying the same primary key value.

Relational DBMSs extend the relational data model with Views. But updates to Views are typically restricted or even prohibited. Moreover, an update of an updatable View results in changes to the relations which contribute to the view. This would mean that shared data is overwritten for all tenants and not only for the updating tenant as required by Data Overriding.

2.3.4 Branching

The proposed data model supports branching in the evolution dimension, as an instance may have more than one successor according to the Predecessor Relationship. Branching is known from revision control systems and may become more and more important for SaaS, as even the development of cloud applications moves to the cloud. Recently, an open-source project, called Eclipse Orion[8], has started to develop a web-based development environment that runs in the cloud and is accessed via a browser-based interface. Branching allows capturing the evolution of a Polymorphic Relation along the development history of the corresponding application.

Figure 2.16 shows another example where branching in the evolution dimension is beneficial. At a given point in time, tenant T_1 was using the *Base* application in version v_{m-1} and tenants T_2 and T_3 were using version v_{n-1} of an extension *Ext* to this *Base* application version. Afterwards, a new version of the *Base* application (v_m) and of the extension *Ext* (v_n) was released and tenants T_1 and T_3 migrated to the new versions. Tenant T_2 did not migrate to the new version of the *Base* application and extension *Ext* right away, because there were incompatibilities between the new version of extension *Ext* and tenant-specific extensions of tenant T_2. Furthermore,

[8] `http://www.eclipse.org/orion` (retrieved 08/28/2012)

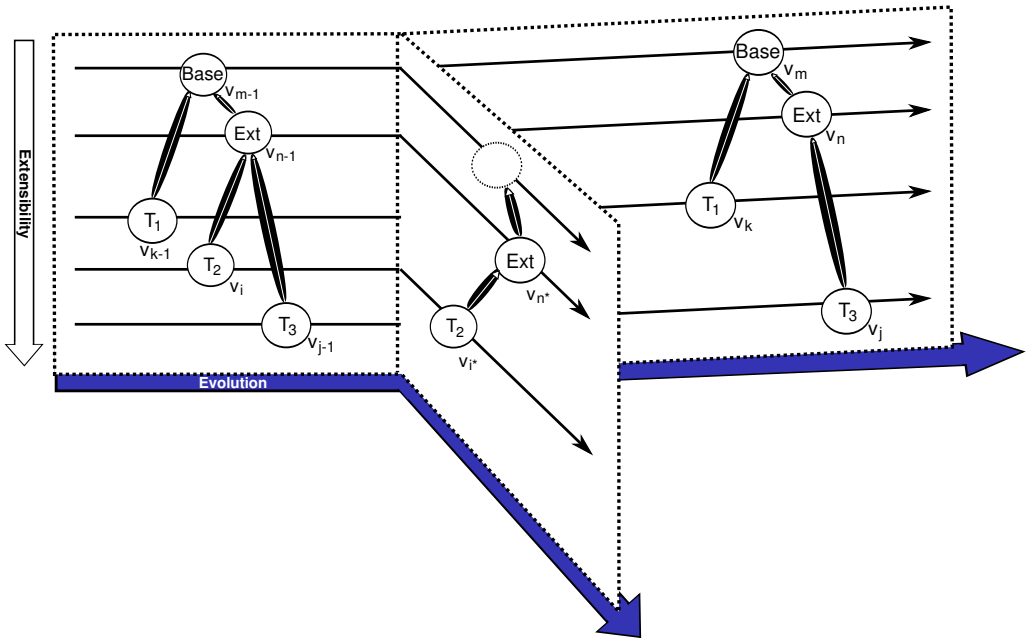


Figure 2.16: Branching

it turned out that the migration effort would be considerable although tenant T_2 did not need any of the new features of the new version. Therefore, tenant T_2 decided to stay with the older version for some time. But there were some security related changes in the new version of extension Ext, that did not cause incompatibilities with the tenant-specific extensions of tenant T_2, and that were relevant for tenant T_2. Therefore tenant T_2 decided to pay the ISV who developed extension Ext for back-porting the security-related fixes to the older version of the extension. The resulting version $v_{n'}$ of extension Ext was created in a separate branch of the evolution dimension and tenant T_2 migrated to that version. Because the data model supports branching explicitly, it was possible to enhance an older application version without affecting other tenants and without service provider interaction.

With branching, the evolution dimension can be seen as a graph of Instance versions. A differential data organization scheme would materialize some Instance versions corresponding to nodes in the graph. Apart from that, only Deltas corresponding to edges in the graph would be stored. Any Instance version in this graph could be materialized and it even would be possible to add new edges to the graph by computing the Deltas. This could be used to reduce the number of deltas on the shortest path from any materialized Instance to a target Instance. For a given workload, the graph could be augmented with node weights corresponding to

the popularity of an application version. Based on such a representation it may be possible to find the optimal combination of deltas and materializations for a given workload. In [9], we analyzed different options for physical data organization and assumed a typical SaaS workload, where the latest available version was materialized. Next generation SaaS applications could offer several branches, e.g. a current branch with the newest features and a stable branch with long term support. In this case it would make sense to materialize more than one Segment of a Segment Sequence, as two versions of the application would be very popular.

2.3.5 Data Organization

The proposed data model should be organized differently for archival purposes and at runtime.[9] The entire history of a multi-tenant application can be captured with the proposed data model, including schema changes and changes to shared and private data. This information should be archived as it can be used to automate administration and maintenance tasks. At runtime a more efficient data representation should be used which can be derived automatically from the archived representation of the data model. Using the archived representation directly would cause space and performance overhead. On the one hand, data overlay is required to derive the "virtual" relation which corresponds to a given instance of the proposed data model. This operation may cause performance overhead. The incurred performance overhead depends on the number of parent instances and the number of predecessor instances of the given instance and its parent instances. On the other hand, the different versions of the same relation are stored as a set of Polymorphic Relations in a Relation History. Due to decomposition, only the differences are stored. But if a tuple with a given artificial primary key changes between two versions, both versions are stored in the data model. This may incur significant space overhead, depending on the update rate and the number of versions stored in the model.

Space overhead and performance overhead can be reduced by reorganization. For archival purposes, the fully decomposed data model with all versions of all instances should be stored on secondary or tertiary storage. From this archived data model

[9] A discussion of physical data organization techniques can be found in [9]. We compared different physical data organization techniques and analyzed their suitability for seamless administration and maintenance mechanisms. Based on this analysis, we proposed an optimization based on differential XOR-Deltas.

any "virtual" relation in any version can be derived. At runtime, an optimized representation should be used. The runtime data model should only contain the versions of those instances which are currently used by active tenants. To further reduce performance overhead redundant copies of the data model can be materialized which are optimized for a subset of the active tenants. As tenants may become active and inactive over time and may upgrade to newer versions over time, the runtime data model has to be reorganized over time. This reorganization has to be performed regularly and therefore should be fully automated. As automatic reorganization is out-of-control of the tenants, it has to be performed transparently and in an on-line fashion without impacting performance for active tenants.

We predict that next-generation SaaS applications are based on a next-generation DBMS which actually manages the data and provides multi-tenancy capabilities. Next-generation DBMSs have to be optimized for emerging server hardware which provides big main-memory capacity and many processing cores. But such a main-memory DBMS has to use main-memory efficiently, as it is an expensive resource. Data that is not needed anymore has to be removed from main-memory as soon as possible. This is a big challenge. Especially in the multi-tenancy context, when data is shared between tenants. The proposed data model allows identifying what data is needed for a given set of tenants, that are using given versions of the SaaS application and its extensions.

Optimization Problem

For the proposed data model, the reorganization of the runtime representation can be formulated as an optimization problem.

The input to the optimization problem is the following. First, the archive representation of the data model. Second, information about active tenants and an estimation on which tenants will become active or inactive in the near future. Third, an estimation on when tenants will upgrade to newer versions of the base application and its extensions. Fourth, estimations on future resource requirements of tenants and finally resource requirements for reorganization.

The output to the optimization problem is the following. First, an assignment of tenants to servers. Second, information on how the runtime representation of each server should be reorganized. Third a schedule on when planned reorganizations shall be performed.

The optimization goals are the following. First, flexibility has to be ensured, as tenants need to be migrated between servers according to their future resource requirements. Furthermore, tenants need the ability to upgrade to newer versions of the base application and its extensions. Second, space overhead has to be minimized. Third, performance overhead has to be minimized. Fourth, the number of active servers has to be minimized in order to maximize the resource utilization of active servers.

The assignment of tenants with given resource requirements to servers with given capacity corresponds to the famous bin packing problem, which represents an NP-hard optimization problem.

Reorganization

In the following, we present a simple approach for reorganizing the proposed data model in order to derive an improved runtime representation. We make the following simplifications. First, tenants are either active or inactive. Second, each active tenant uses exactly one version of the application and its extensions. Third, we do not consider changing resource requirements of tenants. Fourth, we do not consider migration of tenants two newer versions of the application and its extensions. Fifth, we do not consider resource requirements of reorganization operations.

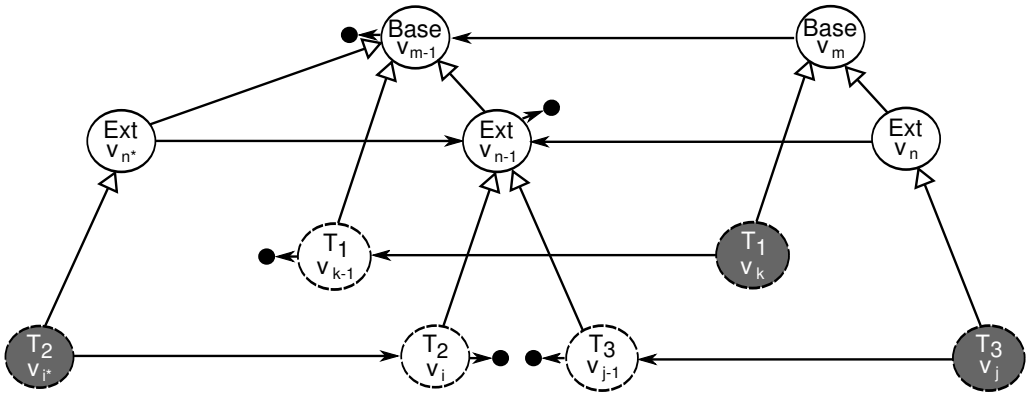


Figure 2.17: Model for Example from Figure 2.16

For reorganization, we simplify the proposed data model and represent it as a graph by using Instances as nodes and Predecessor and Inheritance Relationships as edges. The resulting graph consists of trees and we maintain this information by marking edges corresponding to Predecessor and Inheritance Relationships differently. Figure 2.17 shows how the example from Figure 2.16 can be represented as

a graph. In this figure, dummy root nodes (black dots) are used in the predecessor hierarchies. These dummy root nodes allow storing all data in the form of deltas on the edges of the Predecessor Relationships. Tenant-specific Instances are located at the leaf level of the inheritance hierarchies and gray leaf nodes represent current Instances used by active tenants.

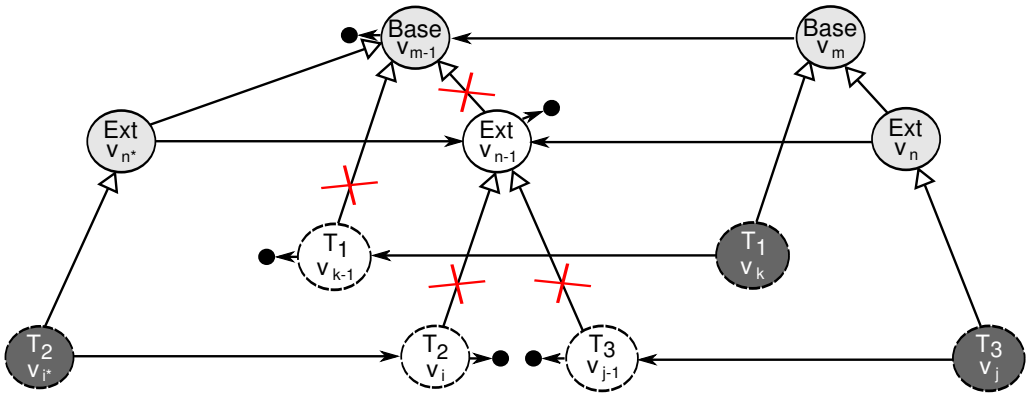


Figure 2.18: Reorganization Step One

For the runtime representation, we only need the ability to derive "virtual" relations for current Instances of active tenants (gray nodes in Figure 2.17). In order to derive a "virtual" relation for a given Instance, we need all Instances on the path from the given Instance up to the root of the corresponding Inheritance hierarchy and all predecessors of these Instances. In Figure 2.18, the nodes on the paths from a dark-gray node to the root of the corresponding inheritance hierarchy are marked light-gray. Moreover, Inheritance Relationships of white nodes can be removed to simplify the graph.

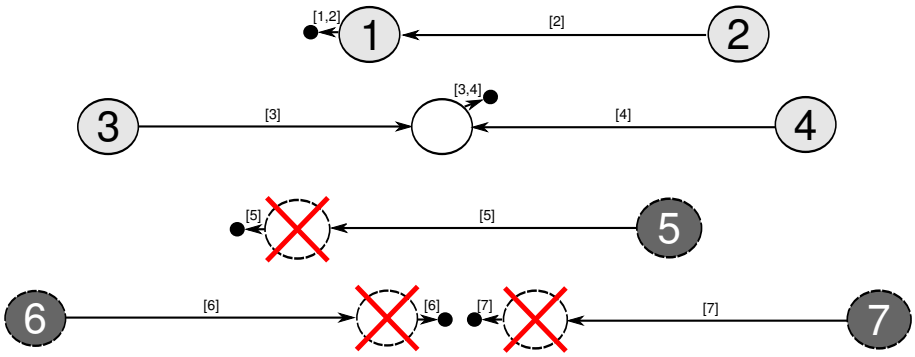


Figure 2.19: Reorganization Step Two

In the next step, we determine the predecessors of those Instances whose node is marked gray. Figure 2.19 only shows the Predecessor Relationships and the gray

nodes have been enumerated. We start at the gray nodes, go along the predecessor relationships and mark the visited edges with the number of the gray start node. If two successive Predecessor Relationship edges are marked by the same numbers, this corresponds to two successive deltas that are required by the same start nodes. Therefore, such edges can be combined and the node in-between can be removed. Of course, this requires that the deltas on the edges of the Predecessor Relationships are merged.

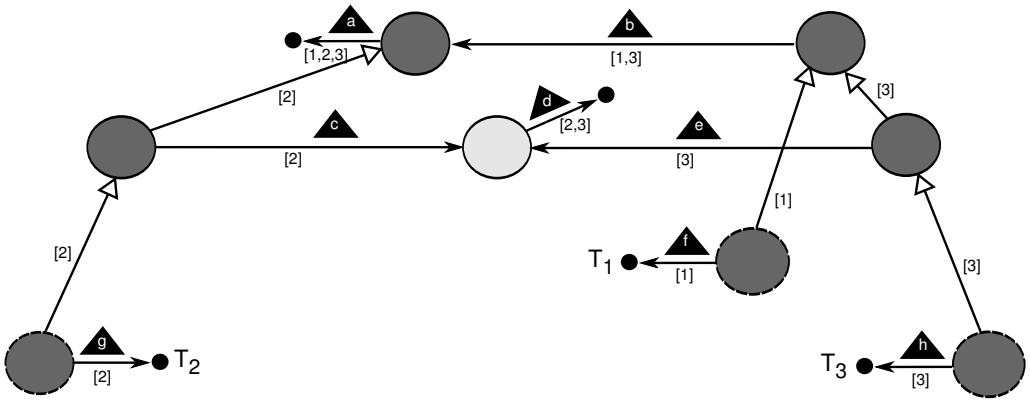


Figure 2.20: Reorganization Step Three

Figure 2.20 shows the resulting graph after merging deltas on the Predecessor Relationship edges. The deltas on the Predecessor Relationship edges are illustrated as black triangles. For each active tenant, there is one leaf node of an inheritance hierarchy. These leaf nodes have exactly one dummy predecessor, because the data on this level is tenant-specific and therefore the predecessors have been collapsed in step two. There is one such dummy node per tenant and we mark this dummy node with the tenant ID. We start from these tenant dummy nodes, follow all edges and mark all edges with the tenant ID. Thereby, the Predecessor Relationship edges are marked by all tenants that require the corresponding delta for deriving their "virtual" relation.

Based on the information from the last step, a new graph can be constructed (see Figure 2.21). There is a node for each tenant and this node is marked with IDs of deltas that are only required by the given tenant. Moreover, there are separate nodes for deltas that are required by more than one tenant. These deltas can be shared between tenants. There are edges between tenants and the shared deltas that they require. In our example there are three tenants and three deltas that can be shared between tenants. If a tenant node is marked with more than one delta IDs, it may

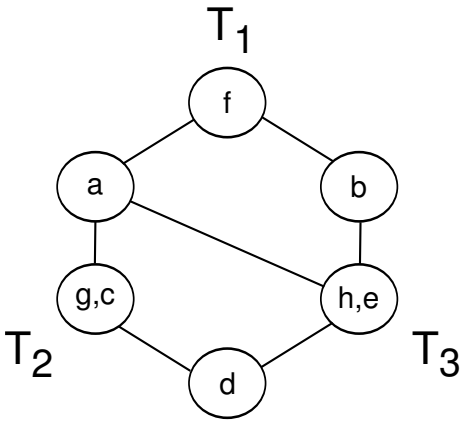


Figure 2.21: Reorganization Step Four

be possible to merge these deltas without replicating shared data. In our example, tenant T_2 has two "private" deltas: g and c. However, these deltas cannot be merged, because they are located on different levels of the inheritance hierarchy and it is not possible to materialize the instance on the upper level without replicating shared data, because c has a predecessor that is shared with tenant T_3. The same is the case for tenant T_3.

In the last step, tenants and shared data have to be assigned to servers. Curino et al. describe a workload-aware approach for database partitioning and replication based on graph-partitioning [33]. A specific graph representation is used that attaches resource requirements to nodes and the cost for separating nodes into different partitions to edges. In our case, this approach can be used to derive an assignment of tenants and shared data to servers. Our graph representation from the previous step can be used for that purpose. The sum of the delta sizes of each node, as well as the processing requirements can be used as node weight and the access frequency for shared data as edge weight. Apart from partitioning, replication may also be applied, as described in [33]. In our case, shared data is mostly read. Therefore, the cost for distributed updates may be negligible and replication of shared data may be acceptable, but may lead to reduced consolidation due to the resulting space overhead.

2.3.6 Seamless Upgrades and Extensions

Today, when several customers are consolidated onto the same hardware and software infrastructure, either all administration and maintenance tasks are performed

by the service provider or the service provider has to coordinate administration and maintenance tasks performed by customers and ISVs. The proposed data model enables seamless upgrades and extensions, when the reorganization of the runtime data model is automated and performed transparently in an on-line fashion. Reorganization of the runtime data model can be performed transparently in an on-line fashion by leveling out the costly reorganization process. A multi-tenant DBMS may perform only light-weight logical schema changes on request and adaptively schedule heavy-weight physical data reorganization operations.

Seamless upgrades and extensions, allow tenants and ISVs to perform the administration and maintenance tasks for their extensions themselves without service provider interaction. The service provider only has to perform administration and maintenance tasks for the base application and extensions provided by the service provider. This approach may lower administration and maintenance efforts of the service provider significantly and can help to reduce total cost of ownership by reducing coordination efforts.

Moreover, the model helps to minimize the costs of tenants and ISVs for performing administration and maintenance tasks on their extensions, by enabling tool-support for these tasks. For example, a tenant wants to upgrade to version $x + 1$ of the base application. Before the upgrade can take place, the tenant has to check if its tenant-specific extensions which work with version x of the base application are compatible with version $x + 1$ of the base application and to identify and resolve possible conflicts. To help the tenant with this task, the differences between the different versions of the base application can be derived from the proposed data model and possible conflicts may be identified automatically by matching the changes in the base application with changes in the tenant-specific extension.

2.4 Conclusions

In this chapter, we introduced multi-tenancy and gave an overview on different implementation options. We pointed out that multi-tenant applications need certain schema flexibility and that current techniques based on application owned schemata have severe drawbacks. We argued that these issues should be solved by integrating multi-tenancy support into the DBMS and proposed an integrated model that allows capturing the evolution and extensibility of a SaaS application explicitly, including data sharing. Our major contribution is the proposed data model which supports branching in the evolution dimension and enables seamless upgrades that may help to reduce administration and maintenance costs significantly. The proposed data model eliminates redundancy by decomposition. Thus, model components have to be overlaid to derive "virtual" relations and partial overlays may be materialized to improve performance. Finally, we formulated the question which partial overlays should be materialized as an optimization problem and presented a simple approach for reorganizing the proposed data model in order to derive an improved runtime representation.

Chapter 3

Cloud Data Management Platforms

Currently, novel cloud data management solutions are emerging, that have interesting properties with regard to scalability and availability. In this section, we analyze the suitability of emerging cloud data management solution for building a SaaS business application, like CRM. The goal is to utilize an infrastructure consisting of a farm of commodity servers with multi-core architecture and large main-memory efficiently. Furthermore, multi-tenancy, automated administration procedures and on-line application upgrades should be supported to minimize administration and maintenance costs while maximizing resource utilization.

3.1 Overview of Cloud Data Management Platforms

Web 2.0 companies require high performance data management solutions in order to serve their interactive web sites to huge numbers of users that also update data apart from reading data. Furthermore, these web-sites have to be highly available and operational costs need to be minimized. Interestingly, today's big Web 2.0 companies like Google, Amazon and Facebook do not use traditional database management systems, but instead have developed custom data management solutions themselves. Google and Amazon have published scientific papers about their systems, called Bigtable and Dynamo. Facebook has open-sourced a system called Apache Cassandra, that is based on architectural concepts of both, Dynamo and Bigtable. These systems have in common, that they are designed to run on a large server farm of commodity hardware.

Although many advances have been made in recent years, traditional general-purpose database management systems cannot scale-out across a large server farm of commodity hardware out-of-the-box. This means that the performance of a traditional DBMS cannot be significantly improved by simply adding a huge number of commodity servers. Instead traditional database clusters are typically scaled-up by adding or replacing components. DBMS vendors typically offer options to their commercial products for improved scalability and availability. As these options typically are not part of the main product, expert knowledge is required for administration and maintenance which results in high operational costs in addition to the license costs. For example, Oracle Real Application Clusters[1] improves availability based on a shared-disk architecture, but is not designed to scale-out across hundreds of servers. MySQL Cluster is based on a shared-nothing architecture and is able to scale-out. But according to MySQL documentation[2], the total maximum number of nodes in a MySQL Cluster is 255. There are commercial offerings that claim to achieve high availability and scalability, like IBM DB2 for mainframes and Oracle Exadata Database Machines, but these offerings typically require non-commodity hardware. Furthermore, there are data warehousing solutions that achieve good scalability for read-mostly workloads, e.g. from Teradata[3] or Greenplum[4].

On the one hand, current cloud data management platforms typically provide fewer features than traditional DBMSs. First, weaker concurrency control mechanisms are used that do not guarantee ACID properties of relational DBMSs. Second, only system-specific APIs and client libraries are provided instead of full-fledged SQL interfaces. On the other hand, cloud data management platforms have to tackle new challenges posed by large server farms of commodity hardware. In large data centers, failures of individual components occur all the time [12]. Thus, cloud data management platforms have to anticipate these failure rates and mask failures of individual components. First, scalability is achieved by data partitioning, often referred to as automatic sharding. Second, data is replicated to improve availability and performance. Moreover, the mentioned techniques may help to improve the availability of database systems. However, actual outages in IT operations are often caused by IT changes that are typically conducted manually by IT personnel. IT

[1] `http://www.oracle.com/us/products/database/options/real-application-clusters` (08/28/2012)
[2] `http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-limitations-limits.html` (08/28/2012)
[3] `http://www.teradata.com` (retrieved 08/28/2012)
[4] `http://www.greenplum.com` (retrieved 08/28/2012)

changes may lead to network outages, unavailability, temporary and even durable loss of customer data. A prominent example is a major outage that occurred in Amazon's US east coast data center on April 21^{st} 2011[5], which led to significant disruptions on customer services. In a conjoint paper with Sebastian Hagen at NOMS 2012 [57], we analyzed this incident and proposed techniques to avoid such incidents by automated detection of conflicting IT changes and violations of safety constraints. Moreover, we compared techniques for efficient generation of IT change plans on large infrastructures and proposed optimizations. This work was published in a conjoint paper with Sebastian Hagen at CNSM 2012 [56].

3.1.1 Google Bigtable

Google Bigtable [27] is a distributed storage system that is used by many Google services, e.g. for web indexing and Google Earth. It provides massive scalability and is designed to scale-out across thousands of commodity servers. The system relies on a single master and many tablet servers that manage data partitions. The master is responsible for assigning and reassigning data partitions to tablet servers. Tablet servers store data in the distributed Google File System [48] which maintains several copies of the data on different servers, depending on the replication level. To achieve high availability, Bigtable relies on a highly-available and persistent distributed lock service called Chubby [21]. Chubby is responsible for keeping track of tablet servers and to ensure that there is at most one active master at any time. High performance is achieved by minimizing client interaction with the master. To achieve this, the clients communicate directly with tablet servers for read and write operations.

Bigtable does not support the full relational data model, but manages structured data. A table in Bigtable is a sparse, distributed, persistent multi-dimensional sorted map that stores uninterpreted strings and is indexed by a row key, column key and timestamp. Row keys are arbitrary strings and data is maintained in lexicographic order by row key. Therefore, reads of short row key ranges are efficient, because data is dynamically partitioned by row key ranges. The Bigtable data model allows to reason about the locality properties of the data. Data locality can be controlled by selecting the row key in such a way that data which is often accessed together is located in contiguous rows. Apart from horizontal data partitioning, vertical data

[5] `http://aws.amazon.com/message/65648` (retrieved 08/28/2012)

partitioning can also be controlled by grouping column keys into column families and column families into locality groups. A column family is compressed together and different locality groups are stored separately. The Bigtable data model is flexible, because once a column family has been created, any column key can be used without announcing it first. Bigtable does not provide global transactions, but every read or write of data under a single row key is guaranteed to be atomic.

There is an open-source implementation of the Bigtable architecture, called Apache HBase. We describe this system in more detail in Section 3.3.1.

3.1.2 Amazon Dynamo

Amazon Dynamo [39] is a highly available key-value store that is used in production for Amazon's world-wide e-commerce platform. It has been designed for applications that neither require relational schema nor general-purpose querying functionality provided by traditional DBMSs. Only simple read/write operations to single data items are supported, no operation spans multiple data items. Data items are uniquely identified by a key and contain binary data.

Dynamo has a completely decentralized architecture based on a structured P2P network similar to Chord [104] and supports continuous growth, as nodes can be added and removed without manual data redistribution. But in contrast to Chord, Dynamo uses a special zero-hop routing protocol. Data partitioning and replication is based on consistent hashing. The output value range of a hash function is mapped onto a fixed circular ring. The location of a data item can be determined by hashing the key of the data item onto the ring. The IDs of storage nodes are also hashed onto the ring and a storage node is responsible for the region between itself and its predecessor node on the ring. To improve availability, data is replicated on multiple storage nodes.

Dynamo gives the application developer control over the trade-offs between availability, consistency and performance. Very high availability may be achieved, but in this case only weak consistency and no isolation is guaranteed. Updates are propagated to all replicas asynchronously. This approach is called eventual consistency, because all updates reach all replicas eventually. Therefore, reads on several replicas may return different, maybe conflicting versions of a data item. Such conflicts are resolved during read operations using versioning information. The application developer may either write own conflict resolution mechanisms or choose simple policies

like last-write-wins. There is an open-source implementation of the Dynamo architecture, called Project Voldemort[6].

3.1.3 Apache Cassandra

Cassandra was open-sourced by Facebook in 2008 and development continues as a top-level project of the Apache Software Foundation. Apache Cassandra is a highly scalable distributed data management system that combines Bigtable's data model with Dynamo's fully distributed architecture. Cassandra is in use at large Web 2.0 sites like Twitter, Netflix and Reddit. The largest known Cassandra cluster consists of 400 servers and manages over 300 TB of data [7].

Cassandra extends the Bigtable data model with super columns that allow grouping of multiple columns. According to the official documentation[8], column families may serve as "indexes" for data stored in a different column family and super columns are useful in this case to represent several matches per indexed value.

Cassandra supports different data distribution options. The selected "partitioner" decides how keys are mapped onto the ring[9]. "RandomPartitioner" is the recommended option, because it distributes data evenly around the ring based on hashing similar to Amazon Dynamo. The "OrderPreservingPartitioner" option does not use hashing and allows to efficiently retrieve a contiguous range of keys, but may lead to an unbalanced data distribution on the ring. Thus, the recommended configuration does not allow to control data locality.

Cassandra allows the application developer to tune consistency levels per query and thereby control the trade-off between availability and consistency. Several consistency levels are supported including strong consistency at the row-level and eventual consistency.

3.1.4 Conclusions

For our application scenario, Bigtable seems more suitable than Dynamo and Cassandra. First, Bigtable gives the application developer implicit control over data locality. The Bigtable design enables this feature by partitioning data by range

[6] `http://project-voldemort.com` (retrieved 08/28/2012)
[7] `http://cassandra.apache.org` (retrieved 08/28/2012)
[8] `http://www.datastax.com/docs/0.8/data_model/supercolumns` (retrieved 08/19/2011)
[9] `http://www.datastax.com/docs/0.8/operations/clustering` (retrieved 08/19/2011)

rather than distributing data according to a hash function. Second, neither Bigtable nor Dynamo provides full transaction support, that is typically required by business applications. But Bigtable at least provides atomic operations at row-level and does not put additional burden on the application developer, as Dynamo does with application-level conflict resolution for eventual consistency. Therefore we use Apache HBase, an open-source implementation of the Bigtable architecture, for our experimental evaluation.

Driven by the cloud computing trend, various specialized data management systems have emerged and have received a lot of attention. Distributed key-value stores like Membase, document stores like CouchDB and graph database systems like Neo4j. A more extensive survey on cloud data management systems can be found in [26] and further information on various emerging systems can be found on the web-site nosql-databases.org[10].

Business applications, like CRM, have different characteristics than the applications behind popular Web 2.0 sites, like social networks. Social networks have a huge number of users and it is very difficult to separate data into more or less independent data partitions, as it is difficult to predict who will interact with whom. In contrast, the number of users of an on-premise CRM business application is much smaller and more predictable. The number of users of a SaaS business application depends on the number and size of its customers. But even if the total number of users of a SaaS business application would be as large as for a social network site, it would be much easier to handle, as data can be partitioned easily by customer or tenant. For multi-tenant SaaS business applications, many independent DBMS instances on a large server farm with automated administration procedures may be sufficient. In contrast, Web 2.0 sites require "web-scale" data management solution which can handle a huge number of users on a common data set of high volume.

[10] `http://nosql-databases.org` (retrieved 08/28/2012)

3.2 Multi-Tenant Database Testbed

We use a multi-tenant database testbed called MTD[11] to analyze the suitability of cloud data management platforms for SaaS business applications. MTD simulates a simple but realistic CRM service.

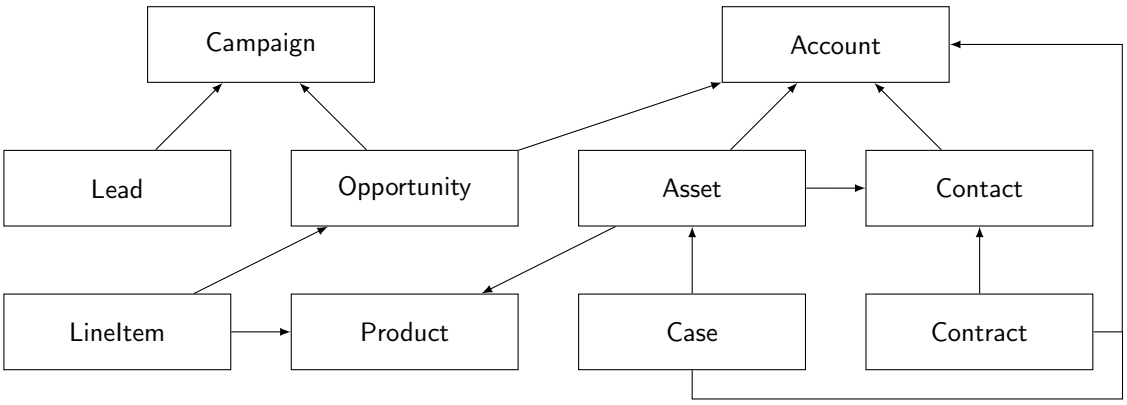


Figure 3.1: MTD Base Schema (adapted from [8])

Figure 3.1 shows the entity types of the base schema and illustrates the relationships between these entity types. The base schema can be extended for individual tenants by extending entity types of the base schema with additional fields of various types. Tenants have different sizes and tenants with more data have more extension fields, ranging from 0 to 100. The characteristics of the dataset are modeled on statistics published by salesforce.com [77].

The workload contains single- and multi-row create, read, and update operations as well as basic reporting tasks that correspond to the following nine request classes:

Select 1: Select all attributes of a single entity as if it was displayed in a detail page in the browser.

Select 50: Select all attributes of 50 entities as if they were displayed in a list in the browser.

Select 1000: Select all attributes of the first 1000 entities of a given entity type as if they were exported through a Web Services interface.

[11] MTD was introduced in [7] and an enhanced version with tenant-specific extensions has been published in [8]

```
SELECT   p.Name, COUNT(c.Case_id) AS cases
FROM     Products p, Assets a, Cases c
WHERE    c.Asset = a.Asset_id
  AND    a.Product = p.Product_id
GROUP BY p.Name
ORDER BY cases DESC
```

Figure 3.2: Reporting Query

Reporting: Run one of five reporting queries that perform aggregation and/or parent-child roll-ups. See Figure 3.2 for an example query that reports the number of cases per product.

Insert 1: Insert one new entity as if it was manually entered into the browser.

Insert 50: Insert 50 new entity instances as if data were synchronized through a Web Services interface.

Insert 1750: Insert 1750 new entity instances as if data were imported through a Web Services interface.

Update 1: Update a single entity as if it was modified in an edit page in the browser.

Update 100: Update 100 entity instances as if data were synchronized through a Web Services interface.

The distribution of requests is controlled using a card deck mechanism similar to the one described in the TPC-C specification [111].

The testbed mimics the behavior of a typical application server by creating a configurable number of connections to the database back-end. These connections are distributed among a set of worker hosts, each of them handling a few connections only, to model various sized, multi-threaded application servers.

The multi-tenant database testbed can be adapted for different database configurations. Each configuration requires a plug-in to the testbed that transforms abstract actions into operations that are specific to and optimized for the target database.

3.3 Experimental Evaluation

In the following experimental evaluation, we analyze the performance of cloud data management platforms in multi-tenancy scenarios where the data volume and workload of any tenant fits on a single server. According to statistics published by salesforce.com [77], this is the case for many small and mid-sized tenants. We use the cloud data management platform Apache HBase and compare it to a commercial disk-oriented DBMS using the MTD-Benchmark. Our results have been published at SIGMOD 2009 [8].

3.3.1 Apache HBase

Apache HBase calls itself "the Hadoop database" [12] and represents a "web-scale" data management platform. According to Borthakur et al. [19] HBase is used at Facebook for the Facebook Messages service. HBase is an open-source project, whose architecture is modeled after Google Bigtable [27], and was originally designed to support the exploration of massive web data sets with clusters of commodity hardware. By now, the project's goal is to support random, real-time read/write access to very large tables with billions of rows and millions of columns. The basic architecture of HBase and Bigtable is very similar. Bigtable leverages the distributed data storage provided by the Google File System and likewise HBase is built on top of the Hadoop Distributed File System [13].

The HBase data model provides a special kind of table that groups columns into column families. Each table consists of a row-key column and one or more column families. One row of a column family can contain zero, one or more label-value pairs. Column families are schema elements that are not owned by the application, because they must be defined in advance using the HBase API. According to best practices, there should not be more than tens of column families in a table and they should rarely be changed while the system is in operation. In contrast, a label-value pair with a new label can be added to a row of a column family without announcing it first. Column families and label-value pairs have generic types, as all values are stored as Strings. Different rows in a table may use the same column family in different ways. Different rows of the same column family can contain different numbers of

[12] `http://hbase.apache.org` (retrieved 08/28/2012)

[13] `http://hadoop.apache.org/hdfs` (retrieved 08/28/2012)

label-value pairs and different labels, hence labels are owned by the application. The column families of a table are stored sparsely and HBase stores the data of a given column family physically close on disk and in memory. Therefore, items in a given column family should have roughly the same read/write characteristics and contain logically related data. Thus, a column family is essentially a Pivot Table as described in Section 2.1.2. Pivot table representation allows storing sparse data in a traditional row-oriented DBMS. A single table with the three columns object identifier, attribute name and attribute value can be used, as described in [3]. Logically, tables are made up of rows whose columns can be accessed by *columnfamilyname:label* and the rows are stored in ascending order by row-key. HBase was designed to scale out across a large farm of servers and uses range-partitioning to achieve that. Physically, tables are broken up into row-key ranges called regions and an entire table is formed by a sequence of contiguous regions. Therefore data locality is controlled by the row-key. Applications can define the key structure and thereby may implicitly control data locality. Rows that have large common key prefixes are likely to be adjacent. The column families of a given region are managed and stored separately. Thus, the rows on each server are physically broken up into their column families.

In the following we describe a generic HBase schema for multi-tenant business applications, like CRM. We propose to use only one table in HBase for storing all data of all tenants and to create one column family per source-table. As each source-table is packed into its own column family, each row has values in only one column family. This causes no overhead in storage volume as the column families of a table are stored sparsely. A source-row is stored as label-value pairs in the corresponding column family. The label is the name of the column and the value is the data in that column of the row. The data of a given source-table is stored in the same column family for all tenants. This works also for source-tables with tenant specific extension columns, because label-value pairs with arbitrary labels can be added to a given row of a column family, as described above. We propose to use a compound key as row-key which is made up of three concatenated parts: the tenant ID, the name of the source-table and the key value of the row in the source-table. In keeping with best practices for HBase, this mapping tries to cluster data together that is likely to be accessed within one query. To minimize communication overhead, it makes sense to store all data that is likely to be accessed in a query on the same node or only few nodes. This is feasible in our scenario, because any query only

accesses data of a single tenant and many queries only access data of one source-table, e.g. search accounts, display or update a given account. Furthermore, the data of any tenant fits on a single node in our scenario.

RowKey	Account	Contact
1Account17	[Name: Duck Inc]	[]
1Account35	[Name: Mouse Inc]	[]
1Contact17	[]	[Name: Dagobert, AID: 17]
1Contact35	[]	[Name: Donald, AID: 17]
1Contact42	[]	[Name: Micky, AID: 35]
2Account17	[Name: Rabbit Inc]	[]
2Contact17	[]	[Name: Bugs, AID: 17, SN: 12345]
2Contact35	[]	[Name: Daffy, AID: 17, SN: 54321]

Figure 3.3: HBase Multi-Tenancy Layout

Figure 3.3 shows how the described HBase multi-tenancy layout works for our multi-tenancy example from Section 2.1.2. The data of both tenants (tenant 1 and tenant 2) is stored in the same table. There is one column family for the source-table Account and one for the source-table Contact. Data is stored as a set of label-value pairs in the corresponding column family.

The single large table is automatically split into regions by HBase. Thereby, a tenant's data may be segmented into two different regions, although the data of a tenant usually fits into one region. This may happen because regions are managed automatically and may be split at any row-key. With the described schema we can only guarantee, that the data of tenants which fit into one region are split at most across two regions. We also tried out a different schema which guarantees, that all data of a tenant, which fits into one region, is contained in a single region. The alternative schema contains one table per tenant and uses a compound key made up of the name of the source-table and the key value of the row in the source-table. But with the alternative schema it was not possible to store the data set for 195 tenants on our test server. We assume, that one region per tenant caused too much overhead as the benchmark requires 195 regions in this case.

The reporting queries in our testbed require join, sort and aggregation operations. HBase currently does not provide the mentioned high-level operators. Therefore, we implemented these operators outside the database in an adaptation layer that runs on the client side. The adaptation layer utilizes operations in the HBase client API such as update single-row, get single-row and multi-row scan with row-filter from the HBase client API for Java. The query operators are executed on the client side and only the underlying HBase operations are executed on the server side. As an

example, consider the reporting query shown in Figure 3.2, which produces a list of all Products with Cases by joining via Assets. To implement this query, our adaptation layer scans through all Cases for the given tenant and, for each one, retrieves the associated Asset and Product. It then groups and sorts the data for all Cases to produce the final result.

We do not use the Hadoop map-reduce framework for query processing, although it is supported by HBase. In our scenario of business applications like CRM, hundreds of small and mid-sized tenants can be managed by a multi-tenant database on a single commodity server. In this setting, it would not be advantageous to spread the data for a single tenant across multiple nodes and to process queries in a distributed fashion, because the overhead for managing the data distribution would probably outweigh benefits resulting from parallelization. Of course, the ideal SaaS database should also scale out to handle large tenants in addition to handling many small tenants efficiently. But Hadoop map-reduce is currently not recommended for small jobs, as the initialization delay can be longer than one second. HBase can be used in combination with the Hadoop map-reduce framework and this combination is very successful in various analytical applications. But the map-reduce framework is not required and HBase itself does not use map-reduce internally. Furthermore, map-reduce on HBase tables could be used for extending our benchmark with business intelligence queries across tenants. Moreover, a performance evaluation of HBase with map-reduce can be found in [25].

We implemented a plug-in to MTD for HBase. For our tests, we used release 0.19.0 of Apache HBase. At the time of our tests, HBase was under heavy development and significant performance improvements were anticipated for the upcoming 0.20 release. In our experiments, HBase was configured in the following way. First, HBase offers only row-at-a-time transactions and we did not add a layer to extend the scope to the levels provided by commercial databases. Second, compression of column families was turned off. Third, neither major nor minor compactions occurred during any of our experiments. Fourth, replication of data in the Hadoop file system was turned off. Fifth, column families were not pinned in main-memory. Sixth, the system was configured such that old attribute values were not maintained.

3.3.2 MS SQL

In our scenario, extension fields contain sparse data. According to [13], sparse data poses a challenge for relational database management systems. Sparse data consists of relations which have many attributes that are null for most tuples. Storing sparse data in normal tables causes a lot of space overhead. HBase is designed to support sparse data, as any label can be used within a column family without announcing it first. Thus, the question is how sparse data can be handled efficiently in a traditional relational DBMS, like MS SQL. MS SQL Server 2008 has a feature called Sparse Columns. Sparse Columns were originally developed to manage data such as parts catalogs where each item has only a few out of thousands of possible attributes. Acharya et al. [1] describe how MS SQL Server implements Spares Columns using a variant of the Interpreted Attribute Storage Format [13], where a value is stored in the row together with an identifier for its column. In combination with the shared table approach and the basic layout described in Section 2.1.2, the Sparse Columns feature can be used to handle NULL values more efficiently. The base tables are shared by all tenants and every extension field of every tenant is added to the corresponding base table as a Sparse Column.

The resulting schema is not owned by the application, as sparse columns must be explicitly defined in the database by a CREATE/ALTER TABLE statement. The shared table approach with basic layout and Sparse Columns requires only a small, fixed number of tables, which gives it a performance advantage over Private Tables; Aulbach et al. show in [7] that having many tables negatively impacts performance. Nevertheless, there is some space overhead for managing Sparse Columns. According to MS SQL Server documentation[14], "Sparse columns require more storage space for nonnull values than the space required for identical data that is not marked SPARSE". The percentage of the data that must be NULL to achieve net space savings depends on the data type. For the int data type, Microsoft states that at least 64 percent of the data must be NULL to achieve net space savings of 40 percent.

For our experimental evaluation, we used an existing plug-in to MTD for MS SQL with Spare Columns. For comparability, we extended the existing plug-in to support the adaptation layer that we use for HBase.

[14] `http://msdn.microsoft.com/en-us/library/cc280604.aspx` (retrieved 08/28/2012)

3.3.3 Experimental Results

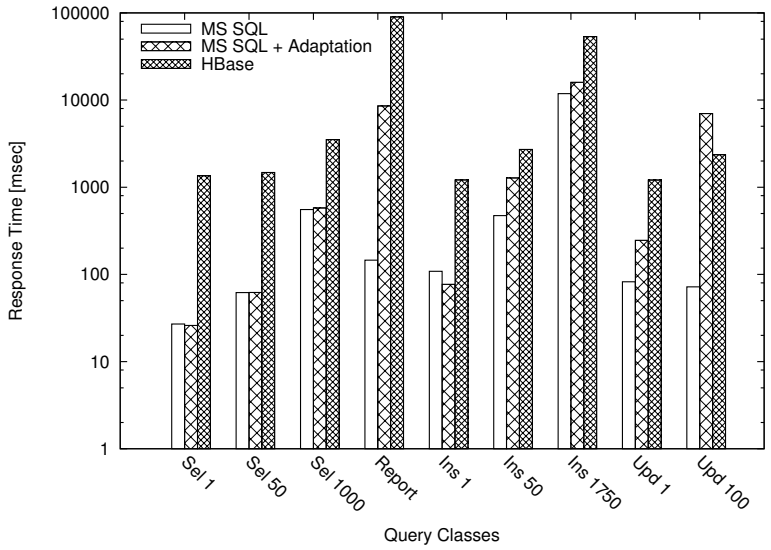


Figure 3.4: HBase Performance

Figure 3.4 shows the results of running MTD on HBase along with two MS SQL Server configurations: one using Sparse Columns and one using our adaptation layer. In the latter case we use our adaptation layer on top of MS SQL Server instead of using corresponding SQL statements. Recall that the adaptation layer performs join, sort and group operations outside the database. To further approximate the HBase mapping, we made the base columns as well as the extension columns sparse in this case. It turns out that this change was not significant. According to independent test runs of MTD, making the base fields sparse has little impact on the performance of SQL Server.

In comparison to the Sparse Columns mapping in MS SQL Server, HBase exhibits a decrease in performance that ranges from one to two orders of magnitude depending on the operation. One reason for this decrease is the reduced expressive power of the HBase APIs, which results in the need for the adaptation layer. This effect is particularly severe for reports and updates, where SQL Server with adaptation also shows a significant decrease in performance. These results are consistent with observations by Franklin et al. [45], which show that shipping queries to the server rather than shipping data to the client can have significant performance advantages, especially if locality of data access is poor at clients. This applies for our scenario, because we do not cache data at the client side, as we assume high update

rates. The performance decrease for updates is primarily due to the fact that the adaptation layer submits changes one at a time rather than in bulk. HBase has a bulk update operation, however it appears that, in the version we used, changes are not actually submitted in bulk unless automatic flushing to disk is turned off. Furthermore, HBase accesses disks over the network via the Hadoop File System. In contrast, shared-nothing architectures typically put disks on the local SCSI bus while shared-disk architectures use fast SANs.

The conclusion we draw from these experiments, is that commercial DBMSs achieve better performance than cloud data management platforms like Apache HBase when the data volume of any tenant fits on a single server as in our multi-tenancy scenario.

3.4 Service Models for Cloud Data Management

Today many businesses already use cloud computing for business applications like CRM. Business applications typically rely on relational database management systems for managing data. Therefore, it seems worthwhile to assess how database systems can be provided efficiently as a service in the cloud. We compare different service models for cloud databases and conclude that cloud database services represent a category of its own: Database-as-a-Service. Moreover, we discuss migration towards Database-as-a-Service. [15]

3.4.1 Database-as-a-Service

Database systems can be provided as a service over the Internet. The simplest option is to deploy traditional relational database systems (RDBMS) on virtual machines of IaaS providers like Amazon EC2, as customers are allowed to deploy almost any software on virtual machines. Figure 3.5 (a) shows a DBMS and an Application instance App running inside a virtual machine VM1. VM1 runs together with other virtual machines, e.g. VM2, on the same virtualized infrastructure of a IaaS provider. But deploying traditional RDBMS on virtual machines is only a first step, because the customers of IaaS offerings still have to administer and maintain all software that runs inside their virtual machines themselves. The remaining

[15] Parts of this work have been published in our Datenbank-Spektrum journal article [93].

database management activities incur high operational costs, because skilled personnel is required for installation, configuration, administration, performance tuning and maintenance of database systems. Regular maintenance activities include tedious tasks like application of patches and creation of backups. Moreover, it is a big challenge to elastically scale traditional RDBMS across several virtual machines, as these systems were not designed for this feature.

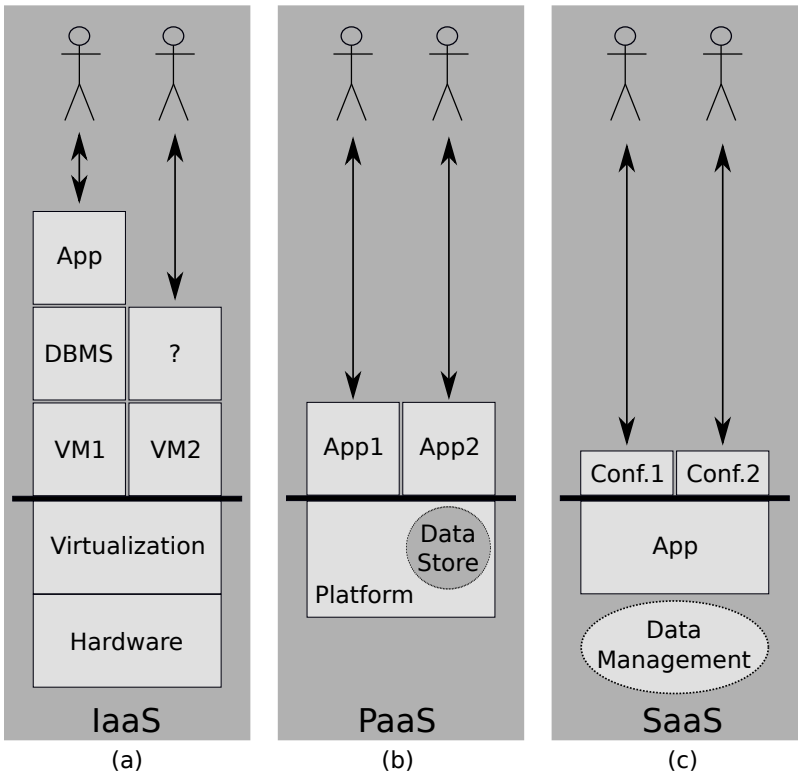


Figure 3.5: Service Models for Cloud Databases

In order to reduce operational costs further, database management tasks should be out-sourced to the service provider who may automate administration and maintenance procedures to achieve economies of scale. Such a cloud database service can be provided on-demand and may be priced according to the pay-per-use principle without long-term contract or up-front payment. Amazon and Microsoft already provide cloud database services: Amazon Relational Database Service (Amazon RDS)[16] and Microsoft SQL Azure. But currently customers still may have to take care of some database administration tasks. Amazon RDS recently added support for Oracle, one of the most widely used commercial RDBMS. A research group at

[16] `http://aws.amazon.com/rds` (retrieved 08/28/2012)

ETH Zürich has shown, how database systems can be built on top of commercial cloud services [20].

How can we classify such a cloud database service? Cloud database services definitely do not fall under the IaaS category, as databases reside at a higher level of abstraction than virtual machines. But it is not clear if cloud database services should be classified as PaaS or SaaS. On the one hand, PaaS typically integrates the data store tightly with the provided platform (see Figure 3.5 b) and requires applications to be built with specific tools in order to utilize a common software platform, e.g. force.com. In contrast, cloud database services provide a very generic interface and put almost no restrictions on how the application is implemented. Even on-premise applications may use database services in the cloud. On the other hand, SaaS applications are typically accessed by end-users using a thin client, like a web browser, over the Internet. In contrast, database services typically are not accessed by end-users directly, but by application software. SaaS applications use a data management solution internally, but typically the stored data can only be accessed via the provided application (see Figure 3.5 c). In our opinion, cloud database services represent a category of its own: Database-as-a-Service (DbaaS). There is some related early work by Hacigümüs et al. [55] and recent work like the "Relational Cloud" project by Curino et al. [35].

DbaaS is similar to PaaS, but provides more flexibility with regard to application development. Apart from the core data management functionality, DbaaS should provide user authentication and authorization features to control data access in a fine-granular manner. Recently, salesforce.com released a cloud database service, called database.com. This service basically repackages the data management layer that is used by the SaaS CRM application salesforce.com and the PaaS force.com, but apart from authentication and authorization features the new service provides a REpresentational State Transfer (REST) API that makes application development more flexible. Currently, many new DbaaS providers emerge, like FathomDB and MongoHQ (see also the DbaaS Product Directory[17]).

[17] `http://dbaas.wordpress.com/database-as-a-service-dbaas-product-directory` (retrieved 08/28/2012)

3.4.2 Migration towards Database-as-a-Service

Ease of migration plays an important role in the adoption of new paradigms like DbaaS. As part of his bachelor thesis, Sebastian Wöhrl analyzed how to migrate an existing on-premise data center, that mainly hosts dedicated servers leased by internal clients (e.g. other departments), from the traditional managed-hosting paradigm towards the novel cloud computing and DbaaS paradigms.[18] Due to security concerns and legal requirements, existing DbaaS offerings from the public cloud like database.com may not be an option. Instead, the existing on-premise data center may be transformed into an internal DbaaS provider for such private cloud scenarios. The most important goal is to improve flexibility. By automating administration tasks that are currently done manually, e.g. apply patches, create new database instances, manage user accounts and permissions across several systems, the server and database provisioning process can be made faster and less error-prone. Furthermore, automation may help to reduce frequency and duration of maintenance windows. A big challenge is that on-premise data centers today typically host many legacy applications. This often includes many proprietary business applications with three-tier architecture that require specific commercial database systems and maybe even specific versions of them. But there may be several — if not many — applications that require the same commercial database system (maybe they can even use the same version). In order to migrate these applications towards DbaaS, application-level changes may be necessary and this requires huge effort. Therefore, Sebastian Wöhrl proposes a soft incremental migration with several intermediate levels. The first step is to automate administration and management of existing database systems, similar to Amazon RDS. The second step is to make authentication and authorization mechanisms more independent from the used database systems. The third step is to develop an internal DbaaS abstraction layer that wraps a commercial or open-source database system, provides abstract interfaces and prohibits use of vendor-specific extensions. On the one hand, the goal is to support a large subset of features needed by many applications (SQL features, support for stored procedures, etc.) to reduce the amount of application-level changes. On the other hand, the DbaaS interfaces should be kept as simple as possible for

[18] Sebastian Wöhrl did his bachelor thesis "Automated Server-Provisioning at Siemens CIT: Design and Implementation" at Siemens Corporate Information Technology (Siemens CIT) and I (Michael Seibold) was his advisor.

manageability. The last step is to use the DbaaS solution for new applications and to migrate more and more legacy applications toward the DbaaS.

3.5 Conclusions

In this chapter we gave an overview on emerging cloud data management solutions and analyzed their suitability for SaaS business applications, like CRM. Furthermore, we analyzed how multi-tenancy can be realized with such systems and proposed a multi-tenant schema mapping approach for one of these systems, namely Apache HBase. Moreover, we discussed different service models for cloud data management. This is relevant for SaaS business application, as a SaaS provider may use a DbaaS data management solution internally and one of the presented cloud data management solutions may form the basis of such a DbaaS offering. None of the presented systems offers multi-tenancy support out-of-the box. But cloud data management platforms, like Apache HBase, offer a certain schema flexibility that may make it easier to implement multi-tenancy with support for on-line application upgrades. For our application scenario of multi-tenant business applications like CRM, we assume that the processing and storage requirements of any tenant can be fulfilled by a single server. Therefore, we compared the single-server performance of an open-source "web-scale" data management solution with a commercial DBMS. We conclude that many independent DBMS instances on a large server farm with automated administration procedures may be sufficient for multi-tenant SaaS business applications and our experimental results suggest that this approach may enable more efficient resource utilization. We do not consider "web-scale" data management platforms further and instead focus on optimization techniques for consolidating several small and mid-sized tenants.

Chapter 4

Mixed Workloads

Operational Business Intelligence systems have to process analytical queries (OLAP) and business transactions (OLTP) at the same time on the same data. Managing the resulting mixed workloads (OLTP and OLAP on the same data) poses a big challenge for current disk-based DBMSs [70]. For mixed workloads, it is very difficult to achieve high performance, serializability and data freshness at the same time, as OLTP and OLAP workloads have very different characteristics. Main-memory database architectures could be the right means to tackle the challenge that mixed workloads pose for disk-based DBMSs. Requests can be processed at much smaller time scales and there is less variation in execution times, when data can be accessed without disk I/O. Our goal is to process mixed workloads of SaaS business applications, like CRM, with Operational Business Intelligence features, e.g. analytic dashboards, according to strict SLAs with stringent SLOs and enable service providers to commit to significant penalties. With stricter SLAs it would be easier to compare different cloud offerings with on-premise solutions and thus cloud computing could become more attractive for potential customers.

In the following, we analyze the characteristics of mixed workloads and outline the challenges posed by mixed workloads in our application scenario. We give an overview on techniques for handling mixed workloads and present a special purpose main-memory DBMS prototype, called MobiDB, for handling the mixed workload of our application scenario that allows for stringent service level objectives and significant penalties.[1]

[1]Parts of this work have been published at CLOUD 2011 [95] and have been accepted for publication in IT Professional [92].

4.1 Characteristics

In the context of business applications, the mixed workload consists of an OLTP component and an OLAP component.

The OLTP workload is generated by transaction-oriented applications that are vital to day-to-day business operations. These applications are typically used by a large number of concurrent users and individual business transactions typically have relatively short execution times. Business transactions read, insert, update and delete data in the database and form part of many parallel and independent OLTP sessions, as they are triggered by many different users. A business transaction typically corresponds to a single database transaction in a straight forward manner. Although, business transactions are sometimes split into several database transactions to improve performance.

The OLAP workload is generated by business intelligence applications that are used for analyzing large amounts of business data to support strategic decision making, e.g. computing sales revenue of a company by products across regions and time. These applications are typically used only by a small number of concurrent users and individual business queries typically have relatively long execution times due to the complexity of the queries and the volume of analyzed data. A business query involves a sequence of one or more read-only database queries which need to be evaluated on the same consistent database state and therefore form a single read-only database transaction.

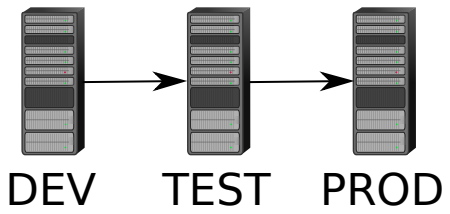


Figure 4.1: Three System Landscape

In general, business transactions may be formulated ad-hoc, but we assume that business transactions are canned and therefore change in the production system only due to application upgrades. This is reasonable for today's business applications with web-based user interfaces, as the SQL database interface is typically only used by the application layer and a three-system landscape is used for developing, testing

and deploying business applications (see Figure 4.1).

Furthermore, we assume that business transactions are interaction-free, which means that interaction with the user may take place only at the beginning and at the end of a business transaction, but not during a business transaction. User-interaction within business transactions may cause severe synchronization overhead as a business transaction typically corresponds to a single database transaction. Thus, for DBMSs with lock-based concurrency control, the duration for which locks are held depends on the responsiveness of the user. High-throughput OLTP systems typically prohibit user-interaction within business transactions to avoid "user stalls", as discussed in [105]. Moreover, we assume that business transactions are deterministic, as most business transactions can be made deterministic. Thomson et al. propose to use a preprocessor, that performs any necessary non-deterministic work in advance and to pass on the results as transaction arguments [109].

In general, business queries may be ad-hoc or canned and may involve user-interaction within the business query. We assume that most business queries are periodic business queries which are canned and interaction-free. There may be few ad-hoc business queries which only occur once in a while. This is reasonable for our Operational Business Intelligence scenario, as some analytical queries are processed on the operational database, but very complex and long-running OLAP queries are still processed on separate data warehouse systems. Therefore, we assume that business queries typically require longer execution times than business transactions, but not as long as complex OLAP queries typically found in data warehouses. As shown by Funke et al., 16 out of 22 TPC-H-like queries can be executed within less than 500 milliseconds with a modern main-memory DBMS running on commodity hardware [47]. We conclude that periodic and short-running queries should be processed on an operational business intelligence system and the longer-running queries on a separate data warehouse. Thereby, the update cycles of data warehouses can be kept large enough for processing long-running queries efficiently.

4.2 Challenges

In a mixed workload that contains OLTP and OLAP components, many short OLTP transactions, which make changes to the database, may conflict with longer-running read-only OLAP queries. Conflicts are highly probable if business transactions and business queries are performed concurrently on the same data. This incurs heavy synchronization overhead which negatively affects performance and results in low overall resource utilization.

With a lock-based concurrency control mechanism for example, a longer-running business query may delay concurrent short-running business transactions due to lock conflicts. This may cause reduced throughput and increased response times for concurrent business transactions. If throughput and response time guarantees for the business transactions are still met depends on the execution time of the business query, the execution times of the business transactions and the expected response times of the business transactions. One business query can delay a concurrent business transaction at most by the execution time of the business query. If the expected response time of the business transaction minus the execution time of the business query is larger than the execution time of the business transaction, the response time guarantee for the given business transaction may still be met. But throughput guarantees may require that many concurrent business transactions meet their response time guarantees while the given business query is performed. In order to fulfill response time and throughput guarantees of the business transactions, the longer-running business query may have to be aborted. Depending on the arrival rate of business transaction requests, this may lead to starvation and cause violation of response time and throughput guarantees for business queries.

4.3 Related Work

There are commercial DBMSs, like TimesTen [108], which achieve high OLTP throughput rates by keeping most or all data in main-memory. As shown by Kallman et al., typical business transactions, like order entry or payment, can be executed within less than 100 microseconds with a modern main-memory DBMS running on commodity hardware by removing traditional DBMS features, like buffer management, locking and latching [61, 105]. Their research prototype H-store [61] and

its commercial successor VoltDB achieve extremely high throughput rates by minimizing synchronization overhead. Transactions are processed by multiple single-threaded engines and data is partitioned across a database cluster. The RAMcloud [85] development at Stanford shows the feasibility of a main-memory-only approach. For OLAP, there are highly efficient query processors based on column store technology, as pioneered by MonetDB. Instead of the traditional tuple-at-a time pipelined execution paradigm, MonetDB uses a column-at-a-time paradigm and aims at utilizing large main-memory and multi-core architectures more efficiently [76]. Recently, a new system called HyPer has been proposed which can handle mixed workloads consisting of OLTP and OLAP at extremely high throughput rates, based on a low-overhead mechanism for creating differential snapshots which relies on hardware supported operating system mechanisms and works at the granularity of virtual memory pages [64].

Our approach of handling OLTP is similar to H-Store, but our approach is not limited to OLTP. Like HyPer, our approach is optimized for mixed workloads. When snapshots are created periodically, as described in [64], queries are performed on the last snapshot at the time a query session started. Thus, data freshness is not guaranteed, as changes may have been committed between the time the last snapshot was created and the time the query session started. This lack of data freshness depends on how often snapshots are created. For HyPer the lack of data freshness is limited, as snapshots can be created within few milliseconds. Nevertheless, many snapshots may be required to guarantee data freshness, one per query session in the worst case. Instead, we propose to queue query requests and to delay the start of a query session until the next periodic snapshot is created. Our special-purpose system always guarantees data freshness and tries to avoid snapshots whenever possible, because of the incurred space overhead. The proposed system is not general-purpose, because it is optimized based on the characteristics of our application scenario (see Section 4.1). But there is another major difference to the mentioned general-purpose systems that process queries and transactions in a best-effort manner. For our special-purpose system, stringent response time and throughput guarantees can be given based on our proposed queuing approach, an analytical model and resource reservation.

Tenant placement is an area of active research in the Cloud Computing and service computing community. Zhang et al. formulate the Online Tenant Placement

Problem, "given a fixed number of nodes, how to optimally place on-boarding tenants to maximize the total supported number of tenants without violating their SLA requirements", and show that it is NP-hard [119]. We discussed a similar problem in Section 2.3.5 and formulated it as a graph-partitioning problem. In Section 4.5.3, we present an analytical model that allows analyzing if the mixed workload of one or several tenants can be processed on a given infrastructure according to our proposed queuing approach.

4.4 Techniques for Handling Mixed Workloads

In the following we give an overview on techniques which help to process mixed workloads more efficiently [2]. We look at the space and processing overhead of these techniques and analyze how overheads are distributed across the different components of the mixed workloads.

4.4.1 Reduced Isolation Levels

Commercial DBMSs offer a number of reduced isolation levels, e.g. read-committed, which may require less synchronization within the DBMS and thus improve performance for mixed workloads from the perspective of the DBMS. But anomalies may occur which have to be precluded or handled explicitly by the application layer which is error-prone, makes the application layer more complex and thus increases maintenance costs. The goal stated in the introduction is to maximize resource utilization and to minimize administration and maintenance costs at the same time. Reduced isolation levels may reduce synchronization overhead on the database layer and therefore enable better resource utilization, but in order to minimize administration and maintenance costs, potential interference from multi-user operation has to be precluded by the DBMS with the highest isolation level, serializability, in order to shield the application layer from the associated complexity.

Moreover, certain techniques for implementing reduced isolation levels may cause additional space overhead. Even when reduced isolation levels are used for processing mixed workloads, the isolation level should be high enough to ensure that certain anomalies do not occur. For example, business queries should not see uncommitted changes of concurrent business transactions. DBMSs employ special techniques to preclude this anomaly with isolation level read committed, like shadow paging, which requires storing more than one copy of certain data pages [42] and thus causes additional space overhead. In our scenario, space overhead has to be minimized, as SaaS providers typically employ multi-tenancy techniques for reducing costs by consolidating several tenants onto the same infrastructure. Additional space overhead may limit or even prohibit consolidation and thus may impact resource utilization negatively.

[2] Most of the mentioned techniques are also described in [63] and chapter 20.2 of [62]. Beyond that, we focus on the applicability of these techniques for our SaaS scenario.

Recently, techniques have been proposed to make snapshot isolation, a reduced isolation level defined in [14], serializable by preventing anomalies at runtime [22]. But on the one hand transactions may be aborted even due to potential anomalies, which may cause starvation when anomalies are highly probable. On the other hand, snapshot isolation is typically implemented with multiversion concurrency control, which stores several versions of the data and therefore causes a certain space overhead [42]. Therefore, this approach does not seem feasible in our mixed workload scenario.

In our scenario, the highest isolation level, serializability, is required while synchronization and space overhead has to be minimized. There are two major kinds of serializability: view serializability and conflict serializability [15]. We focus on conflict serializability, as most mayor DBMSs employ lock-based concurrency control mechanisms which are based on the notion of conflict serializability. Thus, in the following, serializable really means conflict serializable.

4.4.2 Separation by Copying Data

The components of a mixed workload can be separated by processing them on their own snapshot of the data. Thereby, synchronization overhead can be reduced significantly at the cost of additional space overhead. In our scenario, business queries can be processed on a consistent copy of the data while business transactions are processed on the current data. This approach does not reduce the isolation level, as business queries are read-only[3]. But many snapshots may be required to ensure serializability and data freshness. There are different approaches for creating a consistent copy of the data which differ in the time needed to create the snapshot and the resulting performance and space overhead.

Versioning

Insert-only database systems never update tuples in-place. Instead, several versions of a tuple are stored. The open-source DBMS PostgreSQL introduced this approach by treating the log as normal data managed by the DBMS and provides support for queries on historic data that implicitly define a snapshot [106]. The advantage of

[3]Standard DBMS interfaces, like JDBC, allow to mark database transactions as read-only. This additional information can be used by the DBMS to optimize transaction processing.

this approach is that business transactions and business queries can be performed concurrently without synchronization overhead, as they work on different versions of the data. As all versions are kept, the required data for any snapshot is available. The obvious disadvantage of this approach is that the database grows quickly as all tuple versions are kept. The data volume may become several times larger than the data volume of only the latest versions. To reduce this space overhead, a "vacuum" operation has to be performed regularly to move old versions to tertiary storage, as described in [106]. This technique either adds processing overhead for determining the latest versions and the versions corresponding to a given snapshot. Alternatively, directory or index structures may be used to reduce processing overhead at the expense of additional space overhead.

Complete Snapshots

Some of the space overhead introduced by versioning can be avoided by keeping only those versions of tuples which correspond to snapshots that are still required. The complete snapshot approach does not share common data between snapshots and is similar to a two system approach with an operational database, a data warehouse and an ETL process. Managing both copies in a single system may enable optimizations to reduce the overhead caused by ETL. For example, a snapshot may be updated by applying business transaction requests in bulk, instead of using change data capture to retrieve changes from an operational database. With a single snapshot, the update frequency limits the maximum execution time of a business query. But the update frequency depends on data freshness requirements. Therefore, more than one snapshot may be required which could cause significant space overhead.

An advantage of complete snapshots in comparison to other snapshotting techniques is that different schemata and data representations may be used for current data and snapshots. For example, normalized tables and row representation can be used for processing business transactions while star-schema and column representation may be used for processing business queries. Thereby, mixed workloads may be processed more efficiently, at the expense of higher overheads for snapshot creation. Then again, using the same schema and the same representation may make request processing less efficient, but enables sophisticated techniques for updating snapshots quickly. Cao et al. propose such techniques in the context of checkpoint recovery [24].

Space overhead depends on the number of snapshots and incurs at least a factor of two. In our SaaS scenario, a space overhead of a factor two or more may limit consolidation severely when memory is the limiting factor. Therefore, the complete snapshot approach does not seem feasible for our scenario.

Computed Snapshots

The space overhead of complete snapshots can be eliminated by keeping only the latest version of tuples. The versions required for older snapshots can be computed on demand by undoing operations according to the undo log. This technique causes no overhead for business transactions, but may add significant processing overhead for business queries. This processing overhead can be reduced by caching complete or partial snapshots which in turn incurs space overhead. This approach does not seem feasible for our Operational Business Intelligence scenario, as business queries have to be processed with low response times at high throughput rates.

Differential Snapshots

Depending on the update characteristics of the application, the current data and the complete snapshots contain lots of redundant data. This redundancy can be reduced by sharing data between current data and snapshots. We refer to this mechanism as differential snapshots. The amount of redundancy that can be eliminated depends on the granularity of the differential snapshot mechanism and the update characteristics of the application. In the following, we focus on row level and page level granularity. Row level granularity seems very suitable for business transactions which typically process data row-wise. A common way to realize a differential snapshot with row level granularity is a delta mechanism that stores changes in a delta structure instead of performing the changes on the current data. This causes a certain overhead for business transactions, as they have to consider the delta for read operations. Furthermore, the delta grows over time when changes are made while the snapshot is active. Thus the snapshot should be removed as soon as it is not needed anymore. But to remove such a snapshot, changes stored in the delta have to be merged into the current data. There is recent work by Krüger et al. [69] on how to optimize this merge process for modern servers with multi-core architectures. Furthermore, there is a novel way to realize differential snapshots with page level granularity that is based on copy-on-write mechanisms of the operating system. This approach is used

by HyPer and is described by Kemper and Neumann in [64]. Copy-on-write snapshot mechanisms are an area of active research. Sowell et al. [102] recently proposed a copy-on-write snapshot mechanism for B-trees and Sidlauskas et al. [98] observed no impact on read performance when comparing page level differential snapshots — created similar to the HyPer approach — with complete snapshots. Moreover, a general disadvantage of differential snapshots in comparison to complete snapshots is that the same schema and the same representation have to be used for current data and snapshot. In contrast, complete snapshots may employ different schemata, e.g. normalized tables and star schema, and different representations, e.g. row store and column store.

4.4.3 Separation by Time

For mixed workloads, synchronization overhead can be reduced by separating the individual components of the mixed workload and processing them more independently from each other. For business applications, the OLTP and OLAP components can be separated by controlling when business transactions and business queries are executed.

Cyclic Scan Processing

There is a big difference in execution times of OLTP and OLAP requests, because the database is accessed in different manners. Business transactions typically are supported by indexes, mostly use point-wise accesses and therefore have short execution times. Business queries typically have to read lot's of data, require mostly scan accesses and therefore have longer execution times. Instead of point-wise accesses, business transactions could also use scan accesses, but this probably would result in longer execution times. It depends on the application scenario if the required response times can still be met.

There are special-purpose systems that read the entire database in repeating circular scans, called cycles, and let multiple operations share the scan cursor. The goal of these cooperative scans is to improve cache locality and to cope with limited main-memory bandwidth. During a single cycle several operations corresponding to different business queries and business transactions may be performed for each scanned tuple. Thereby, the available processing power may be utilized better, while

the next tuple is prefetched through the memory hierarchy. This approach can help to improve resource utilization if memory bandwidth is the bottleneck. During a cycle, each tuple is loaded once for a given set of operations. First write operations are performed in arrival order. Then read operations are performed on the given tuple. Thereby each cycle corresponds to a consistent snapshot of the database. Special-purpose systems like Crescando apply and extend this approach to achieve predictable performance for unpredictable workloads [113, 50]. This approach can be categorized as time-based, because business queries and business transactions have to be processed according to the mentioned cycles. The question arises if this approach can be applied to business applications like CRM. On the one hand, this approach slows down business transactions in favor of business query throughput. On the other hand, consistent snapshots are only available during a given cycle. This means that a business transaction has to complete all its read operations within one cycle in order to work on a consistent snapshot and can only read tuples in the order of the circular scan. It may be necessary to process and maybe even cache tuples that might be needed later on in the cycle, which may lead to additional space overhead. Due to the mentioned restrictions and the impact on execution times of business transactions, this approach does not seem feasible for our scenario.

Admission Control and Priority-based Scheduling

In the workload management area a lot of research has been done to optimize the workload mix for handling transactions and queries in the same DBMS. Requests are mapped to different service classes. An SLA defines objectives for each service class and penalties if these objectives are not fulfilled, as described by Krompass et al. in [67]. Workload management tries to meet all objectives as long as there are sufficient resources. The only way to control how resources are distributed among the components of a mixed workload is to specify penalty costs of the corresponding service classes accordingly. When there are too many requests, admission control and priority-based scheduling are applied to minimize the penalty costs for requests that miss their objectives. Workload management solutions typically treat the DBMS as a "gray box". Monitoring techniques are used to build up a model about the resource requirements of transactions and queries. This model is used by admission control and priority-based scheduling techniques. Workload management has some space and processing overhead, which mainly depends on the number of

queued requests and the monitoring overhead, but not on the data volume of the database. The workload manager controls when queries and transactions are submitted to the DBMS and an execution controller may abort queries which consume too many resources. But workload management solutions typically do not change how the DBMS works internally. For a given DBMS it may be the case that a given mixed workload cannot be performed without missing its objectives due to synchronization overhead within the DBMS. Furthermore, current workload management solutions are typically added on-top of commercial general-purpose DBMSs which are typically disk-based and execute requests in parallel in order to mask delays caused by disk I/O. In such systems, stringent guarantees cannot be given, because the parallel execution of requests makes it very difficult to derive accurate execution time estimates due to unpredictable interferences between the different requests and resource contention, especially for mixed workloads[4]. Therefore, traditional workload management is not sufficient for our application scenario that requires strict SLAs with stringent service level objectives and significant penalties.

Request Queuing and Resource Reservation

Instead of doing workload management in an on-line best effort manner, we propose a soft real-time approach based on techniques known from the research area of workload management. For a known workload, the resource requirements can be assessed off-line and by reserving the required resources, known workloads can be processed according to soft real-time guarantees. Depending on the objectives of the service provider, sufficient resources can be reserved to meet all objectives or to keep penalty costs below a certain limit. This approach would enable service providers to include stricter SLOs regarding response time and throughput in their SLAs and thereby could make cloud computing more attractive for potential customers.

This approach is feasible, as emerging main-memory DBMSs allow to predict execution times quite accurately, as shown by Schaffner et al. in [90]. In contrast to traditional disk-based DBMSs, main-memory DBMSs do not have to process requests concurrently in order to mask delays caused by disk I/O. Instead transac-

[4]Researchers lead by Ashraf Aboulnaga have observed that interactions between queries running concurrently in a query mix can have significant impact on performance. Tozer et al. [110] propose to improve admission control decisions based on a model of expected query execution times that accounts for the mix of queries being executed.

tions can be processed in a single-threaded fashion and one single-threaded engine per CPU core and/or cluster node can be used, as described by researchers lead by Michael Stonebraker [61, 105]. We extend this approach for mixed workloads. Business transactions and business queries should not be processed in parallel on different cores of a modern multi-core server, in order to avoid the resulting synchronization overhead. Instead, requests should be queued and the queued business transaction requests should be processed within one time-slot and the queued business query request within a different time-slot. The required time slot duration can be determined based on throughput and response time requirements and execution time estimates. When business transactions and business queries are performed in separate time slots, data is changed only in one of the two time slots, as business queries are read-only. Therefore, business queries can be performed directly on the data as if it were a snapshot. We call that a time-based snapshot. During the business query time slot only read-only queries are processed, therefore queries can be processed in parallel without synchronization and multi-query optimization techniques may be applied. There is recent related work by Giannikis et al. [49] on multi-query optimization based on batched query execution that aims at response time guarantees in high load situations.

The major advantage of time-based snapshots is that both workloads are performed on the same data and thus no data has to be copied. Additional space is required for queuing requests, but requests are typically small - a couple of numbers and strings - and the number of requests only depends on the guaranteed request rate and not on the data volume of the database. There is some synchronization overhead at the beginning and at the end of each time slot, but this can be implemented efficiently with hardware supported barrier synchronization methods. The major restriction of this approach is that it only works if the workload is known in advance. Furthermore, a given mixed workload can only be processed according to this approach on a given infrastructure, if the execution times of the expected business query requests are short enough for keeping the OLAP time slot short enough, such that concurrent business transaction requests are not delayed too much and still meet their response time goals. This approach can be applied for those components of our mixed workload that are known in advance: business transactions and periodic business queries. Additional measures have to be taken for handling ad-hoc business queries that are not known in advance.

4.4.4 Conclusions

Reduced isolation levels do not meet our requirement of minimizing administration and maintenance costs. Thus, the highest isolation level - serializability - is required while synchronization and space overhead has to be minimized. Separation by time seems most promising for minimizing space overhead for known workloads and separation by copying data may be required for handling ad-hoc queries. In the next section a combination of the *Request Queuing and Resource Reservation* approach and the *Differential Snapshots* approach is proposed to achieve this goal. The former approach is applied for those components of our mixed workload that are known in advance: business transactions and periodic business queries. The latter approach is applied for handling ad-hoc business queries that are not known in advance.

4.5 MobiDB: Special-purpose Main-Memory DBMS

Cloud Computing and SaaS foster the development of data management systems that are optimized for specific application scenarios. Once a SaaS provider has identified an application scenario whose potential customer base is large enough, the entire software and hardware stack should be optimized based on the characteristics of the specific application scenario in order to gain competitive advantage relative to on-premise solutions and more general-purpose cloud offerings. MobiDB is a special-purpose main-memory DBMS prototype which guarantees serializability and can handle the mixed workload of our Operational Business Intelligence scenario with low response times at high throughput rates while minimizing space overhead and adhering to maximal response time and minimal throughput guarantees. MobiDB combines the *Request Queuing and Resource Reservation* approach and the *Differential Snapshots* approach described in the preceding section (Section 4.4).

4.5.1 SaaS architecture

In this section, the traditional architecture of business applications is presented and its suitability for our SaaS scenario is discussed. Based on this discussion an architecture optimized for SaaS applications is proposed.

Three-Tier SaaS-Architecture

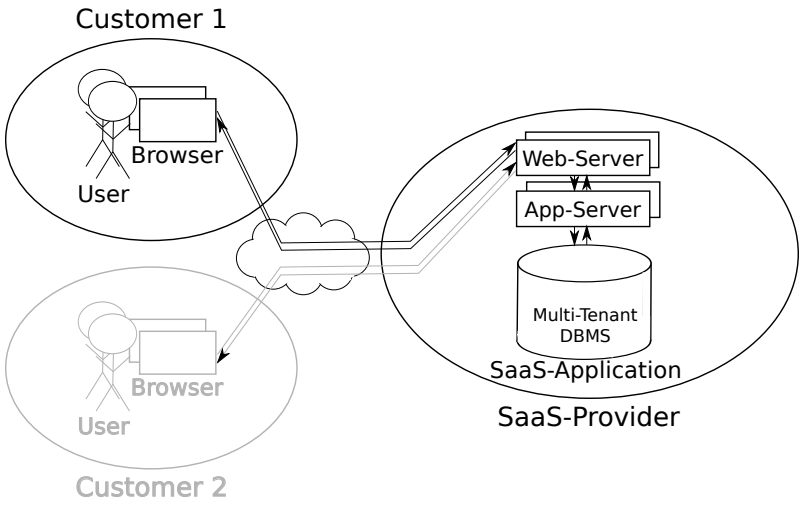


Figure 4.2: Three-Tier SaaS-Architecture

Traditionally, business applications have a three-tier architecture. Business transactions and business queries represent high-level operations which traditionally span all three layers: presentation layer, application layer and database layer. Figure 4.2 shows how this traditional architecture can be applied for SaaS business applications. The SaaS provider operates all three layers and may use a multi-tenant DBMS for improving consolidation. The different layers may be deployed into separate virtual machines and many instances may be run on the server farm of a large data center.

For traditional business applications with three-tier architecture it is a big challenge to fulfill SLOs for high-level processes within the business application because it is difficult to predict execution times accurately as low-level data management operations cannot be correlated well to high-level business processes. Presentation layer, application layer and database layer are separated and several round-trips between the different layers may be necessary even for a single business transaction. This may be one of the reasons why SLAs for business applications usually lack stringent SLOs and significant penalties today. With stricter SLAs it would be easier to compare different cloud offerings with on-premise solutions and thus cloud computing could become more attractive for potential customers. Moreover, the separation of presentation layer, application layer and database layer introduces many layers of abstraction whose overhead and complexity should be eliminated in a SaaS application to minimize administration and maintenance costs.

Proposed Changes to Three-Tier SaaS-Architecture

We propose two changes to the traditional three-tier architecture for business applications and the SaaS scenario.

The first change concerns the presentation layer. Modern web-browsers, like Mozilla Firefox[5] and Google Chrome[6], support techniques like AJAX and HTML5 that allow generating dynamic web pages within the browser on the client device. If all dynamic content is generated within the users' browsers, web servers are only needed for delivering static content. This approach eliminates the presentation layer and relocates the generation of dynamic content from the data center onto the users' browsers. Beyond that, HTML5 features for offline web applications allow to cache static content on client devices.

[5] `http://www.mozilla.com/en-US/firefox/fx` (retrieved 08/28/2012)

[6] `http://www.google.com/chrome?hl=en` (retrieved 08/28/2012)

The second change concerns the application layer. Most major database vendors support some form of stored procedures that allow to process business logic within the database layer. Thereby the number of round-trips between the application layer and the database layer can be reduced. Today the programming languages provided for stored procedures vary by vendor and mostly include proprietary extensions. Traditional tree-tier business applications typically do not use stored procedures much, to make it easier to support several database systems of different vendors. But a SaaS provider probably will use DBMSs of the same vendor for most or even all customers. Furthermore, application layer and database layer should be tightly integrated in our SaaS scenario in order to minimize administration and maintenance costs. Emerging main-memory DBMSs like HyPer [64] and SAP HANA [44] allow to process business logic inside the DBMS. Thereby performance may be improved significantly, as the number of round-trips between application servers and the DBMS can be reduced. We propose to process all business logic within the database system. Thereby, the application layer effectively is merged into the database layer.

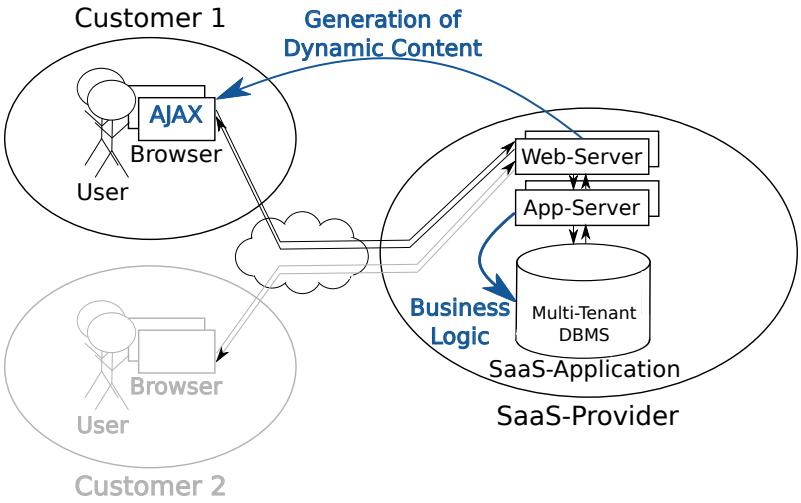


Figure 4.3: Proposed Changes to Three-Tier SaaS-Architecture

Figure 4.3 illustrates these changes and shows how to eliminate presentation and application layer. Such a simplified non-three-tier architecture makes it easier to correlate low-level data management operations with high-level business processes.

Proposed SaaS-Architecture

The proposed SaaS architecture generates dynamic content on the users' browsers and merges the application layer with the database layer. The correlation of low-level data management operations with high-level application processes is simplified, as business transactions and periodic business queries can be processed with only one round-trip, as described below.

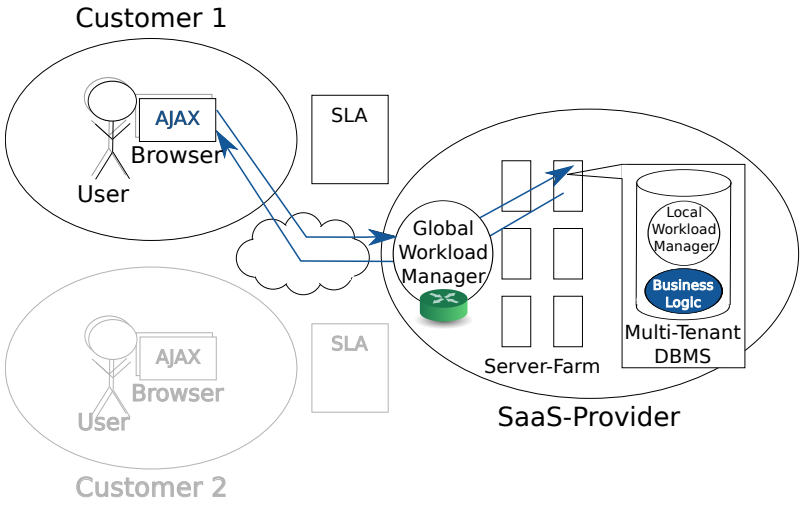


Figure 4.4: Proposed SaaS-Architecture

Figure 4.4 illustrates the proposed SaaS architecture. SaaS providers operate data centers with large server farms. Users access a SaaS application running on one of these server farms over the Internet. As discussed in Section 4.1 business transactions and periodic business queries are canned and interaction-free. Therefore, requests only need to contain the requested business transaction or periodic business query type and parameter values. A modern AJAX-based user interface runs on the user's browser and sends requests to a routing component of the global workload manager which directs requests to the server to which the given tenant is currently assigned. The multi-tenant DBMS instance on this server processes incoming requests completely and sends a response back to the user interface.

Apart from processing business logic, the multi-tenant DBMS manages the data. Each server runs one instance of the multi-tenant DBMS. Several customers or tenants are assigned to the same multi-tenant DBMS instance, e.g., Customer1 and Customer2 in our example. A local workload management component, which is part of the multi-tenant DBMS, ensures that the workloads of different tenants do

not interfere with each other. We assume that the capacity of a single off-the-shelf server is sufficient for the workload of any tenant. Our approach could be extended to support database clusters, but this is out of the scope of this thesis. If the computing capacity of a single server is not sufficient for the workload of all assigned tenants, some of the tenants have to be migrated to different servers which have sufficient resources available. A global workload manager takes care of tenant placement within the server farm. For security reasons, firewalls and encryption should be used.

In our SaaS scenario, SLAs define what the customers can expect from the service, what the service provider has to deliver and what happens if the provided service does not fulfill these requirements. Today, these SLAs are typically pretty vague, which may discourage businesses to adopt a cloud computing strategy. In our scenario, SLAs should include maximal response time and minimal throughput guarantees for business transactions and periodic business queries to attract more potential customers. SLOs should be adapted constantly to the requirements of the customer, as the service provider may project data volume growth and estimate resource requirements based on these objectives.

4.5.2 Multi-Tenant DBMS Architecture

As a multi-tenant DBMS, MobiDB has to process the workloads of all assigned tenants according to their SLOs. MobiDB incorporates native support for multi-tenancy and is optimized for multi-core CPUs. Business transactions of different tenants are executed in parallel according to the one-thread-per-core model, as data is partitioned by tenant and each data partition is assigned to one CPU core. MobiDB is optimized for the mixed workload of our Operational Business Intelligence scenario. According to the *Request Queuing and Resource Reservation* approach, incoming business transaction and periodic business query requests are not processed right away, but added to a queue and processed later. The queued requests are analyzed to estimate how long it would take to perform the queued requests. MobiDB decides adaptively when to execute a queued request based on this analysis and the required throughput rate and response time guarantees. Execution times can be estimated quite accurately, as all data is kept in main-memory and interferences between different users and different tenants are precluded, as requests are processed according to the one thread-per-core model and different components of our mixed

workload are separated by time, i.e., read-write business transactions are executed in a different phase than read-only business queries. Time is divided into fixed-length intervals. The interval length is determined based on an analytical model which is described in the next section (Section 4.5.3).

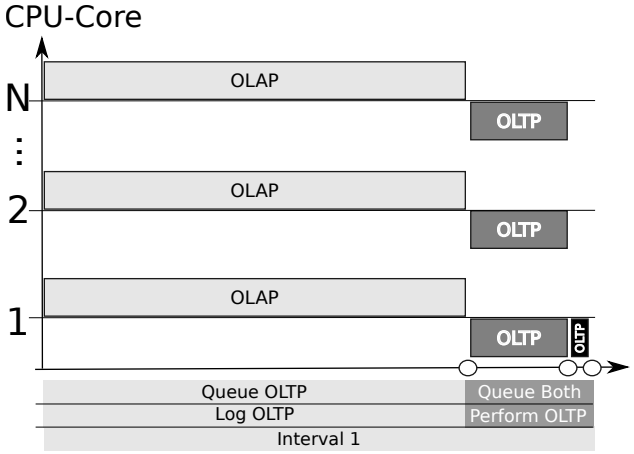


Figure 4.5: Queuing Approach Interval 1

In the following the queuing approach is explained in more detail. Figure 4.5 shows an interval that illustrates how MobiDB processes business transaction and periodic business query requests. The first part of the interval is the OLAP phase during which periodic business queries are performed. During this phase, business transaction requests are queued and logged to disk without performing the corresponding business transactions. This works for our workload, because business transactions are deterministic, canned and interaction-free. Therefore, business transaction requests can be replayed and it is sufficient to log the business transaction requests and not the changes made by business transactions. This special-purpose logging approach is explained in detail at the end of this section. The queued business transaction requests are executed in batch during the OLTP phase at the end of the interval. Requests corresponding to different tenants are separated based on the tenant ID parameter and are executed in parallel on different cores of a multi-core server. Cross-tenant transactions may also be required from time to time, for example if a larger company uses one tenant per department. These cross-partition business transactions are executed in a short single-threaded single-core phase at the end of the OLTP phase. Requests for business transactions and periodic business queries which arrive during the OLTP phase are queued. There is some synchronization overhead at the beginning and the end of each phase. The synchronization

points are indicated in Figure 4.5 by dots on the time axis. This approach can be implemented efficiently with hardware supported barrier synchronization methods. The space overhead of our approach is limited, as additional space is only required for queuing requests, but requests are typically small - a couple of numbers and strings - and the number of requests depends only on the guaranteed request rate, not on the data volume of the database.

The required queue length is determined based on the analytical model described in Section 4.5.3 and excess requests may be dropped when the queue is full. Small bursts in demand can even out across tenants which are assigned to the same DBMS instance, as the queue is shared between tenants. To avoid interferences between tenants, required requests - which have to be processed according to guarantees - have to replace queued excess requests of other tenants.

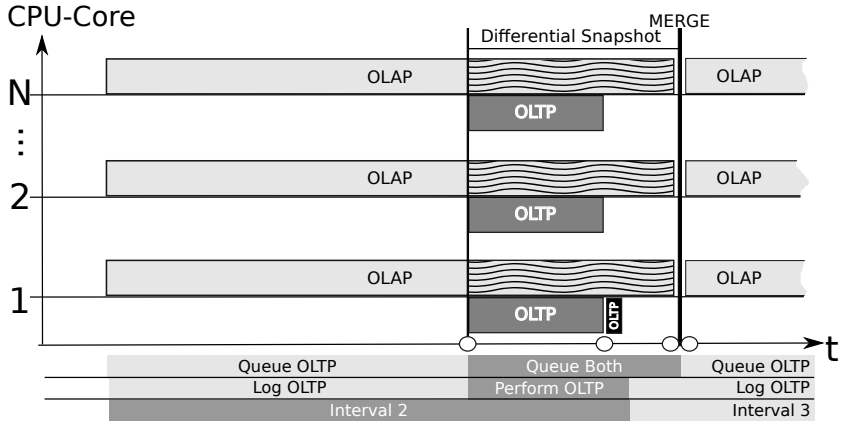


Figure 4.6: Queuing Approach Interval 2

In addition to business transactions and periodic business queries, ad-hoc business queries may be required from time to time. As this kind of query is not known in advance, no guarantees can be given. Furthermore, ad-hoc business queries may slow down and delay the execution of business transactions and periodic business queries. This may lead to a violation of response time and throughput guarantees. Therefore, MobiDB performs ad-hoc business queries in a best-effort manner while processing business transaction and periodic business query requests according to guarantees. Intervals two and three of Figure 4.6 show how MobiDB processes ad-hoc business queries. Ad-hoc business queries which arrive during the OLTP phase are delayed until the next OLAP phase. During the OLAP phase, ad-hoc queries are delayed until all queued periodic business queries have been performed. If an ad-hoc

query does not finish before the end of the OLAP phase, a differential snapshot is created on-demand before the OLTP phase starts. Differential snapshots are implemented using a delta mechanism and therefore can be created with low overhead. All changes made during the OLTP batch execution phase are recorded in the delta which has to be considered only by business transactions. Ad-hoc business queries are performed without considering the delta. The differential snapshot is dropped as soon as possible by merging the delta into the current database state. During the OLTP phase, business transactions block ad-hoc queries, as two threads per core are used instead of the pure one-thread-per-core model and the OLTP thread has higher priority than the OLAP thread. Only ad-hoc business queries are performed on the differential snapshot to ensure serializability. Periodic business queries which arrive while a differential snapshot is active have to be queued. MobiDB may decide to kill ad-hoc business queries, when the delta becomes larger than a defined threshold or when it detects that queued periodic business queries would miss their response time guarantees. For this decision, MobiDB has to consider the time to merge the delta which is estimated based on the current size of the delta.

The execution time of a business query may be reduced by using intra-query parallelization. Thereby even more complex queries may be performed according to the queuing approach. This seems very promising, as modern servers have many processing cores and there are no conflicts during the OLAP phase as only read-only queries are performed. Even with sub-optimal speedups, it may be possible to reduce overall costs with the queuing approach, if intra-query parallelization enables to process complex queries according to the queuing approach and the resulting reduction of space overhead allows for more consolidation. In our scenario, intra-transaction parallelization does not make sense for OLTP, as the execution times of business transactions are very short and because there are sufficient business transactions per time interval to keep all CPU cores busy, as we assume many parallel and independent OLTP sessions. Inter-transaction parallelization makes sense, but may require synchronization. The most common synchronization approach in relational DBMSs is based on pessimistic concurrency control with locks at different granularities. A description of multi-granularity locking can be found in [62]. MobiDB partitions data by tenant and employs locking at tenant granularity to minimize synchronization overhead and utilize all available processing cores in our multi-tenancy scenario.

Serializability

For a DBMS that guarantees serializability[7], all generated transaction histories have to be serializable [42]. According to conflict serializability, as defined in chapter two of [15], a transaction history is serializable if it is equal to a serial execution history. Two histories are equal if conflicting operations are executed in the same order. Two operations (read or write) of different transactions are conflicting operations if they operate on the same data item and at least one of them is a write operation. Single-partition business transactions, which represent read-write transactions, are performed sequentially by a single read-write thread which is responsible for that data partition. Cross-partition business transactions are performed by an exclusive read-write thread and longer-running business queries, which represent read-only transactions, are performed concurrently using one or more read-only threads. Therefore, the partial history of the read-write transactions of one data partition is serializable, as it is a serial execution, and the partial history of the read-write transactions of several data partitions is serializable, as there are no conflicting operations between data partitions. Furthermore, the partial history of the read-only transactions is serializable, as there are no conflicting operations. But the complete history may not be serializable. As both transaction types work on the same data, non-serializable histories are highly probable when read-only transactions read large parts of the database and there are many concurrent read-write transactions. To improve performance, the different transaction types can be separated by performing read-only transactions on a snapshot of the database and read-write transactions on the current database state. When read-only transactions are performed on older snapshots of the database, the complete history may not be serializable. In the worst case, the snapshot on which a read-only transaction is performed has to reflect the consistent database state right before the read-only transaction was started. This is due to write-read conflicts which is the only kind of conflict that can occur between a read-write and a read-only transaction. For example, a read-only transaction T_{ro} reads data item x among other data. It may be the case, that a read-write transaction T_{rw} changed data item x and was committed right before T_{ro} was started. In this case, T_{ro} has to see the changes made by T_{rw} , as the conflicting operations have to be performed in this order. Creating a new snapshot for each read-only transac-

[7] As defined in Section 4.4.1, we use this abbreviation for conflict serializable.

tion would cause a lot of overhead. MobiDB reduces this overhead by delaying the start of read-only transactions. Several read-only transactions are executed together on a consistent database state and during the corresponding OLAP phase no read-write transactions are performed. Data freshness is guaranteed, as the consistent database state reflects all committed changes before the OLAP phase begins.

Special-purpose Logging Approach

Commercial database systems can be configured to fulfill ACID properties and typically use write-ahead logging and the ARIES transaction recovery method [79] to ensure durability. Even main-memory database systems, like TimesTen, ensure durability by checkpointing and logging [108]. According to the write-ahead-log protocol and the force-log-at-commit rule [53], all log records containing changes made by a given database transaction have to be written to stable storage before the database transaction can be committed. Main-memory is usually volatile. Therefore log records have to be flushed to stable storage, e.g. hard disks. Especially in main-memory database systems, logging may dominate response times of database transactions, as the log flush may be the only access to stable storage required for each transaction. With high transaction rates, bandwidth of stable storage may become a performance bottleneck. In the following, we give an overview on techniques to reduce the performance impact of database logging and present a special-purpose logging approach for our application scenario.

Several database systems, including TimesTen, have configuration options to relax durability guaranties. Performance may be improved by committing transactions without waiting for log records to be flushed to disk. The database log can be kept in main-memory and be flushed to disk asynchronously. But in case of a system crash, all committed transactions whose log records have not been flushed to disk yet may be lost. This approach is not feasible for application scenarios that require ACID properties and there are other techniques to reduce the performance impact of database logging without sacrificing durability.

Group commit aims to utilize the bandwidth of stable storage devices more efficiently by processing log-flush requests of several transactions together. Log records, including commit records, are appended to a queue in main-memory. But the user is not informed that a transaction has committed until all log records up to the transaction's commit record have been flushed to disk. Several log records from the

queue are flushed to disk together in a single disk operation. Transactions whose commit records are transferred by the same disk operation are committed as a group, as described by DeWitt in [41].

Stonebraker et al. argue, that even with group commit, forced writes of commit records cause too much performance overhead [105]. Instead, they suggest, that recovery for transaction systems can be accomplished by copying missing state from other database replicas. Such an approach was introduced in [72] for highly available data warehouse systems. Highly available database systems typically employ some form of replication to tolerate failures. If a replica fails, it may be possible to recover by using data from other replicas. First the failed replica has to be brought into a consistent state. This may be achieved by restoring the last savepoint and requires that savepoints are created periodically at all replicas. Changes since the given savepoint may be retrieved from other replicas. But this requires that replicas keep a certain amount of historic data. In order to tolerate k failures (k-safety) during the interval required to recover a single replica, at least k+1 replicas are required. This requires redundant copies of the data and also causes space overhead on single replicas, as a certain amount of historic data has to be kept. The amount of historic data that is kept determines the earliest possible point in time for a point-in-time recovery which may be required due to human mistakes or software errors. The level of durability that can be achieved with this approach depends on the availability and reliability of the distributed replication system. One of the first commercial systems that implement this approach is VoltDB [8]. But even VoltDB relies on periodic database snapshots to stable storage apart from k-safety [9] and VoltDB Enterprise Edition has additional commercial logging features [10].

Group commit can be further optimized. One technique is called early lock release and has already been described by DeWitt [41]. A transaction can release all locks, once it has added its commit record to the in-memory log queue. Thereby, other transactions can read dirty data of such a pre-committed transaction and thus become dependent on it. The pre-committed transaction has to be committed before its dependent transactions, which can be ensured by flushing the log records from the in-memory queue to disk sequentially in a FIFO manner. Another technique has been proposed recently by Johnson et al. [60] and is called flush pipelining. Apart

[8] `http://www.voltdb.com` (retrieved 08/28/2012)

[9] `http://community.voltdb.com/faq` (retrieved 08/28/2012)

[10] `http://community.voltdb.com/docs/UsingVoltDB/ChapCmdLog` (retrieved 08/28/2012)

from disk I/O, latency may be caused by context switches which are required to let other processes (or threads) use processing resources while a process (or thread) is waiting for I/O operations. Johnson et al. state that the resulting scheduling overhead is significant, especially for systems with multi-core architecture and fast solid state storage devices. Flush Pipelining tries to eliminate this scheduling bottleneck by decoupling the transaction commit from thread scheduling. The technique is based on a daemon thread, which performs the actual log flush, and so-called "agent threads", that are able to execute other work during log flush by enqueuing state at the log.

Purely single-threaded transaction processing systems process transactions sequentially. Therefore, the current transaction cannot commit before all log records have been flushed to disk and all succeeding transactions have to wait. Emerging main-memory database systems, like HyPer, process transactions according to the single-threaded model (at least those of a given partition), but run certain tasks in concurrent threads to improve performance. This allows using group commit, as the log buffer can be flushed to disk asynchronously and users are informed that a transaction has committed once the corresponding commit record has reached stable storage, as described in [64]. Processing the next transaction before the current transaction's commit has been finished completely, allows transactions to read dirty data of the pre-committed transaction. This is somewhat similar to the early lock release technique. Although a single-threaded transaction processing system may work without locks when a single thread exclusively performs all processing on a given partition. Moreover, a single-treaded transaction processing system tries to minimize context switches as does flush pipelining.

Emerging main-memory database systems, like HyPer [64], use a special kind of log records, that does not contain changes made by transactions, but instead the ID and parameter values of stored procedures. This special form of logical logging works for deterministic transactions that correspond to a stored procedure which is called with the given parameter values. The major advantage of this approach is that the number of log records may be reduced significantly, as there is only one log record per transaction. If the number and size of parameters is small, overall log size may be reduced. In combination with consolidation, the available stable storage bandwidth may be utilized more efficiently. Furthermore, Harizopoulos et al. argue that the cost for executing a transaction in a main-memory database system is low

enough to compete with the cost for replaying the changes made by a transaction based on a traditional redo log [59]. A specific requirement of this approach is that a change history of stored procedures has to be archived for being able to replay the log records.

We propose a special-purpose logging approach. Business transactions are deterministic, canned and interaction-free in our application scenario. Therefore, business transaction requests can be replayed and it is sufficient to log the business transaction requests and not the changes made by business transactions. Thomson et al. observed that deterministic transactions can be ordered in advance [109]. Similarly, we suggest writing logical log records to stable storage without even executing the corresponding database transactions. The log sequence on disk defines an order according to which the business transactions - corresponding to the logged business transaction requests - have to be executed. Actually only a partial order is required, as it is sufficient to define A) the order of cross-partition business transactions, B) the order of single-partition business transactions within a given partition and C) the relative order of single-partition business transactions and cross-partition business transactions. The order between single-partition business transactions of different partitions does not have to be defined, as described in [64], and allows for parallelism. We apply this approach to our mixed workload scenario, and take advantage of the mentioned characteristics with the queuing approach. Log records are written during the OLAP phase before the corresponding business transactions are executed during the OLTP phase. Thereby, there is no disk I/O during the OLTP phase which helps to make execution times of short-running business transactions more predictable. Similar to group commit, requests can be accumulated and logged to disk using bulk transfers. Moreover, the optimization techniques early lock release and flush pipelining can be applied to single-threaded transaction processing system as discussed above.

4.5.3 Analytical Model

In this section, we present an analytical model for processing business transactions and periodic business queries according to the queuing approach without snapshots. For ad-hoc business queries, we analyze the space and processing overhead of differential snapshots with different granularities.

Queuing approach

The analytical model can be used to determine a suitable interval length for a given set of business transactions and periodic business queries with given response time and throughput rate guarantees, a given data volume and a server with given capacity. Execution times are estimated based on test runs which are performed on a server with the given capacity. If the required throughput rate is too high or the required response time is too low, a server with the given resources may not be sufficient and the analytical model determines that no suitable interval length exists. In this case, a different server of the server farm with more capacity has to be used. We assume that the workload of any single customer or tenant fits on a single server of the server farm.

We define a mapping tp which returns the expected throughput guarantee for a given business transaction type (BT_0 .. BT_i) or periodic business query type (BQ_0 .. BQ_j). Then the maximum number of requests per interval of each business transaction type and periodic business query type can be determined, based on the expected throughput (tp) and a given interval length (IL). We refer to this figure as the request count (rc).

$rc(x) = tp(x) * IL$

We define a mapping et which returns the average execution time for processing a given business transaction type or periodic business query type on a given data volume. Based on the request count (rc) and the execution time (et), the overall execution time of business transactions (ET_T) and periodic business queries (ET_Q) can be calculated.

$ET_T = \sum_{x=BT_0}^{BT_i} rc(x) * et(x)$

$ET_Q = \sum_{x=BQ_0}^{BQ_i} rc(x) * et(x)$

We define a mapping rg which returns the response time guarantee for a given business transaction type or periodic business query type. Based on the response

time guarantee, the minimal response time guarantee of business transactions (MRG_T) and periodic business queries (MRG_Q) can be calculated.

$MRG_T = \min_{BT_0}^{BT_i} rg(x)$

$MRG_Q = \min_{BQ_0}^{BQ_i} rg(x)$

Based on the minimal response time guarantee, the overall execution time and the average network delay from client to server (ND), the maximal acceptable delay of business transactions (MD_T) and periodic business queries (MD_Q) can be calculated.

$MD_T = MRG_T - ET_T - 2 * ND$

$MD_Q = MRG_Q - ET_Q - 2 * ND$

A suitable interval length has to fulfill the following condition.

$ET_T <= MD_Q \wedge ET_Q <= MD_T$

If it exists, such a suitable interval length can be determined as follows.

$IL = \min(MD_T + ET_T, MD_Q + ET_Q)$

The maximal acceptable delay and the overall execution time depend on the interval length themselves which can be resolved via fixed point iteration.

For a given interval length, the required queue length QL can be determined as follows.

$QL = \sum_{x=BT_0}^{BT_i} rc(x) + \sum_{x=BQ_0}^{BQ_i} rc(x)$

The expected resource utilization RU can be determined as follows.

$RU = \frac{ET_T + ET_Q}{IL}$

If the resource utilization is less than one, the given server is oversized. By consolidating several tenants onto the same infrastructure, resource utilization can be improved. Tenants can be consolidated if there is a suitable interval length for their combined workload.

Differential Snapshots

The space overhead of differential snapshots depends on the update characteristics of the application. At creation time of a snapshot, all data can be shared between current state and snapshot. Data can be appended without altering the snapshot, as it is sufficient to record the information about the last valid row of each table. But when existing data is overwritten, the old version of the data has to be kept for the snapshot and the new version has to be made available for the current state. Therefore, the incurred space overhead depends on the granularity of the

snapshot technique and the distribution of updates across the data set. We focus on row granularity, which seems very suitable for transactional business applications which typically process data row-wise, and page granularity, which can benefit from hardware supported operating system mechanisms. So either an entire row or an entire page may be copied, when a value is updated. At most two copies are required and it makes no difference how often a given row or page is updated until the snapshot is merged.

If we assume that updates are distributed uniformly across the data set, the process can be modeled by an urn model with replacement and without order. The expectation for random variable X, which represents the number of "dirty" blocks containing updated objects, can be determined according to Cardenas formula.

$E[X] = b * (1 - (1 - \frac{1}{b})^k)$

There are N objects, and b blocks. For page granularity, each block has the size of a page. Each block contains N/b objects and there are k update operations which are distributed uniformly among the N objects and the b blocks. For row granularity, each row corresponds to an object. Therefore, b = N and each block contains exactly one object as N/b = 1.

The number of update operations depends on the rate at which operations are performed and the operation mix. The maximal rate without snapshot may be reduced due to overhead of the snapshot technique and its implementation. For page granularity, we model the HyPer approach as described in [64]. This approach requires forking a new process at the beginning of each interval to create the snapshot. The duration of this operating system operation depends on the amount of memory used by the forked process, as among other things the page table of the parent process is copied. Depending on the hardware, this operation may take around 8 ms per 1 GB of memory that is used by the parent process. If we model the data volume as a constant, then this operation reduces the maximal rate and the difference to the maximal rate without snapshot represents the incurred performance overhead. For row granularity, a delta mechanism can be used which imposes only very low overhead for snapshot creation. Apart from the creation, the process of releasing a snapshot may also cause performance overhead. For page granularity, the snapshot can be released by terminating the child process at the end of the interval. Among other things, this operating system operation frees the page table copy and frees pages which are not referenced any more. As the operating system performs these

operations lazily to even out the associated performance overhead, we do not have to take this into account. For row granularity with delta mechanism, the delta has to be merged at the end of each interval which can have a significant performance impact. The duration of the merge operation depends on the size of the delta. The maximum size of the delta depends on the maximum number of update operations performed per interval and the resulting number of dirty blocks. We define the function db which returns the number of dirty blocks for a given rate. We define the function *md* which returns the merge duration for a given rate. The merge duration can be estimated based on the number of dirty blocks for the given rate and the average merge duration per block (*amd*).

$md(rate) = db(rate) * amd$

The maximal rate in turn depends on the duration of the merge operation, as no other operations can be performed during the merge operation, and can be determined as follows. We define a helper function f that returns the deviation from the interval length for a given rate based on the average duration of a single operation (c), the merge duration for the given rate and the interval length (IL).

$f(rate) = rate * c + md(rate) - IL$

The maximal rate at which operations can be performed with row granularity and delta mechanism can be determined by finding the location of a function root of f. The difference between this rate and the maximal rate without snapshot represents the incurred performance overhead.

The space overhead depends on the update characteristics of the application. For our application scenario, hot spots are characteristic and therefore uniform distribution is unrealistic. To get a more realistic estimation of space overhead, we split the data set into two subsets - hot and cold - and assume uniform distribution only within each subset. The number of hot spots has to be considered and can be modeled as the distribution of the hot subset across the data set.

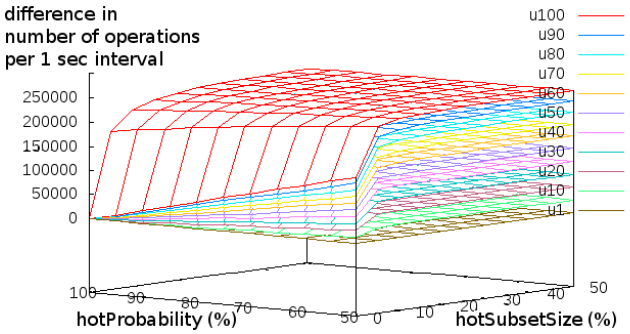
4.5.4 Experimental Evaluation

For evaluating OLTP performance of MobiDB, we use a benchmark similar to TPC-C [111]. In [61], also a TPC-C-like benchmark was used for evaluating H-Store. For MobiDB, we implemented a TPC-C-like benchmark that relies solely on point-wise operations. MobiDB achieves throughput numbers of more than 120'000 new-orders per second with 8 warehouses on an off-the shelf server (2 Intel Xeon X5570 Quad

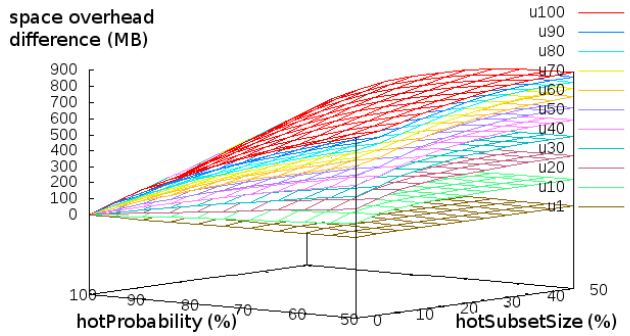
Core-CPUs, 2.93 GHz, 64 GB RAM, Linux). Thus, OLTP performance of MobiDB is in the same order of magnitude as H-Store, with published numbers of more than 70'000 new orders per second on a slightly less powerful machine [105]. MobiDB and the benchmark are implemented in pure Java while H-Store relies on a native library written in C++. Furthermore, as H-Store relies on replication instead of logging, we disabled logging in MobiDB. Moreover, cross-warehouse transactions were disabled for this comparison, because H-Store did so, as described in [114].

To evaluate the mixed workload performance of MobiDB, we used a fixed TPC-C-like OLTP workload of 6'250 new-orders per second with cross-warehouse transactions and evaluated how much OLAP business queries can be performed while adhering to the guarantees for OLTP. The used OLAP workload consists of one simple periodic business query type called top-customer-query which determines the ten best customers for a given warehouse and a given district according to sales volume based on the TPC-C schema. The top-customer-query meets our workload characteristics with an average execution time of 10 milliseconds while TPC-C-like business transactions have execution times of only a couple of microseconds. We measured a throughput of 6'250 new-orders per second and the 95th percentile of response times was below 1'075 milliseconds (the upper limited for the 90th percentile required by the TPC-C Specification is higher than two seconds [111]). Thus, the guarantees for OLTP were fulfilled and, at the same time, an OLAP throughput of almost 250 top-customer queries per second was achieved.

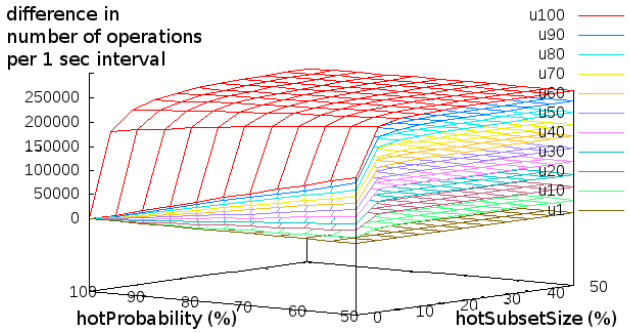
For business transactions and periodic business queries, the space overhead of MobiDB is limited, as additional space is only required for queuing requests which depends on the guaranteed request rate and the request size. MobiDB is a special purpose system which minimizes space overhead for its application scenario while guaranteeing serializability. It is difficult to compare this with other systems. On the one hand, commercial DBMSs only achieve much lower throughput rates for mixed workloads, even at the reduced isolation level read-committed, as shown by Funke et al. in [47] with System "X". On the other hand, HyPer achieves even higher throughput numbers than MobiDB and H-Store using a novel query compilation strategy that aims at good code and data locality in combination with a predictable branch layout [81]. But HyPer incurs a certain space overhead, as snapshots are required even for periodic business queries. For ad-hoc queries, the space and performance overhead of differential snapshot mechanisms with row and



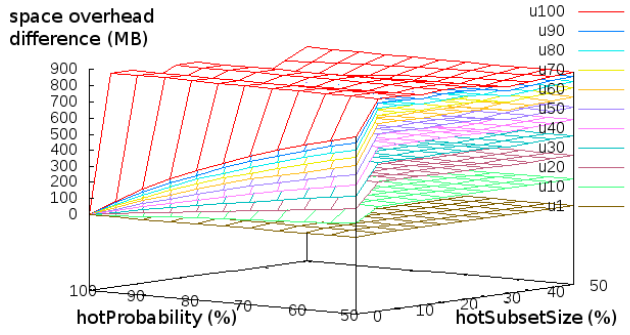
(a) Scenario 1: Difference in performance overhead



(b) Scenario 1: Difference in space overhead



(c) Scenario 2: Difference in performance overhead



(d) Scenario 2: Difference in space overhead

Figure 4.7: Comparison of row and page granularity snapshots

page granularity can be compared according to our analytical model. We assume a data volume of 1 GB per tenant, consisting of approximately 10 million small objects

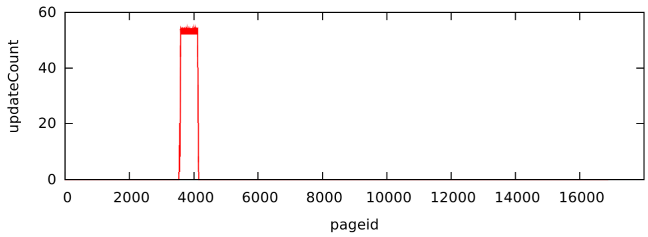


Figure 4.8: TPC-C-like update characteristics

with average size of 100 byte, and look at the overhead caused by a single tenant. The operations mix represents business transactions and is made up of read, append and update operations. Read operations form 74 percent of the operations mix similar to TPC-C and the distribution of the remaining operations among append and update operations is varied according to the update share between 1% and 100% (u1 - u100). Time is divided into 1 second time intervals; a snapshot is created at the beginning of the interval and merged before the end of the interval. The size of the hot subset and the probability that an update hits the hot subset is varied to model different application characteristics. Regarding the number of hot spots or how the hot subset is distributed across the data set, we look at two extreme scenarios. Figures 4.7a and 4.7b show results for the first scenario where hot objects are clustered together in a single hot spot. Figures 4.7c and 4.7d show results for the second scenario where hot objects are distributed uniformly across the data set to model many small hot spots. The characteristics of real applications probably lie somewhere in-between these two extreme scenarios, but these extreme scenarios show that the best trade-off between space and performance overhead depends on the characteristics of the application. Some applications can reduce their space overhead with row granularity and the resulting performance overhead may still be acceptable. Some applications may accept higher space overhead in order to reduce performance overhead. Depending on application characteristics, the space overhead may reach almost a factor of two, although snapshots are only active for less than 1 second. This corresponds to space overhead which can be saved with the queuing approach for every second that no ad-hoc business query is in the system. All in all, the queuing approach is feasible for the mixed workload of our application scenario.

We analyzed the space overhead of row and page granularity snapshots with TPC-C and measured only a very small difference. This is due to the update characteristics of TPC-C. We did a micro-benchmark with our TPC-C-like implementation to analyze its update characteristics. The WAREHOUSE table for a single warehouse

accounts for 1 page and per second 16'611 updates are performed on this single page. The DISTRICT table also accounts for 1 page and 75'303 updates per second. The CUSTOMER and STOCK tables account for 5'266 and 8'252 pages respectively. During the time interval of one second all pages of these tables are updated, but with varying frequency. The ORDER table accounts for 16'874 pages, but only a few hundred pages are updated during a one second time interval (see Figure 4.8). The tables ITEM, HISTORY and ORDER-LINE are never updated and account for 2'392, 37'499 and 487'499 pages respectively at the time of measurement. The HISTORY and ORDER-LINE tables constantly grow during the benchmark run. If TPC-C would represent realistic update characteristics of a real business application, it would not matter if page or row granularity was used as the space overhead of both is very small and merging deltas of this size only causes very limited performance overhead. Probably the TPC-C benchmark was not built for that purpose. Furthermore, for this kind of business applications a pure main-memory approach would be a waste of resources, as most data is cold. A hybrid approach which offloads cold data to a different media, like flash memory, would be much better suited. A hybrid approach keeps only hot and warm data in main-memory. Thus, the difference between page and row granularity matters as the above scenarios show.

4.6 Conclusions

The presented SaaS architecture combined with the queuing approach used by MobiDB, may enable SaaS business applications, like CRM, with more powerful analytical features at competitive prices. Furthermore, the presented approach enables service providers to offer strict SLAs with stringent response time and throughput guarantees. With stricter SLAs it would be easier to compare different cloud offerings with on-premise solutions and thus cloud computing could become more attractive for potential customers.

The main difference between OLTP and OLAP workloads is that they have different predominant access patterns (point-wise vs. scan). This results in a big difference with regard to average execution times. To a certain degree, it is possible to change the access pattern of a query by materialization. Introducing materialization can turn a scan-based access into a point-wise access and removing materialization can turn a point-wise access into a scan-based access. If materializations are not kept up to date, this is similar to reduced isolation levels and leads to additional complexity in the application layer. Materializations are typically kept up to date by the application code of business transactions, e.g. revenue per district in TPC-C. Today, introducing or removing such materializations requires changes to the application code. The overall space and performance overhead with and without materializations depends on the data representation of the DBMS, e.g. row-store or column-store, and hardware characteristics, e.g. cache hierarchy. As DBMSs and hardware typically evolve during the life cycle of an application, it is a bad idea to specify materialization by manual coding at development time. Recently, techniques are emerging, that allow translating declarative query languages automatically into either scan-based queries or point-wise accesses and imperative code for updating materializations. Ahmad et al. propose such a technique as part of the DBToaster project [5] and recently presented promising experimental results [4]. Such declarative query languages provide more flexibility and may enable the DBMS to apply materializations adaptively.

Chapter 5

Mixed Workload Benchmark

Advances in hardware architecture enable Operational Business Intelligence. Emerging database systems can process analytical queries directly on the operational database without impeding the performance of mission-critical transaction processing too much. In order to evaluate the suitability of database systems for Operational Business Intelligence, we propose the mixed workload CH-benCHmark[1], which combines **transactional load** based on TPC-C order processing with **decision support load** based on a TPC-H-like query suite. This combination constitutes a mixed workload, as both loads are run **in parallel** on the **same tables** in a **single database** system. In contrast, single-workload benchmarks can be installed on a single database instance and run in parallel, but this does not constitute a real mixed-workload, because the different loads are run on separate data.

TPC-C and TPC-H are two standardized and widely used benchmarks addressing either transactional or analytical workloads. We derived our mixed workload CH-benCHmark from these two standardized and widely accepted benchmarks, as to our knowledge, currently there is no widely accepted mixed workload benchmark. Recently another mixed workload benchmark has been devised, called Composite Benchmark for Transaction processing and operational Reporting (CBTR) that

[1]The CH-benCHmark was developed as a research project (`http://www-db.in.tum.de/research/projects/CH-benCHmark/`, retrieved 08/28/2012) lead by Prof. Alfons Kemper, Ph.D. and Prof. Dr. Thomas Neumann. Florian Funke, Stefan Krompass and I (Michael Seibold) worked together on this project. We had several external partners from the IT industry and were supported by bachelor student Adrian Streitz. After a related benchmark - called "TPC-CH" - had been published at BTW 2011 by Florian Funke et al. [47], the CH-benCHmark was devised and published at DBTest 2011 [30] and TPCTC 2011 [46].

includes OLTP and reporting components [18]. This benchmark is not based on standardized benchmarks, but instead uses the actual data of a real enterprise. In contrast, CH-benCHmark is derived from the two most widely used TPC benchmarks and produces results that are highly relevant to both hybrid and classic single-workload systems.

Just as the data volume of actual enterprises tends to increase over time, an inherent characteristic of this mixed workload benchmark is that data volume increases during benchmark runs, which in turn may increase response times of analytic queries. For purely transactional loads, response times typically do not depend that much on data volume, as the queries used within business transactions are less complex and often indexes are used to answer these queries with point-wise accesses only. But for mixed workloads, the insert throughput metric for the transactional component interferes with the response-time metric for the analytic component of the mixed workload. In order to address this problem, we analyze the workload characteristics and performance metrics of the mixed workload CH-benCHmark. Based on this analysis, we propose performance metrics that account for data volume growth which is an inherent characteristic of such a mixed workload benchmark[2].

[2]Parts of this work have been published at DBTest 2011 [30] and TPCTC 2011 [46].

5.1 Overview of the Mixed Workload CH-benCHmark

The CH-benCHmark represents a mixed database workload comprised of transactional and analytical loads that are executed in parallel on the same tables in a single DBMS. The transactional load is based on the business transactions of TPC-C and uses unaltered TPC-C tables. Thus, benchmark sponsors — hardware and software vendors who publish TPC benchmark results — can adapt their existing TPC-C benchmark kits to run the transactional load of CH-benCHmark. The analytical load consists of a read-only query suite modeled after TPC-H. The TPC-H refresh functions are omitted, as the database is continuously updated (and expanded) via the transactional load.

5.1.1 Schema and Initial Database Population

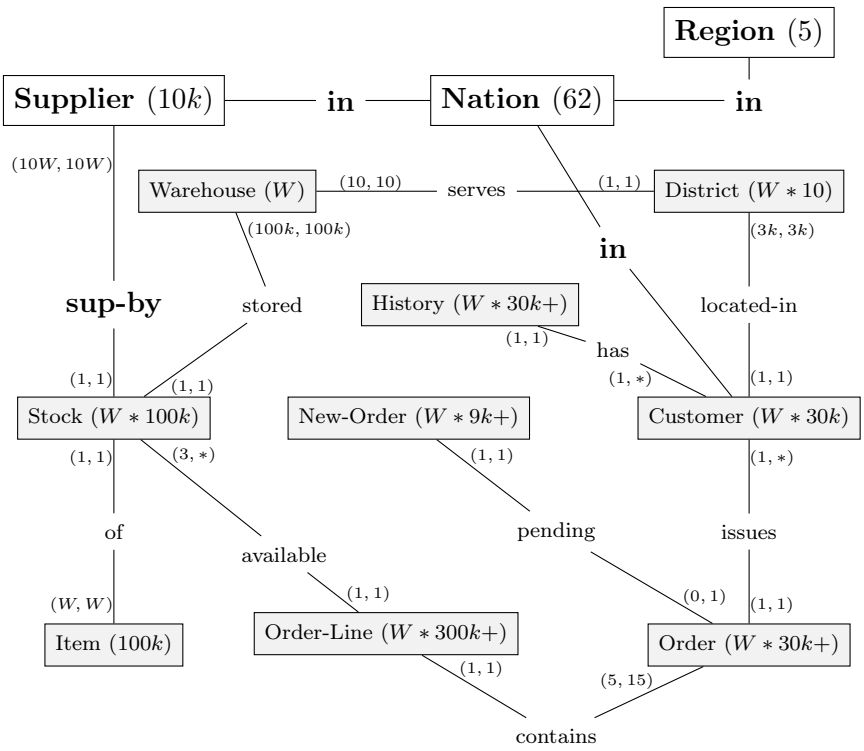


Figure 5.1: Entity-Relationship-Diagram of the CH-Benchmark Database

CH-benCHmark extends the TPC-C schema (see gray boxes in Figure 5.1) with three additional tables from the TPC-H schema: Supplier, Region and Nation (see white boxes in Figure 5.1). These additional tables are read-only, as they are

not modified during a benchmark run. The combined schema allows formulating slightly modified TPC-H queries on TPC-C-like schema and data.

Figure 5.1 denotes the cardinalities of the initial database population in brackets after the name of each entity. The + symbol is used after the cardinality of an entity to indicate that the cardinality is subject to change during a benchmark run, as rows are added or deleted. The initial database population follows the official TPC-C specification. (min, max)-notation is used to represent the cardinalities of relationships after initial database population and during benchmark runs. As in TPC-C, the WAREHOUSE table is used as the base unit of scaling. The cardinality of all other tables (except for ITEM) is a function of the number of configured warehouses (cardinality of the WAREHOUSE table). The population of the three additional read-only tables is defined as follows. The relation SUPPLIER is populated with a fixed number of entries (10,000). Thereby, an entry in STOCK can be uniquely associated with its Supplier via the following formula:
(STOCK.S_I_ID × STOCK.S_W_ID) *mod* 10,000 = SUPPLIER.SU_SUPPKEY
A Customer's NATION is identified by the first character of the field C_STATE. TPC-C specifies that this first character can have 62 different values (upper-case letters, lower-case letters and numbers), therefore we chose 62 nations to populate Nation (TPC-H specifies 25 nations). The primary key N_NATIONKEY is an identifier according to the TPC-H specification. Its values are chosen such that their associated ASCII value is a letter or number. Therefore no additional calculations are required to skip over the gaps in the ASCII code between numbers, upper-case letters and lower-case letters. Region contains the five regions of these nations. Relationships between the new relations are modeled with simple foreign key fields: NATION.N_REGIONKEY and SUPPLIER.SU_NATIONKEY.

5.1.2 Transactional Load

According to the TPC-C specification [111], the original TPC-C workload consists of a mixture of read-only and update-intensive business transactions: New-Order, Payment, Order-Status, Delivery, and Stock-Level. The TPC-C schema contains nine tables: WAREHOUSE, STOCK, ITEM, HISTORY, NEW-ORDER, ORDER-LINE, DISTRICT, CUSTOMER and ORDER (see gray boxes in Figure 5.1). The transactional load of the CH-benCHmark is very similar to the original TPC-C workload. Unchanged TPC-C business transactions are processed on unchanged TPC-C tables.

Even the initial database population follows the official TPC-C specification. But the CH-Benchmark does not simulate terminals, as TPC-C does with keying times and think times. Instead a given number of transactional sessions issue randomly chosen business transactions in a sequential manner without think times or keying times. Furthermore, the distribution of the different business transaction types follows the official TPC-C specification. Moreover, the home warehouses of business transactions are randomly chosen by each transactional session and are evenly distributed across warehouses.

5.1.3 Analytical Load

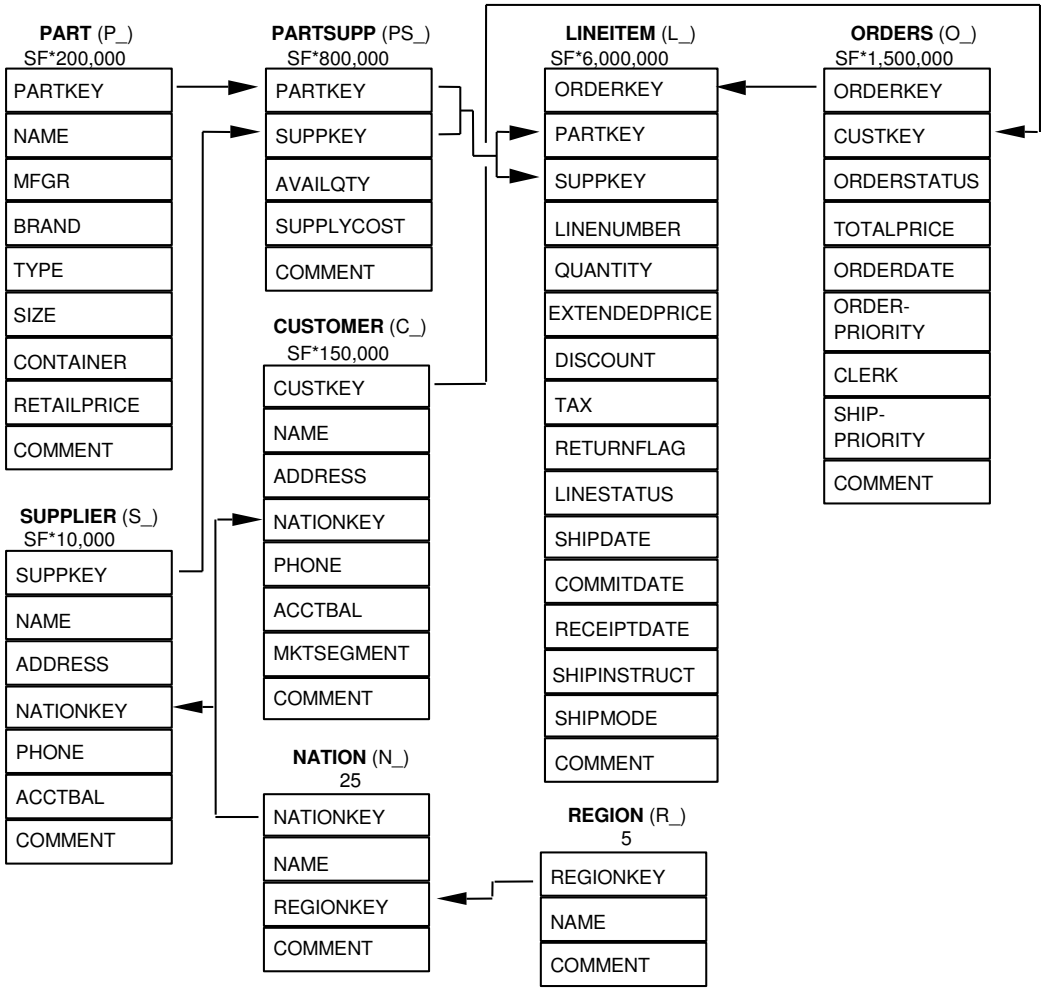


Figure 5.2: Schema TPC-H (adapted from [112])

The analytical load of CH-benCHmark is based on TPC-H. The following description of TPC-H is based on the TPC-H specification [112], which defines a decision support benchmark that was designed to be representative of complex business analysis applications — involving sequential scans of large amounts of data, aggregation of large amounts of data and multi-table joins. TPC-H models the activity of a wholesale supplier and tries to represent the activity of any industry which must manage, sell, or distribute a product worldwide. Data warehouse applications typically use a star schema, as discussed in chapter 17.2.1 of [62]. But, the database schema of TPC-H does not follow the star schema paradigm. Although there is the Star Schema Benchmark, which has a star schema and is derived from TPC-H [84], we use the original TPC-H benchmark, because its schema is more similar to TPC-C. The database schema of TPC-H is shown in Figure 5.2. It is made up of the following eight tables and one-to-many relationships between them: SUPPLIER, NATION, REGION, PART, PARTSUPP, CUSTOMER, ORDERS and LINEITEM. Column names are prefixed with the prefix mentioned in parentheses following each table's name. Furthermore, the number or formula below each table name represents the cardinality of the table which may depend on the scale factor SF. This scale factor determines the size of the initial database population. The minimum database population ($SF = 1$) contains business data from 10,000 suppliers consisting of almost ten million rows and representing a data volume of about 1 gigabyte. A similar initial database size can be achieved in TPC-C by configuring 12 Warehouses.

TPC-H comprises 22 read-only ad-hoc queries and each query is described in terms of a business question that illustrates the business context in which the query could be used. The queries have substitution parameters, that are replaced by random values selected from a uniform distribution and change across query executions. Apart from the queries, the benchmark involves two refresh functions, which perform batch modifications of the database. The "New Sales" refresh function inserts new rows into the ORDERS and LINEITEM tables and is complemented by the "Old Sales" refresh function, which removes rows from the ORDERS and LINEITEM tables. The two refresh functions are executed in pairs and are designed is such a way that the population of the test database is once again in its initial state after a certain number of pairs have been executed. Thus, the data volume does not grow significantly during a benchmark run. The amount of data inserted and deleted depends on the scale factor SF.

According to execution rules, first a load test is performed, which creates and loads the database tables. Then follows the performance test which itself consists of two different tests: Power Test and Throughput Test. On the one hand, Power Test measures the raw query execution power of the system when connected with a single active user. On the other hand, Throughput Test represents a multi-user workload and measures the ability of the system to process the most queries in the least amount of time. For the Power Test, all 22 queries are submitted by a single session of the driver to the system under test, which executes the queries one after another. In parallel, a single refresh stream executes a single pair of refresh functions. The first refresh function is scheduled before and the second one is scheduled after the execution of the queries. The result of the Power Test is the TPC-H Power metric. It is defined as the inverse of the geometric mean of the timing intervals. For the Throughput Test, two or more sessions submit queries on the system under test. In parallel, a single refresh stream executes pairs of the two refresh functions. The number of pairs that need to be executed is equal to the number of query streams used for the Throughput Test. The scheduling of the refresh function pairs within the refresh stream is left to the test sponsor. Thus, query executions can be segregated from database refreshes. The result of the Throughput Test is the TPC-H Throughput metric at the chosen database size. It is defined as the ratio of the total number of queries executed over the length of the measurement interval. The TPC-H Composite Query-Per-Hour Performance Metric (QphH@Size) combines the numerical quantities of the TPC-H Power and TPC-H Throughput metric for the selected database size. TPC-H does not constitute a mixed workload, because data modification operations - in the form of refresh functions - can be performed in bulk when no queries are running.

The analytical load of CH-benCHmark is based on the 22 TPC-H queries. Since the CH-benCHmark schema is different from the TPC-H schema, the queries are reformulated to match the schema. However, we tried to preserve their business semantics and syntactical structure[3]. But, the TPC-H-like queries are performed on extended TPC-C data which may have different characteristics than the original TPC-H data, although both benchmarks model very similar application scenarios. Furthermore, business queries read data from the extended schema, including data from the TPC-C tables and the three additional read-only tables. The contents

[3]The SQL code of the CH-benCHmark queries can be found in the Appendix.

of the unmodified TPC-C tables change during the benchmark run, as business transactions update and insert tuples and these changes have to be accounted for by the business queries depending on data freshness requirements. Therefore, analytical performance in the CH-benCHmark cannot be easily inferred from the performance of a similarly-sized TPC-H installation. Similarly to TPC-H, the analytical load is generated by a given number of analytical sessions. Each analytical session submits business queries sequentially. All 22 business query types are issued in continuous iterations over the set of query types, while each analytical session executes all 22 query types in a randomly chosen permutation sequence to avoid caching effects.

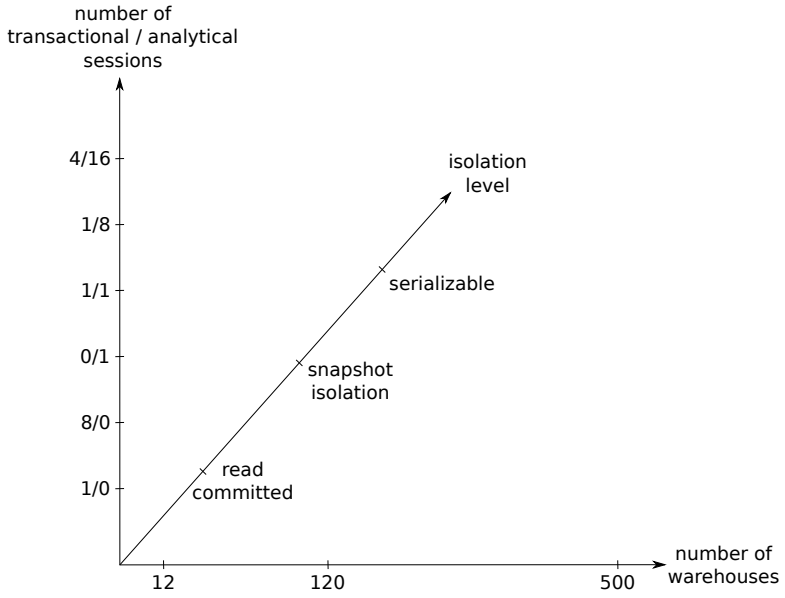


Figure 5.3: Benchmark Parameters

5.1.4 Benchmark Parameters

The CH-benCHmark provides flexible configuration options to model different workload characteristics and application scenarios, e.g. the data volume can be adjusted to fit in main-memory or to require secondary storage. Figure 5.3 illustrates parameters of the CH-benCHmark and shows some exemplary values. Database systems can be compared based on performance metrics by performing benchmark runs with the same parameter values on all systems.

First, the size of the initial database is specified by the number of warehouses, which determines the cardinalities of the other relations similar to TPC-C. Second,

the composition of the workload is specified by the number of transactional sessions and analytical sessions. This parameter allows specifying purely transactional, purely analytical and mixed workload scenarios. Third, the isolation level which has to be provided by a system under test, is a parameter of the CH-benCHmark. Lower isolation levels, like read committed, can be used to measure raw performance with limited synchronization overhead. More demanding isolation levels can be used to account for more realistic synchronization requirements. The isolation level parameter can be specified separately for the transactional and the analytical load. For mixed workload scenarios, the data freshness parameter allows to specify the time or number of transactions after which newly issued queries have to incorporate the most recent data.

5.1.5 Data Scaling

TPC-C and TPC-H employ different scaling models. A scaling model maintains the ratio between the transactional load presented to the system under test, the cardinality of the tables accessed by the transactions, the required space for storage and the number of terminals or sessions generating the system load. TPC-C employs a continuous scaling model, where the data volume has to be increased for higher transaction load. The number of warehouses determines not only the cardinality of the other tables, but also the number of terminals that generate a limited load each due to think times and keying times. For increasing transaction load, the number of terminals has to be increased, requiring a higher number of warehouses and resulting in a larger data volume. In contrast, TPC-H employs a fixed scale factor model, where the database size is set by a scale factor regardless of system performance.

The CH-benCHmark deviates from the continuous scaling model of TPC-C in order to allow for high transaction rates on small database sizes that are common for scenarios supported by emerging main-memory database systems. On the one hand, the continuous scaling model may cause higher response times for analytical queries when transactional load is increased, because analytical queries would have to process larger data volumes. In this case it would not be meaningful to compare query response times of two systems with different maximum transactional loads, as even the size of the initial database population would vary largely. On the other hand, the continuous scaling model of TPC-C requires very large data volumes and expensive secondary storage systems in order to fully utilize modern CPUs. However, the CH-

benCHmark was designed not only for traditional disk-based database systems, but also for emerging main-memory database systems. For measuring performance of high-throughput OLTP main-memory database systems like VoltDB, a TPC-C-like benchmark has been proposed which does not adhere to continuous scaling. Instead, a fixed number of warehouses is used and there are no wait times, according to the benchmark description [114]. Similarly, the CH-benCHmark determines maximum system performance for a fixed initial data volume, which is determined by a parameter of the CH-benCHmark. Thus, the initial database population is determined by a scale factor regardless of system performance, like in TPC-H. In our case, the scale factor is the number of warehouses which determines the initial data volume as in TPC-C. But in contrast to TPC-C, the number of transactional sessions is another parameter of the benchmark that does not depend on the number of warehouses and there are neither sleep nor keying times. Therefore higher system performance can be achieved without increasing the initial data volume.

During the course of a benchmark run business transactions create new orders, adding tuples to relations ORDER, ORDER-LINE, HISTORY and NEW-ORDER. Since the SUPPLIER relation is read-only, the ratio of the cardinalities of these relations changes relative to the SUPPLIER relation. The cardinality ratio relative to the SUPPLIER relation does not change for the relations WAREHOUSE, DISTRICT, STOCK, CUSTOMER, and ITEM, which are read-only or only updated in-place. Due to continuous data volume growth, refresh functions like in TPC-H are not required. But continuous data volume growth during benchmark runs poses new challenges as discussed in the next section.

5.2 Performance Metrics

In order to compare two systems, metrics for the transactional and the analytical load have to be considered. Table 5.1 gives an overview on the current performance metrics of CH-benCHmark.

Currently, the mixed workload CH-benCHmark uses performance metrics similar to those of single-workload benchmarks like TPC-C and TPC-H (see Table 5.1). The two most important metrics are *Transactional Throughput* for transactional load and *Geometric Mean* of response times for analytical load. It may seem obvious to combine the *Transactional Throughput* metric and the *Queries Per Hour* metric in

Transactional Throughput (tpmCH)	Total number of New-Order transactions completed during the measurement interval divided by the elapsed time of the interval in minutes; New-Order transactions that rollback due to simulated user data entry errors must be included; Similar to the Maximum Qualified Throughput metric of TPC-C
Geometric Mean (ms)	For each query type the average response times of queries completed during the measurement interval is determined and the geometric mean of the average response times of all query types is reported.
Duration Per Query Set (s)	Query set consists of 22 queries, one query per query type; Sum of the average response times of all query types; Reported in seconds
Queries Per Hour (QphCH)	Completed queries per hour; Can be deduced from *Duration Per Query Set* metric as follows: $\frac{60\ minutes}{\frac{Duration\ Per\ Query\ Set\ (in\ seconds)}{60}} \times 22$

Table 5.1: Performance Metrics

order to obtain a single metric, but competing systems under test may prioritize transactions and analytical queries differently and this aspect would get lost if a single metric were used. Remember that the transactional load generated by the configured number of transactional sessions is not limited by sleep times or keying times, but can only be throttled by the system under test.

Data volume growth caused by the transactional load of the mixed workload poses a new challenge. The problem is that higher transactional throughput results in larger data volume which in turn may result in longer response times for analytical queries. Therefore, currently reported performance metrics cannot be compared individually, as systems with high transactional performance may have to report inferior analytical performance numbers, although analytical queries have been performed on larger data volumes. The insert throughput metric of the transactional component interferes with the response-time metric of the analytic component of the mixed workload. Note that TPC-H does not consider a database that grows over the course of a benchmark run. To overcome this issue, we propose performance metrics that account for data volume growth which is an inherent characteristic of a mixed workload benchmark like CH-benCHmark.

5.2.1 Response Times and Data Volume Growth

During the course of a CH-benCHmark run, data volume grows over time due to inserts caused by the transactional load of the mixed workload. Figure 5.4 illustrates how response time of a query may increase with growing data volume.

During the course of a CH-benCHmark run, cardinality of the following tables increases: ORDER, ORDER-LINE, HISTORY and NEW-ORDER. The cardinality ratio between the relations ORDER-LINE and ORDER should be approximately ten and should be more or less constant during a run, because an order includes ten items on average according to the TPC-C specification. The HISTORY relation is not read by any query in the TPC-H-like query suite of CH-benCHmark and thus does not impact their response times. One could think that the cardinality of the NEW-ORDER relation would be more or less constant, as each delivery transaction delivers a batch of ten new (not yet delivered) orders and the TPC-C specification states: "The intent of the minimum percentage of mix ... is to execute approximately ... one Delivery transaction ... for every 10 New-Order transactions" [111]. But in practice our CH-benCHmark implementation, like most other implementations of TPC-C, tries to maximize the number of processed New-Order transactions and only processes the required minimum of the other transaction types. This strategy results in approximately 45% New-Order transactions and only 4% Delivery transactions that deliver ten new orders each. Therefore the Delivery transactions cannot keep up with orders created by New-Order transactions and thus the cardinality of the NEW-ORDER relation increases during a benchmark run, as approximately 11% of new orders remain undelivered.

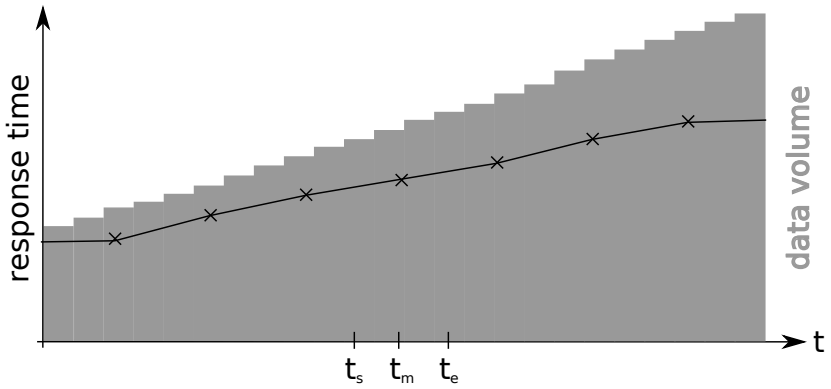


Figure 5.4: Response Times and Data Volume Growth

Whether the response time of a given query is affected by growing data volume depends on the required data access patterns, available indexes and clustering of the accessed data. Data volume growth may affect response times of 19 out of the 22 analytical queries [4], as they access tables whose cardinality increases during the course of a benchmark run. Response times of queries Q2, Q11 and Q16 should not be affected by growing data volume, as they access only tables whose cardinality does not change during the course of a benchmark run. Also Q22 should not be affected too much, if a suitable index on the ORDER table is available.

5.2.2 Analytical Model and Normalization

We propose to monitor data volume growth during a benchmark run and to normalize response times based on an analytical model to compensate the "handicap" caused by larger data volumes. In order to minimize the performance impact of monitoring data volume growth, we propose to monitor the number of non-aborted New-Order and Delivery transactions and to estimate the data volume growth of the ORDER, ORDER-LINE and the NEW-ORDER relation based on this figure. CH-benCHmark, like any other TPC-C implementation, has to monitor the number of New-Order transactions anyway for reporting the throughput metric. Additionally, we have to monitor how many New-Order transactions abort due to simulated input errors.

For each analytic query, the analytical model has to capture how data volume growth affects query response times, e.g. based on the accessed relations and the complexity of required basic operations, like scan, join, etc. Currently our model focuses on the growth of the accessed relations. For a given point in time, cardinalities of accessed relations can be estimated and the analytical model can be used to determine a compensation factor. This factor can be used to normalize query response times and thereby compensate the "handicap" caused by larger data volumes.

The point in time used for estimating data volume of a given query execution depends on the configured isolation level. Depending on the chosen isolation level, a query may even account for data which is added while the query is executed. For example, in Figure 5.4 data volume grows even during query execution as query execution starts at t_s and ends at t_e. Response times of queries may increase over

[4] The SQL code of the CH-benCHmark queries can be found in the Appendix.

time when more data has to be processed. For snapshot isolation and higher isolation levels, the start time of query execution (t_s) can be used. For lower isolation levels, t_s would ignore cardinality changes during query execution and t_e could favor longer execution times. As a compromise, the middle of query execution (t_m) may be used.

5.3 Experimental Evaluation

For our experimental evaluation[5], we configured CH-benCHmark as follows. We use 12 warehouses, as we want to analyze an in-memory scenario. This corresponds to a data volume of around 1 GB and comes close to the minimum database population of TPC-H. We use a single analytical session and no transactional sessions. For our read-only workload, we require that queries are performed on consistent snapshots of the database and use snapshot isolation as isolation level. This isolation level has to be explicitly configured in system "P" and is the default for system "V", which is optimized for read-mostly workloads. We conducted our experiments on a commodity server with two Intel X5570 Quad-Core-CPUs with 8MB cache each and 64GB RAM. The machine had 16 2.5" SAS disks with 300GB that were configured as RAID 5 with two logical devices. As operating system, we used an Enterprise-grade Linux running a 2.6 Linux kernel.

In order to evaluate how data volume growth affects response times of analytical queries, we need the ability to evaluate analytical performance on CH-benCHmark databases of different sizes. For reproducibility, we always use the same fixed data set for a given data volume size. We generate the data sets by configuring CH-benCHmark for a purely transactional workload scenario and dump database contents to disk after a given number of New-Order transactions have been performed. As described in Section 5.2.1, normalization is based on an analytical model and an estimation of data volume growth. We estimate data volume growth based on the characteristics of typical TPC-C implementations (11% of New-Orders remain undelivered). Figure 5.5 shows a comparison between the estimated cardinalities and the actual cardinalities of the relations in the used data sets. This comparison shows

[5]As part of his bachelor thesis, Adrian Streitz helped integrating the benchmark extensions for normalized metrics into the "Performance Measurement Framework", an implementation of the CH-benCHmark, and helped with conducting measurements. Florian Funke and I (Michael Seibold) were advisors for his bachelor thesis "Architecture and Evaluation of the Mixed-Workload CH-benCHmark".

that the estimations are quire accurate. The x-axis represents the factor by which data volume is increased (2x - 64x) in a given data set and the y-axis represents actual or estimated cardinality of the relations. 1x denotes the initial database size with 12 warehouses, 2x denotes twice that amount and so on.

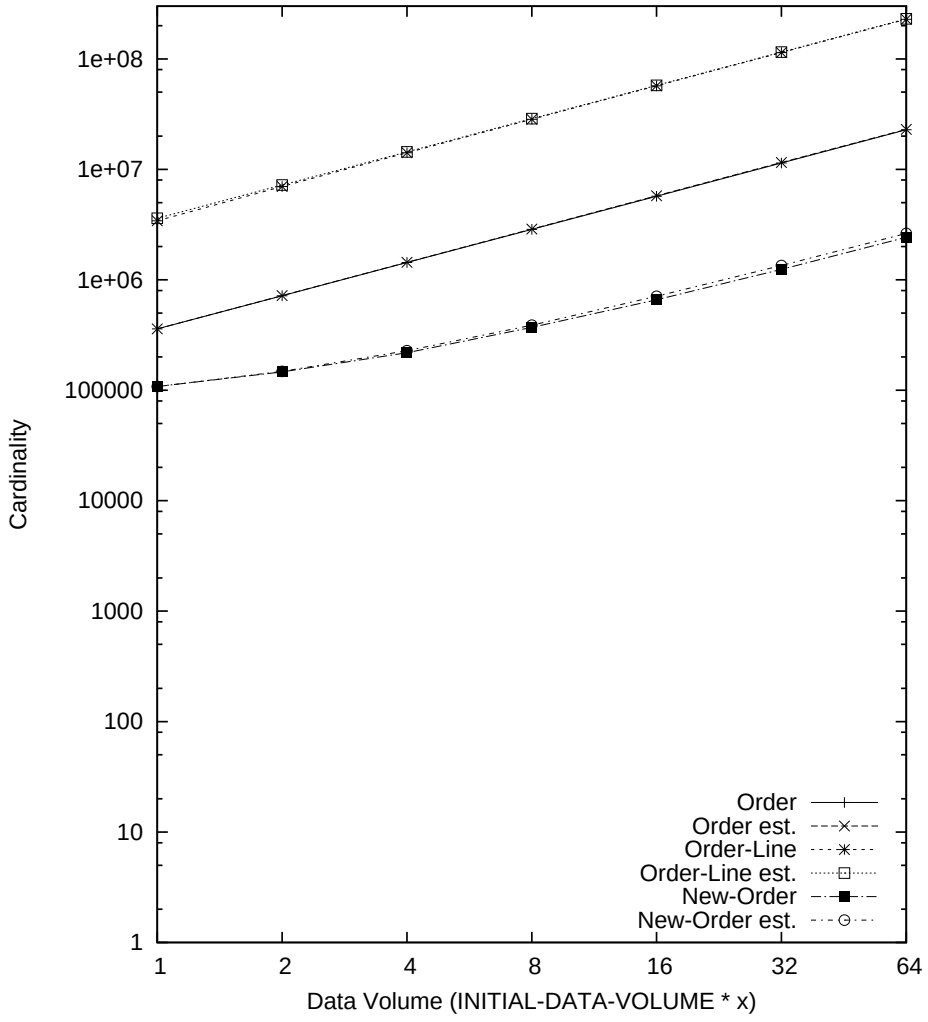


Figure 5.5: Estimated and Actual Cardinalities of Relations

In order to compare purely analytical performance of different database systems, we load data sets of different sizes into two database systems which are representative for different database system architectures. Database system "P" is a general-purpose disk-based database system and adheres to traditional row-store architecture. Database system "V" adheres to emerging column-store architecture, is highly optimized for analytical loads and represents a main-memory database system. We measure response times of analytical queries for each data set. Each query is per-

formed N times on each data set with warm cache and the average response time is compared. Figure 5.8 shows the measured average response times for each of the 22 queries on system "P" and Figure 5.9 shows the corresponding normalized response times. Figures 5.6 and 5.7 show the same for system "V". Average response times for system "V" are one to two orders of magnitude better than for system "P". A difference of two orders of magnitude was expected due to the architectural differences of the two systems. Probably the difference is not always two orders of magnitude, because the data volume fits into the memory-resident buffer pool of system "P". As expected, average response times of queries Q2, Q11 and Q16 are not affected by growing data volume, as they access only tables whose cardinality does not change. After normalization, the average response times of all queries, apart from Q7 for system "P", are more or less constant. Q7 is an outlier. For system "V", average response time increases with growing data volume, but for system "P" it remains more or less constant. Thus, for system "V" normalization works fine, but for system "P" it results in decreasing response times. Probably system "V" scans a lot of data, while system "P" can reduce the processed data volume by applying early filtering techniques.

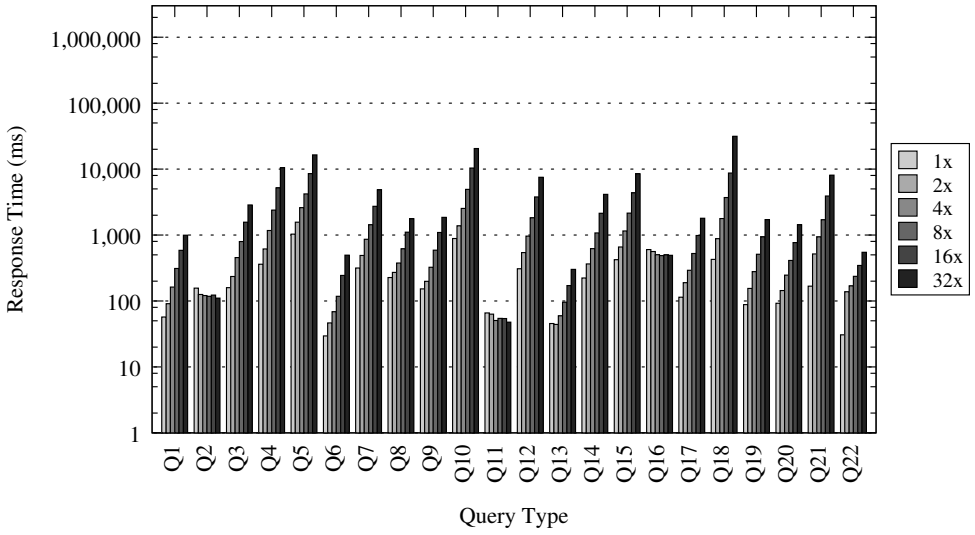


Figure 5.6: Average Response Times of System "V"

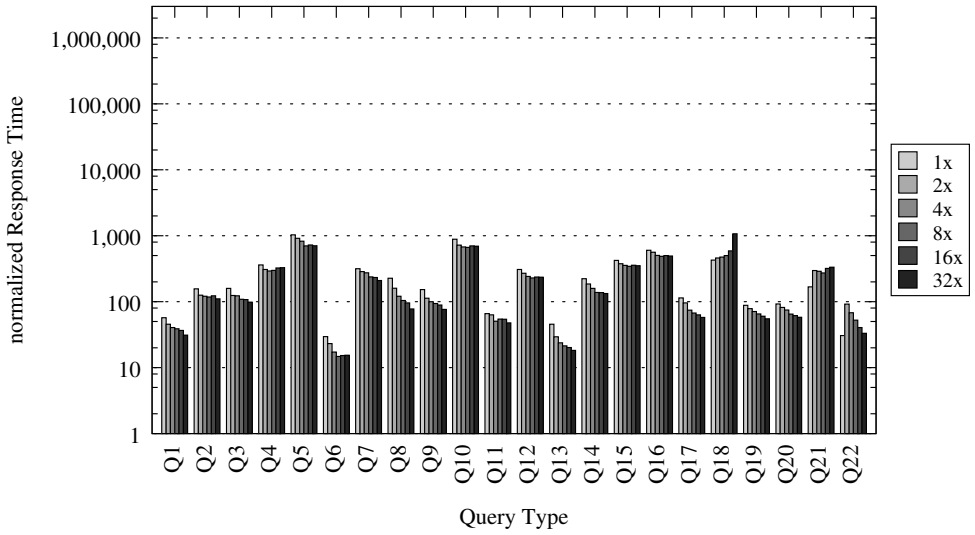


Figure 5.7: Normalized Average Response Times of System "V"

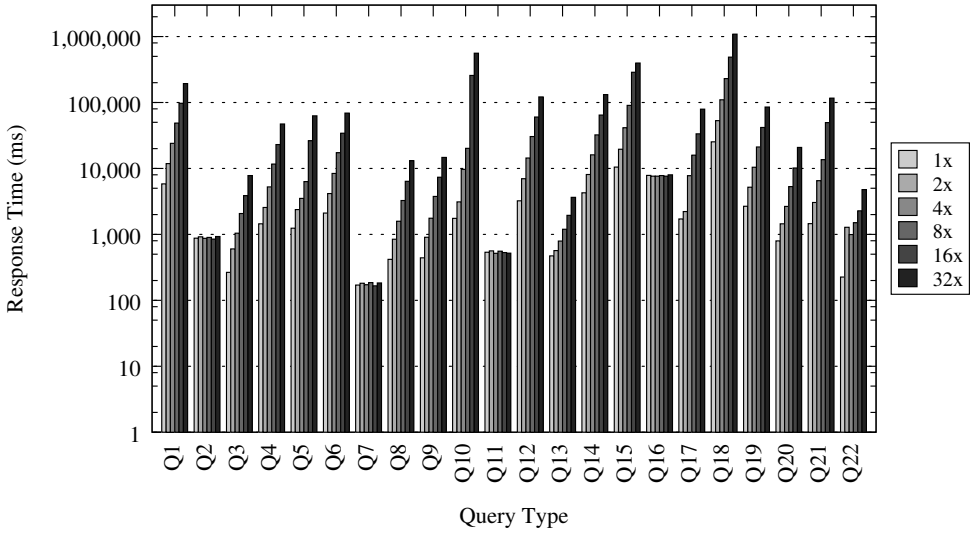


Figure 5.8: Average Response Times of System "P"

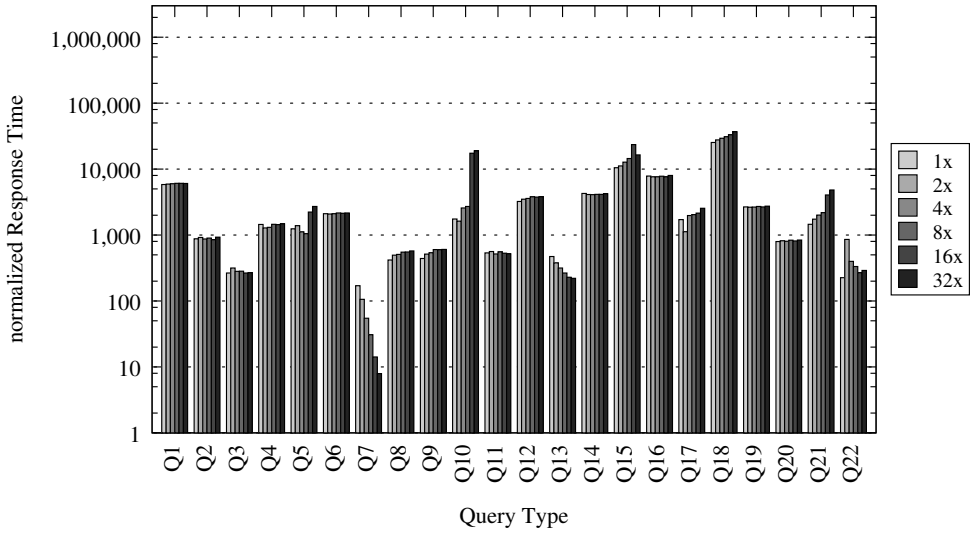


Figure 5.9: Normalized Average Response Times of System "P"

	1 Q. stream on data set with increasing data volume			
	System "V"		System "P"	
Q#	average response times (ms)	average normalized response times	average response times (ms)	average normalized response times
Q1	368	42	63632	6001
Q2	126	126	890	890
Q3	1012	121	2606	281
Q4	3368	319	15202	1403
Q5	5722	818	17104	1624
Q6	168	20	22549	2125
Q7	1782	260	177	65
Q8	729	131	4270	517
Q9	703	105	4829	551
Q10	6760	725	141634	7510
Q11	57	57	538	538
Q12	2497	254	39550	3619
Q13	120	27	1437	314
Q14	1430	163	42819	4165
Q15	2876	368	141051	14804
Q16	525	525	7767	7767
Q17	651	79	23405	1926
Q18	7828	587	332413	30598
Q19	614	70	27786	2683
Q20	517	73	6863	819
Q21	2552	280	31812	2710
Q22	245	53	1847	396
Geometric mean (ms)	146		1621	
Normalized geometric mean	859		10814	
Duration per query set (s)	41		931	
Normalized duration per query set	6		92	
Queries per hour (QphH)	1949		86	
Normalized queries per hour (QphH)	15222		868	

Table 5.2: Reported CH-benCHmark Results

The experiments show that data volume growth affects response times of analytic queries and that our analytical model can be used to normalize these response times for CH-benCHmark. Normalized response times can serve as a performance metric for the analytical load and account for data volume growth. This metric can be used to compare systems whose data volume grows at different rates.

In [30] we defined a tabular format for reporting CH-benCHmark results. In Table 5.2 we show our results according to this format and have added columns and rows for the proposed normalized metrics. The normalized metrics are determined in the same way as the original ones, but with normalized response times. The presented results do not correspond to a full mixed workload, as only one query stream is

performed on fixed data sets with increasing data volume. The advantage of this approach is, that the same data set of given size can be used for both systems. We decided not to compare the two systems with a full mixed workload, because system "V" is highly optimized for analytical loads and is not intended for transaction processing.

5.4 Deviations from TPC-C and TPC-H Specifications

In the following we provide a short summary of those aspects in which CH-benCHmark deviates from TPC-C and TPC-H specifications.

The transactional load of CH-benCHmark deviates from the TPC-C specification in the following aspects. First, client requests are generated directly by transactional sessions instead of simulating terminals and the number of transactional sessions is a parameter of the CH-benCHmark. Second, home warehouses of business transactions are randomly chosen by each transactional session and are uniformly distributed across warehouses, instead of statically assigning home warehouses to terminals. Third, a transactional session issues randomly chosen business transactions in a sequential manner without think times or keying times. But, the distribution of the different business transaction types follows the official TPC-C specification. Fourth, the number of warehouses is a parameter of the CH-benCHmark and it is not necessary to increase the number of warehouses to achieve higher throughput rates. These changes can be easily applied to existing TPC-C implementations, only small modifications of the benchmark driver configuration and implementation may be required.

The analytical load of CH-benCHmark deviates from the TPC-H specification in the following aspects. First, the queries are reformulated to match the extended TPC-C schema. Second, the queries are performed on extended TPC-C data which may have different characteristics than the original TPC-H data. Third, the TPC-H refresh functions are omitted, as the database is continuously updated (and expanded) via the transactional load. Fourth, default values are used for substitution parameters and these values do not change across query executions. Default values have been chosen such that data from the initial database population and data that was generated during the benchmark run is selected. Fifth, CH-benCHmark does not analyze singe-user and multi-user workload separately in each benchmark run like Power and Throughput Test, but different benchmark configurations may be used for that purpose.

A specification of CH-benCHmark has to define the additional tables which have to be added to an existing implementation of TPC-C. Furthermore, the TPC-H-like queries, the scaling model and the performance metrics have to be specified.

5.5 Conclusions

In this chapter we presented the mixed workload CH-benCHmark and analyzed its workload characteristics and performance metrics. Based on this analysis, we proposed normalized performance metrics that account for data volume growth. Thereby, we tackled the problem that higher transactional throughput may result in larger data volume which in turn may result in inferior analytical performance numbers. The reason why we need the proposed normalized performance metrics is not data volume growth itself, but the fact that data volume growth varies largely between different systems under test that support different transactional throughput rates. An alternative approach would be not to measure peak transactional and analytical performance, but to measure how much analytical throughput can be achieved while a fixed transactional throughput rate is fulfilled. We opted for this approach in the experimental evaluation of MobiDB (see Section 4.5.4). However, to be able to compare results with other systems, the same amount of fixed transactional throughput needs to be used. Moreover, a mixed workload benchmark could measure resource requirements or energy consumption while maintaining fixed transactional and analytical performance.[6]

[6] A similar approach has been proposed by Ashraf Aboulnaga, Awny Al-Omari, Shivnath Babu, Robert J. Chansler, Hakan Hacigümus, Rao Kakarlamudi and Michael Seibold at Dagstuhl seminar 12282 "Database Workload Management", as part of a proposal for a workload management benchmark. The key idea is to compare systems by total cost for processing a given workload and to derive the total cost of a system from resource requirements, energy consumption and penalty costs for not meeting SLAs/SLOs.

Chapter 6

Elasticity for Mixed Workloads

Emerging main-memory database management systems, like HyPer [64] or SAP HANA [44], process business logic inside the DBMS and achieve low response times at extremely high throughput rates. HyPer achieves outstanding OLTP and OLAP performance numbers and supports mixed workloads consisting of interaction-free transactions (OLTP) and read-only analytical queries (OLAP) on the same data, based on a low-overhead snapshot mechanism. SAP HANA also supports the efficient processing of both transactional and analytical workloads on the same database by using different storage formats during the life cycle of a record, according to Sikka et al. [99]. Future cloud business applications could be built on top of such systems in order to support Operational Business Intelligence efficiently. These emerging main-memory DBMSs are designed for multi-core servers with huge amounts of main-memory. However, the resource utilization of such database servers is typically low, as they are sized for peak loads and often average load is much lower. Today, multi-tenancy techniques are used to improve resource utilization in cloud computing scenarios. But huge main-memory requirements of single tenants may preclude application of multi-tenancy techniques for emerging main-memory DBMSs.

In this chapter, we present an approach for improving the resource utilization of emerging main-memory DBMSs even if there are no complementary database workloads, by temporarily running other applications on the database server using virtual machines. Furthermore, we present an analytical model to derive attractive service level objectives that can be met by a main-memory DBMS despite being co-located with arbitrary applications running in VMs.[1]

[1]Parts of this work have been published at CLOUD 2012 [96]. Furthermore, parts of this work

6.1 Challenges

Emerging main-memory database systems are designed for multi-core servers with huge amounts of main-memory. Today these systems are typically deployed on dedicated bare metal servers (physical machines, PMs), as illustrated in Figure 6.1 (a). However, resource utilization is often low, because database servers are typically provisioned for peak loads and often average load is much lower. Especially for mixed workloads, as the OLAP load typically increases before the end of a quarter and then often drops significantly, but database servers must be sized for peak OLAP load. Today, multi-tenancy techniques are used to improve resource utilization in cloud computing scenarios, as discussed in Chapter 2.2. The best example is salesforce.com [116]. Furthermore, Soror et al. [101] have shown that database systems with complementary resource requirements can be consolidated and advanced multi-tenancy techniques can be employed to further reduce space and performance overhead of consolidation as discussed by Aulbach et al. [8]. But huge main-memory requirements of single tenants may preclude application of multi-tenancy techniques for emerging main-memory database systems, as most or even all data of co-located tenants has to be kept in main-memory.

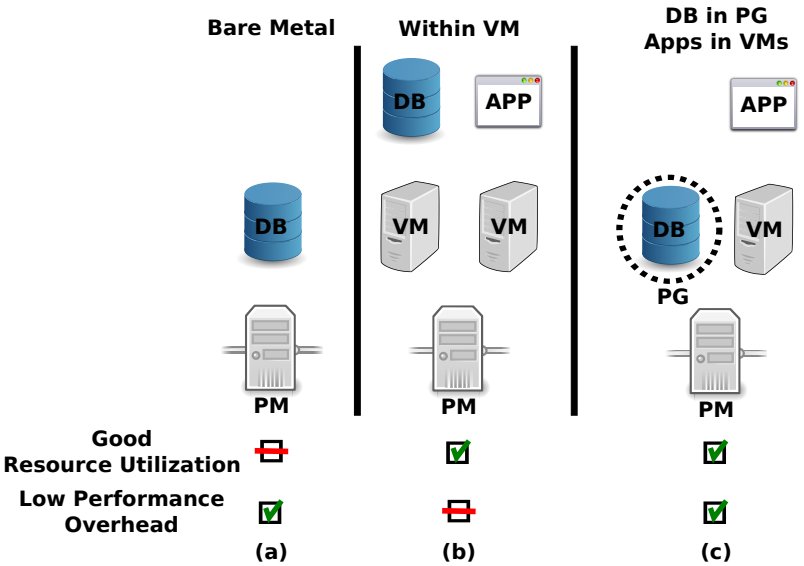


Figure 6.1: Deployment Approach

have been submitted to ICDE 2013 [94].

Virtualization and server consolidation can be employed to improve resource utilization and reduce operational costs, as discussed by Speitkamp and Bichler [103]. Even if there are no complementary database workloads, the database system and other applications can be deployed onto the same server using virtual machines (VMs), as illustrated in Figure 6.1 (b). But unfortunately, running emerging main-memory database systems within VMs causes huge overhead, because these systems are highly optimized to get the most out of bare metal servers. In a recent talk[2], Prof. Thomas Neumann demonstrated that virtualization may reduce TPC-C throughput by 33%. But running these systems on bare metal servers results in low resource utilization, as discussed above. Thus, both deployment approaches presented so far have significant disadvantages. Instead, we propose to deploy emerging main-memory database systems within light-weight containers (process control groups, PGs) and to run arbitrary other applications alongside these containers using VMs. This approach allows to improve resource utilization by temporarily running other applications on the database server without causing too much performance overhead for the database system. The servers on which these VMs would normally run can be suspended, to save energy costs. As shown by Graubner et al. [52], power consumption of data centers can be reduced by consolidating applications running in VMs onto fewer servers and powering off the other servers. Figure 6.1 (c) illustrates this deployment approach. This deployment approach is made possible by a combination of novel technologies supported by the Linux kernel, namely Linux Control Groups and Linux Kernel-based Virtual Machine (Linux KVM).[3]

Emerging main-memory DBMSs like HyPer and SAP HANA [44] allow to process business logic inside the DBMS. Thereby performance may be improved significantly, as the number of round-trips between application servers and the DBMS can be reduced. HyPer minimizes synchronization overhead by processing transactions sequentially according to the one-thread-per-core model. As transactions are processed sequentially, incoming transactions have to wait until preceding transactions have been processed. For high and bursty arrival rates, execution times of individual

[2]The talk with the title "Scalability OR Virtualization" took place on November 18th 2011 at "Herbsttreffen der GI-Fachgruppe Datenbanksysteme" in Potsdam, Germany. The talk was in German, but the slides are in English. A video recording is available online: `http://www.tele-task.de/archive/series/overview/874/` (retrieved 08/28/2012).

[3]We introduced this concept at CLOUD 2012 in our paper "Efficient Deployment of Main-memory DBMS in Virtualized Data Centers" [96] and received the Best Student Paper Award.

transactions thus have to be very low in order to achieve low response times. This is achieved by keeping all data in main-memory and minimizing synchronization overhead by processing transactions sequentially. With this approach low response times and extremely high throughput rates can be achieved. But we need to ensure that the main-memory allocation is never reduced below a certain lower bound so that the whole dataset fits into main-memory and swapping is avoided. This requirement needs to be considered when consolidating VMs onto the main-memory database server. The lower bound depends on two factors: The amount of memory needed for the actual data and the main-memory demand for processing business transactions. The former is typically large, but can be reduced by employing compression techniques. According to Hasso Plattner [88], compression rates of factor 20 can be achieved for typical customer data using column-store technology. The latter is typically small, as only one transaction per core is processed concurrently in the one-thread-per-core model. If main-memory allocation would be reduced below this lower bound during runtime, severe interferences are highly probable. The operating system will start to swap data from main-memory to disk and the execution time of a single transaction would increase dramatically if it touches data that is not available in main-memory. Thus, pure OLTP workloads do not leave much room for improvement.

But HyPer supports mixed workloads consisting of interaction-free[4] transactions (OLTP) and read-only analytical queries (OLAP) on the same data and thereby enables business applications with operational (or real-time) business intelligence features. For OLTP, we need to ensure that main-memory allocation is not reduced below a certain lower bound, as discussed above. For mixed workloads, this lower bound is a bit higher, because additional main-memory is required for snapshots of the data, as HyPer processes read-only analytical queries in parallel on snapshots of the data. But HyPer uses special techniques for minimizing the main-memory and processing overhead of these snapshots[5]. Apart from the snapshots, analytical queries require memory for intermediate results, which may be substantially large, e.g. queries involving large join operations or other pipeline-breakers that require to

[4]HyPer assumes that there is no user interaction during processing of single transactions, which is common for high throughput OLTP engines.

[5]The size of the snapshots depends on main-memory access patterns, as we discussed in Section 4.5.4 and [95], and can be minimized by clustering the current data, that is still modified, as Henrik Mühe et al. proposed in [80].

materialize large intermediate results. If arrival rates are high, it may be necessary to keep these intermediate results in main-memory in order to achieve required throughput rates. But if query arrival rates are low, intermediate results could also be stored on disk, if expected response times are large enough to allow for the required disk I/O. For varying OLAP loads, it depends on the actual load if a lot of main-memory is required for processing lots of queries in main-memory or if there are only few queries that can be processed on disk. If the *GlobalController* would know about the current OLAP load situation and how it will change in the near future, main-memory allocation could be changed accordingly. But if the *GlobalController* changes main-memory allocation without this knowledge, analytical queries may miss their SLOs.

6.1.1 Resource Allocation Changes at Runtime

We analyzed how much execution time deteriorates due to OS-swapping, when the main-memory assignment is reduced while a query is executed. We used HyPer and our 1TB-server described in Section 6.4 to analyze the execution time for joining order and order-line tables from the CH-benCHmark presented in Chapter 5 with 500 warehouses. HyPer allows to specify how much main-memory may be used for a join query and we used Linux Control Groups to enforce further limits on resource usage, including OS disk caches. Initially, we assigned sufficient main-memory to keep intermediate results completely in main-memory. When the main-memory assignment was reduced by 10% shortly after query execution started, query execution did not terminate even after we waited several hours and we decided to abort the query. But even if we took the normal execution time as a baseline and reduced the main-memory assignment by 10% one second before the end of baseline execution time, the execution time still increased by a factor of 7. Figure 6.2 shows that execution time grows when the main-memory assignment is further reduced, again one second before the end of baseline execution time. At a memory reduction of 50% the execution time increased by a factor of 65. The reductions in execution time also show the effectiveness of Linux Control Groups. The conclusions we draw from these experiments is that knowledge about how much main-memory is available to the DBMS is very important to choose the physical operators and their parameters right in order to utilize all available main-memory and to avoid OS-swapping.

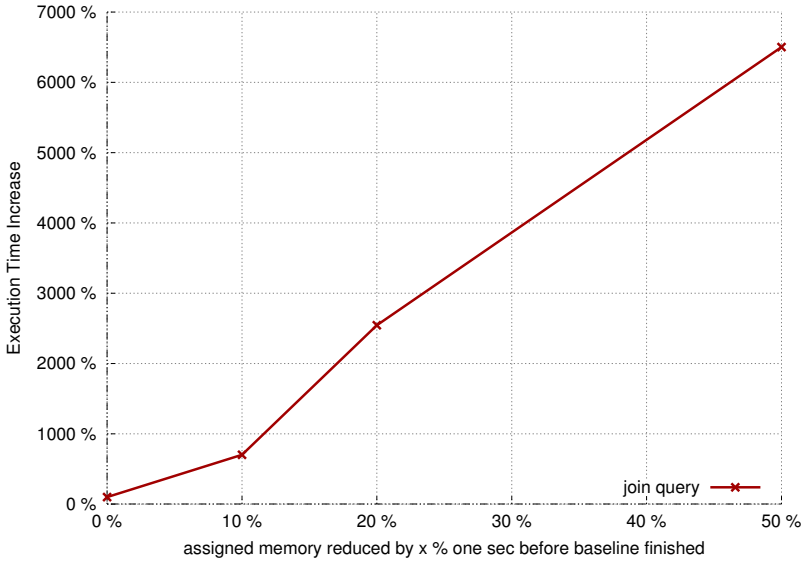


Figure 6.2: Effects of Reducing Main-Memory Assignment

Apart from main-memory, processing resources are also very important for query processing in main-memory DBMSs. With modern virtualization techniques, the CPU shares and the number of virtual cores of a virtual machine can be changed at runtime. Providing an application with more main-memory can only result in improved performance if the larger data volumes in main-memory can be processed efficiently with the available CPU cores and CPU shares. Today, multi-core CPU architectures are common and DBMSs need to process queries in parallel in order to utilize these architectures efficiently. Kim et al. [66] and Blanas et al. [16] investigate the parallelization of the probably most important DBMS operator, the join of two relations. They investigate different ways of adapting sort-based as well as hash-based join algorithms to modern multi-core environments and show that high performance gains can be achieved. However, join operators not only have to exploit all resources available at the time they were instantiated, but they should be able to adaptively adjust the numbers of processing threads to changing resource allocations, including number of virtual CPU cores. Otherwise additional resources would be idle until potentially long-running queries are processed completely or new queries arrive. If the number of CPU cores is reduced, but the join operator does not reduce its number of processing threads, performance may decrease, as there would be several threads per CPU core that are all ready to run, because there is no disk I/O. On the one hand, context-switching would cause a certain overhead and on the other hand, parallelizing into more threads than necessary is counter-productive due to suboptimal speedup.

In the following, we argue that emerging main-memory database systems, should support changes to resource allocation at runtime in order to improve resource utilization and reduce operational costs. To make spare resources on the database server available for hosting other applications (running in VMs), resource allocation of the DBMS has to be adapted at runtime to the current load situation and the load situation of the near-future. But current commercial DBMSs do not handle dynamic changes to resource allocation well. We have observed that resource assignments should not be reduced without taking precautions and that dynamically added resources are often not used right away. First, if resource allocation of a virtual machine is reduced dynamically without reconfiguring the DBMS inside the VM first, performance may deteriorate disproportionately. For example, dynamic reduction of main-memory allocation may have severe performance impacts caused by virtual memory swapping of the operating system running inside the virtual machine (OS-swapping), as we showed in [96]. Second, dynamically added resources are often not used efficiently. For example, additional processing resources are idle due to I/O stalls and additional main-memory would be required to utilize the additional processing resources. But for using additional main-memory resources, changes to database configuration parameters are necessary that take a while to become effective and even may require a restart of the database system (e.g. TimesTen [108]). Furthermore, long-running queries that are already in-progress when resource allocation changes may not be able to use the additional resources, because query operators were instantiated with fixed numbers of processing threads and memory reservations when the query was started. In the following, we focus on emerging main-memory database systems, like HyPer, that are more flexible with regard to resource allocation changes.

For taking resource allocation decisions, accurate monitoring data is required. But it is difficult to measure the true resource requirements of a DBMS, as these systems are typically designed to use all the resources they are given. Furthermore, local adaptive control within the DBMS (e.g. request queuing) makes it difficult to monitor the actual load from the outside and to predict resource requirements of the near future. For example, database systems with adaptive workload management have a scheduler component which limits the number of requests processed simultaneously. Thus the utilization of resources does not reflect requests that are waiting in queue, as resources would still be utilized normally even if the queue

grows due to spikes in the arrival rate. Recently, Curino et al. [34] proposed a technique called "buffer pool gauging" to estimate the main-memory requirements of a traditional disk-based DBMS more accurately. This approach does not follow the common "black box" monitoring approach, as a probe table is created in the database and disk access statistics provided by the DBMS are used. We also depart from the "black box" monitoring approach, but focus on emerging main-memory DBMSs and propose the following cooperative approach: The DBMS communicates its resource demand, gets informed about currently assigned resources and adapts its resource usage accordingly. We assume that main-memory DBMSs can estimate their current resource demand and their resource demand for the near future quite accurately, as the DBMS knows about currently processed requests and request waiting in queue. However, when spare resources are actually used by other applications (running in VMs), DBMS performance may be affected. SLOs defined in SLAs may be missed, as there is a certain delay until resources used by VMs can be given back to the DBMS. For example it may be necessary to migrate VMs away to other servers or to freeze their state and store them on disks of the database server. We propose an analytical model to derive attractive SLOs which can be met by a main-memory DBMS that is co-located with arbitrary applications (running in VMs) and uses the proposed cooperative approach. This analytical model and the cooperative approach form the basis of Elastic Workload Management, which ensures that the database system meets SLOs of mixed workloads despite co-location of VMs. There is related work by Lang et al. [71] that discusses SLO-based hardware provisioning in the context of multi-tenant database systems. Lang et al. claim to be the first to study cost-optimizations for multi-tenant performance SLOs in a cloud environment (Database-as-a-Service). We also focus on SLOs covering performance metrics like response time and throughput. But we focus on main-memory DBMSs, like HyPer [64] or SAP HANA [44], and propose to consolidate the DBMS with arbitrary applications running in VMs instead of traditional multi-tenancy.

6.1.2 Elasticity

Today, elasticity typically means that customers of Infrastructure-as-a-Service providers, like Amazon EC2, can add virtual machines on demand, e.g. via a Web-Service interface. Beyond that, modern virtualization technology allows to migrate virtual machines between the servers of a server farm. Figure 6.3 shows a server farm con-

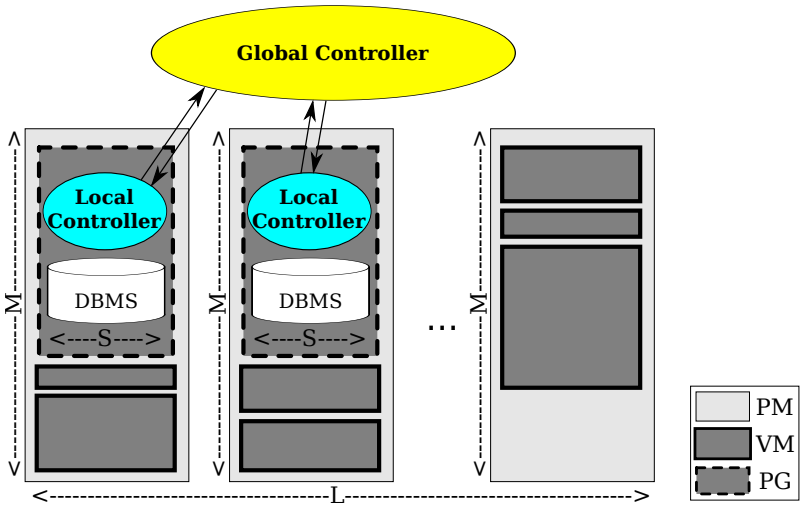


Figure 6.3: Elasticity in the Large (L), Medium (M) and Small (S)

sisting of several servers – or physical machines (PMs) —, that run virtual machines (VMs) of different sizes, representing different resource requirements. Consolidation is achieved by assigning several virtual machines to the same physical machine. The allocation of VMs and the migration of VMs between servers of the server farm is controlled by a *GlobalController*. We refer to this as elasticity in the large. To our knowledge, major service providers do not use VM live migration in production yet although it is supported by well-known virtualization solutions. Instead remaining capacity of active servers is sold at lower prices based on supply and demand, e.g. Amazon EC2 Spot Instances. But VM live migration may make resource allocation more flexible in the near future. Furthermore, modern virtualization technology allows to change the resource allocation of virtual machines dynamically, i.e. the ability to change the amount of assigned main-memory, the amount of CPU shares and the number of virtual CPU cores while the virtual machine is running. We refer to this as elasticity in the medium. There already are products, like vmware vCenter Operations Management Suite, [6] that allow to control the sizing of virtual machines globally and to change the resource allocation of virtual machines dynamically. To our knowledge, major service providers do not allow customers to change the resource allocation of virtual machines at runtime. Instead a small number of instance types are offered, that correspond to a defined amount of resources, and virtual machines have to be restarted to change the instance type.

[6] `http://www.vmware.com/products/datacenter-virtualization/vcenter-operations-management/overview.html` (retrieved 08/28/2012)

Today IaaS providers promote elastic scale-out of applications across several VMs, what typically requires special frameworks for distributed application development and involves changes to application code. In contrast, elasticity in the medium enables elastic scale-up — only limited by the resources of the underlying physical server, which are substantial for modern multi-core servers — and allows to adapt resource allocation dynamically to changing demands within short time intervals. But, applications running inside virtual machines have to be aware of elasticity in the medium to communicate resource requirements and make the most of resources that are dynamically allocated at runtime. We refer to this as elasticity in the small. Elasticity in the small requires some effort for the IaaS customer, but IaaS providers could support elasticity-aware applications and normal VMs on the same infrastructure. The benefits of improved resource utilization could be passed on in the form of lower service fees for elasticity-aware applications. Customers could be charged based on the amount of dynamically allocated resources according to the pay-as-you-go model that made IaaS so successful.

Shen et al. [97] propose to minimize resource provisioning costs in cloud computing scenarios by adapting resource allocation based on online resource demand prediction. They employ elasticity in the medium in the form of "resource capping" and also elasticity in the large in the form of migrating VMs based on predictions about conflicting resource requirements. But they do not discuss how the applications running inside VMs should adapt when resource limits are changed and do not assume any prior knowledge about the applications running inside VMs. In contrast, we address elasticity in the small with our cooperative approach and treat database systems in a special way. In virtualized data centers, all three forms of elasticity have to work together in order to improve resource utilization, to reduce the number of active servers and thereby reduce operational costs, including energy costs. Power consumption of data centers can be reduced by consolidating applications, including database systems, that run in virtual machines onto fewer servers and powering off the other servers, as shown by Graubner et al. [52]. Furthermore, Shen et al. [97] propose to save energy by dynamic CPU frequency scaling. Both techniques may benefit from our cooperative approach, as cooperation allows to allocate resources based on more accurate information about resource requirements and to utilize dynamically assigned resources more efficiently. It is undesirable to operate servers at a medium utilization level, as the power efficiency is not a linear function

of the server's load. The two most power efficient operating modes are suspend and full utilization [11]. Energy efficiency of data centers is an area of active research. Nishikawa et al. [82] consider energy efficiency in the context of several enterprise workloads, like OLTP and decision support, and propose considering application level behaviors to improve timing of power saving actions.

6.2 Elastic Workload Management

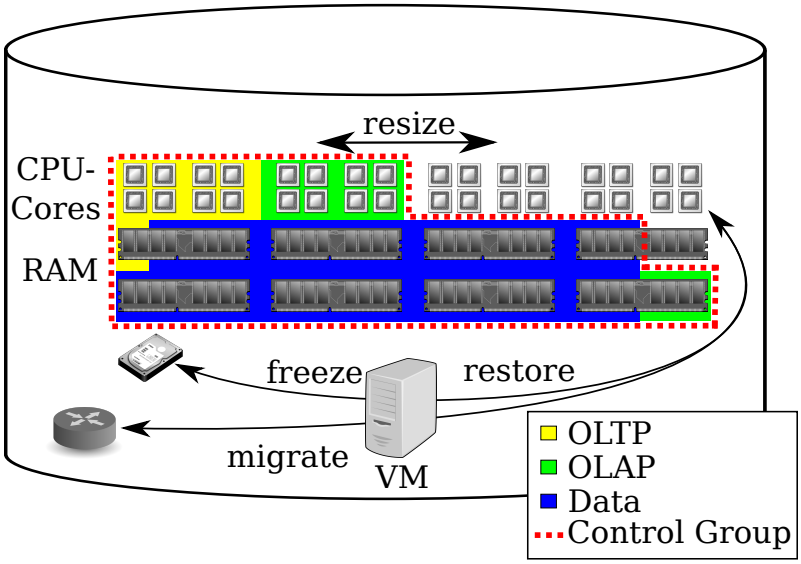


Figure 6.4: Elastic Workload Management

Figure 6.4 illustrates a main-memory database system that runs on a modern multi-core server with many CPU-cores and huge amount of main-memory. The database system needs resources for keeping the data in main-memory and for processing the mixed workload consisting of OLTP and OLAP. The resource allocation of the database system is "resized" according to the current load situation which changes over time, especially for OLAP. "Control Group" corresponds to the amount of resources that are currently assigned to the database system. The remaining resources can be used for running VMs on the database server. These VMs can either be migrated from other servers or restored from disks of the database server. When the database system needs more resources, it may be necessary to migrate VMs away to other servers or to freeze their state and store them on disks of the database server. The "Control Group" is realized using Linux Control Groups, that provide a lightweight form of process encapsulation and allow to limit resource usage of a group

of processes. On the one hand, resource limits of process group containers can be changed quickly. On the other hand, process group containers do not incorporate significant overhead, as all the processes are still running on the Linux Kernel of the hosting server. Moreover, Linux Control Groups can be combined with VMs on the same host using Linux KVM. As of now it is not possible to live migrate containers in the same way as VMs. However, main-memory DBMSs are poor candidates for live-migration anyway, due to high memory demand and data access frequency. We propose not to live migrate the main-memory DBMS, but other applications running in VMs.

6.2.1 Local Coordination

As shown in Figure 6.3, the physical resources of a server are divided between the VMs hosted on the same PM. We propose that each VM should host a *LocalController* that communicates with a *GlobalController* of the server farm to coordinate changes to resource allocation. On the one hand, the *GlobalController* dynamically adapts resource assignments to changing workload demands (elasticity in the medium) and migrates virtual machines between servers in order to reduce the number of active servers (elasticity in the large). It provides information on how much physical resources are currently assigned to a given VM and informs the *LocalController* running inside the VM when the allocation changes. On the other hand, the *LocalController* sends hints regarding future resource requirements to the *GlobalController* and controls how queries are processed within the DBMS (elasticity in the small) in order to make use of dynamically assigned resources. In order to give hints regarding future resource requirements, the *LocalController* queues incoming OLAP query requests and estimates how much resources are required to process them according to their SLOs. For the DBMS instances, we use process groups (PGs) instead of real VMs to avoid unnecessary virtualization overhead. All virtual machines could have local controllers, but in the following we focus on DBMSs.

Coordination allows the DBMS to avoid OS-swapping by starting queries with appropriate main-memory reservations, as the *LocalController* knows how much physical resources are currently available. Once resource allocation changes, the *LocalController* is informed. When resource allocation is reduced, we propose to abort longer-running queries and to restart them with different parameters in order to avoid OS-swapping. There is related work in the area of workload management by

Krompass et al. [68] on how to manage long-running queries. In contrast, we focus on how to prevent queries from taking unexpectedly long due to dynamic changes to resource allocation by restarting them right away when such changes occur. We focus on join operators, because joins are common database operations that often impact the runtime of query execution plans significantly and potentially require large intermediate results. When resource allocation is increased, it may make sense to abort longer-running queries that involve joins and to restart them with different parameters in order to reduce response times and improve throughput. For example, if a join query, that was started on-disk using an external join algorithm, could now be processed in-memory, overall execution time (including aborted partial execution) may be lower than on-disk execution time, as we showed in [96]. The *LocalController* has to decide whether to wait until running (join) queries finish or to abort and restart them. When resource allocation is reduced below the estimated resource requirements of running queries, queries should be restarted even shortly before completion, because execution times may increase strongly even when resource allocation is reduced only shortly before a query finishes as we showed in Section 6.1.1. When resource allocation is increased, the decision whether to restart a query or not should ideally consider the progress of the query. There is related work on progress indicators by Luo et al. [74] and recent work by Li et al. [73] that is complementary to our approach. For simplicity of our prototype implementation, we decided to restart running queries always when resource allocation changes.

In summary, local coordination allows resource allocation of main-memory DBMSs to be changed elastically, by communicating resource requirements (including lower bound on main-memory allocation), and may help to use dynamically assigned resources more efficiently.

6.2.2 Global Coordination

The *GlobalController* can make use of spare resources on database servers by migrating VMs onto the database servers. Coordination has to ensure, that SLOs for the mixed database workload are still meet although spare resources are used for other purposes. With Linux KVM and Linux Control Groups, processing resources can be used for other purposes and be given back to the DBMS with low overhead. It is just a matter of prioritization, changing process limits and process context switching, as VMs are operating system processes. But giving back main-memory

resources to the DBMS after using them for other purposes with a VM causes a certain delay, because the main-memory contents of the VM might still be needed and moving the VM onto another server with VM live migration requires to copy the memory contents over the network first. To avoid that such delays affect SLOs for OLTP, a certain amount of main-memory always has to be allocated to the main-memory DBMS and can never be used for other purposes. This represents the minimum memory requirement of the DBMS and is made up of the main-memory required for the operating system (OS), for keeping all data in main-memory (DATA), for snapshots of the data (SNAP) and for processing OLTP transactions. This minimum memory requirement has to be communicated to the *GlobalController* and has to be adjusted when data volume grows. The sweet spot is the main-memory required for intermediate results of OLAP queries. The physical resources are sized for peak OLAP loads and we assume that the average load is much lower.

Figure 6.5 illustrates the memory requirements of a main-memory DBMS, like HyPer, and two temporarily co-located VMs. We assume that around two thirds of the physical memory are reserved (for OS, DATA, SNAP and OLTP) and around one third of the physical memory can be allocated for OLAP when needed for processing analytical queries with large intermediate results. Initially main-memory for peak OLAP loads is assigned to the DBMS, at t_1 main-memory allocation of the DBMS is reduced and shortly afterwards *VM1* is migrated onto the database server. At t_2 main-memory allocation of the DBMS is further reduced and shortly afterwards *VM2* is migrated onto the database server. At t_3 main-memory allocation of the DBMS is slightly increased without problems, as sufficient spare memory is available. At t_4 main-memory allocation of the DBMS is further increased. Again there are no problems, as *VM2* has already finished its job and has terminated already. At t_5 there is an issue, the DBMS suddenly requires peak main-memory, but *VM1* has not jet finished its job. Therefore *VM1* has to be migrated to another server and there is a delay until sufficient resources have been returned to the DBMS (t_6). Delays due to VM migration may affect SLOs of OLAP queries, e.g. when resource requirements increase suddenly due to bursts in arrival rates and the delay causes too many OLAP queries to miss their response time goals by too far. We propose to ensure that SLOs are met by limiting the number of times that the resource allocation of the DBMS is reduced and propose an analytical model described in Section 6.3.

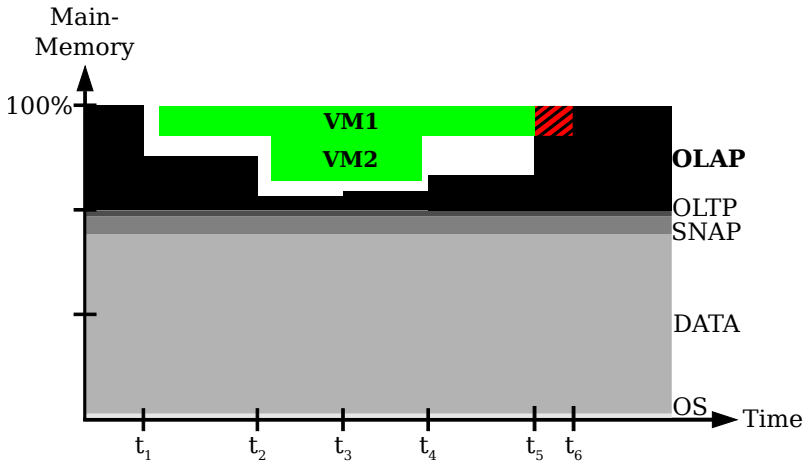


Figure 6.5: Coordination of Main-Memory Requirements

6.2.3 Elastic Scheduling

Query arrival rates are often bursty and commercial DBMSs without workload management extensions typically process incoming queries as fast as possible using all available resources in a best effort manner. This may lead to unnecessary spikes in resource consumption. In order to use spare resources with virtualization and server consolidation, such spikes are counter-productive, because they may lead to unnecessary VM migrations. With workload management techniques, incoming queries can be queued, and a scheduler can decide when to submit queries to the processing engine depending on execution time estimates and knowledge on response time and throughput goals (defined as SLOs in SLAs). There is related work by Chi et al. [28], to schedule queries from diverse customers in a cost-aware way, but they only consider response time goals. In contrast, we consider response time and throughput goals defined in compound SLOs (see Section 6.3). Workload management for traditional DBMSs is very difficult, because it is difficult to predict execution times accurately, as several queries are executed concurrently even on a single CPU core in order to mask delays caused by disk I/O. In contrast, main-memory DBMSs allow to estimate execution times much more accurately, as shown by Schaffner et al. [90]. There are no disk I/O delays as all data is kept in main-memory and in-memory queries are processed sequentially according to the one thread-per-core model to save context-switching overhead and to reduce main-memory requirements — only the intermediate results of one query per CPU core need to be kept in main-memory.

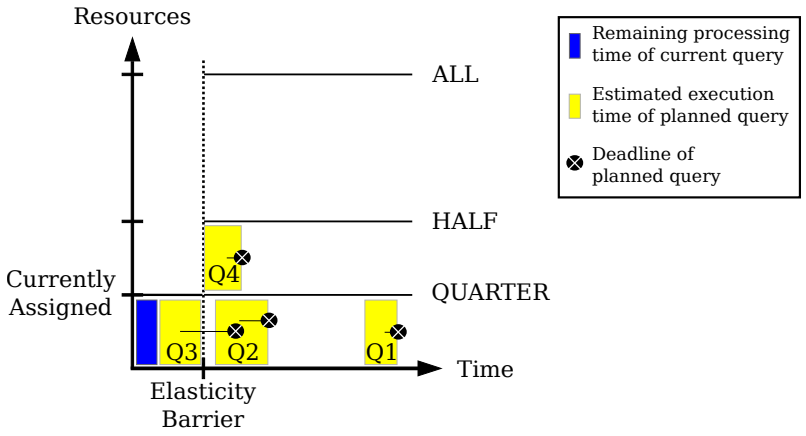


Figure 6.6: Elastic Scheduling

Instead of processing incoming queries as fast as possible, queries could be processed with as few resources as possible while still meeting SLOs. Instead of frequently requiring all resources for short time intervals, it would be better to use e.g. half the resources for a longer time interval. Thereby spare resources may be available for sufficiently long time intervals to co-locate VMs. We propose an elastic scheduling approach called GOMA that is designed for elastic scale-up (elasticity in the medium) and differentiates between two phases (see Figure 6.6). In the first phase only the currently assigned resources can be used, but in the second phase all physical resources can be used and those are always sufficient because the system is sized for peak loads. The *Elasticity Barrier* between the two phases is the time it takes to give all resources back to the DBMS by migrating VMs to other servers. GOMA tries to fit all queued queries on as few resources as possible. If the queued queries can be executed on-disk (with very little main-memory) while still meeting SLOs according to execution time estimates, GOMA does not use more resources and instead lets the *LocalController* signal the low resource requirements to the *GlobalController*. If the queued queries do not fit on disk, GOMA checks if a quarter (or half, etc.) of the main-memory resources are sufficient. If queries can be delayed such that their planned processing start time is in phase two and a quarter (or half, etc.) of the main-memory resources are sufficient for in-memory execution, the query will be planned as in-memory query. If the query has to be started in phase one to meet response time goals, the query is planned to be executed with the currently assigned resources and as soon as possible. If the queued queries do not fit, all resources are required and if they are not currently assigned, the system is overloaded. In any case, GOMA lets the *LocalController* signal the current resource

requirements to the *GlobalController*, based on an execution plan for the currently queued queries. Currently, GOMA takes a very coarse grain approach considering only quarter (or half, etc.) of the main-memory resources and only inter-query parallelization. GOMA does not consider intra-query parallelization and furthermore assumes that the peak load requires to execute queries with maximum memory requirements on all CPU cores in parallel. This leaves a lot of room for optimization, but it makes the proof-of-concept a lot easier to understand, because GOMA only has to consider the number of CPU cores for phase two, as there is always sufficient main-memory per CPU-core to perform any query.

Figure 6.6 illustrates an example. Q1, Q2 and Q4 are planned to be delayed as long as possible, Q3 has to be started in phase one to meet its deadline and is planned to be executed with the *Currently Assigned* resources. *HALF* of resources are sufficient and *QUARTER* would not be sufficient to meet deadlines for queued queries, thus *HALF* resources have to be requested and resource allocation has to be increased before the *Elasticity Barrier* is reached.

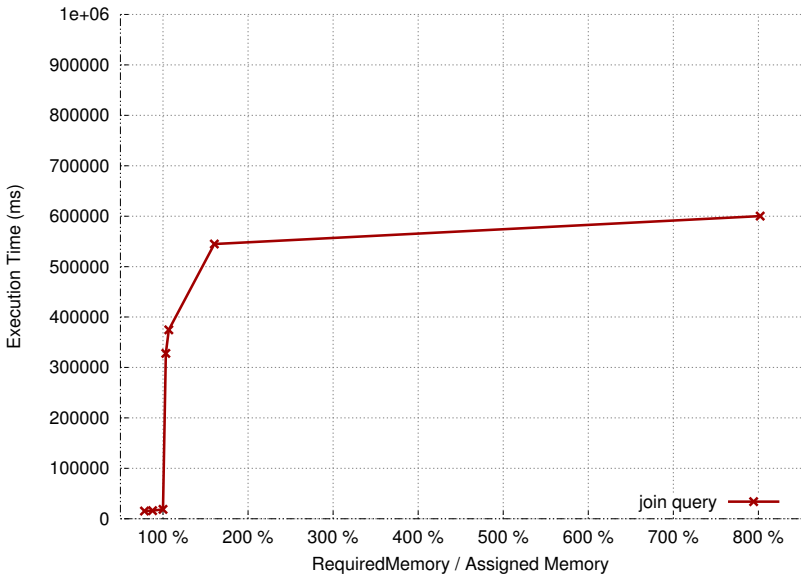


Figure 6.7: Intermediate Results In-Memory vs. On-Disk

We assume that there is sufficient memory to process all queries in main-memory according to SLOs and propose to process queries on disk (with only little main-memory) if we don't need to process them in-memory due to throughput requirements. The goal is to make more spare main-memory available for other purposes by employing virtualization and server consolidation. In both cases we need to es-

timate execution times. There is a big difference between in-memory and on-disk execution times. In order to demonstrate the difference between in-memory and external join, we performed the same join from CH-benCHmark as described in Section 6.1.1 on our 1TB-server described in Section 6.4 and varied the amount of available main-memory. Figure 6.7 shows that there is a huge spike in execution time if intermediate results do not fit completely in main-memory. Just after the 100% mark on the x-axis denoting the quotient of required memory and assigned memory, only little extra memory would be required to keep intermediate results completely in main-memory, but still execution time increases by a factor of 17 — from 19 seconds to 5 minutes. HyPer used the hash join algorithm in this experiment, which is a common join algorithm for main-memory DBMSs. The results suggest that join queries should be processed either completely in-memory or on-disk with only little main-memory, as more main-memory does not improve execution time much as long as the join cannot be processed completely in main-memory. There are other join algorithms, like hybrid hash join [41], that use additional main-memory more effectively and could level off the spike a bit. But in our case trading more main-memory consumption for lower execution time would only be interesting if the execution of on-disk hash join (with only little main-memory) takes too long for meeting SLOs. So far we opt for in-memory execution in this case. Hybrid hash join could allow on-disk processing for even shorter expected processing times and could be integrated with our approach. Patel et al. propose a quite accurate model for estimating execution times of hybrid hash join [87]. Furthermore, there are sort-merge based join algorithms that also settle for only little main-memory in turn of longer execution times. We plan to consider new developments like GJoin [51] as future work. GOMA enables cooperation between *LocalController* and *GlobalController*, allows to allocate resources based on more accurate information about resource requirements and the proposed elastic scheduling approach tries to make spare resource better usable for co-locating VMs.

6.2.4 Elastic Resource Allocation in the Cloud

The proposed approach is very suitable for cloud computing providers that apart from Infrastructure-as-a-Service (IaaS) also offer Database-as-a-Service (DbaaS). Elastic workload management allows to run main-memory DBMSs together with arbitrary VMs on the same host server and thereby enables consolidation across IaaS

and DbaaS. Software-as-a-Service (SaaS) offerings may be realized on-top of IaaS and DbaaS. For example, a SaaS business application, like CRM, may use DbaaS to manage the data and IaaS to run application servers. Consolidation across IaaS and DbaaS may improve resource utilization when main-memory DBMSs alone cannot be consolidated due to non-complementary workloads. Elastic workload management allows to consolidate them with arbitrary applications running in VMs. For cloud computing, it is important that customers agree with the service provider upon SLAs which define the characteristics of the provided service including SLOs, like maximum (max) response times and minimum (min) throughput rates, and define penalties if these objectives are not met by the service provider. There are two different ways of looking at SLOs. On the one hand, the consumer of a service needs a certain service level in order to be productive. For example, in telemarketing, there is a known number of agents that call prospective customers over the phone and have to enter gathered information into a customer relationship management (CRM) system during the call. In order to be productive in this scenario, end-to-end response times below one or few seconds are required for almost all requests and the required throughput depends on the number of agents. Furthermore, supervisors may use operational (or real-time) business intelligence features of the CRM product in order to monitor agent performance and to evaluate the success of marketing campaigns. For those requests, longer response times of one or few minutes may be OK and it may be acceptable if sometimes requests take several of minutes to complete. On the other hand, the provider of a service needs to know what is the best SLO that can be met by the service. This information is critical in order to decide what SLOs to include in the SLA of the provided service, especially if the accepted penalties are significant. For example, the provided service may be a CRM product with operational (or real-time) business intelligence features, that is offered according to the SaaS model and relies on a DbaaS offering based on an emerging main-memory DBMS. As business applications, like CRM, are very data-intensive, the SLOs of the CRM service mainly depend on the SLOs of the underlying DbaaS service. We focus on these SLOs and present an analytical model in the next section (Section 6.3).

6.3 Analytical Model

In this section, we describe an analytical model to derive an attractive SLO which can be met by a main-memory DBMS that is consolidated with VMs running arbitrary applications and uses elastic workload management.

6.3.1 Compound Service Level Objectives

SLOs often involve max response time (g_{rt}) and min throughput goals (g_{tp}) and are typically defined in SLAs. SLOs typically do not guarantee max response times for 100% of the requests, but only for a certain percentile (p) of requests, e.g. 99%. That means that in our example 1% of the requests could have longer response times. The actual number of requests that have to meet the max response time goal depends on the arrival rate, which captures how many requests — sent from consumers of the service — arrive at the provider of the service. The actual arrival rate may vary over time, but the service provider only has to be prepared for arrival rates below the min throughput goal. If more requests arrive, they can be dropped without affecting the SLOs. Thus, the highest arrival rate that the service provider has to be prepared for equals the min throughput goal. In the following we refer to this as the peak arrival rate (r_{a_p}). Arrival rates are typically bursty. The acceptable burstiness can be limited by defining the time interval length of an arrival window (w_a). The arrival rate and the arrival window length limit the number of requests that may arrive during any given arrival window. In the worst case, all requests arrive right at the beginning of the arrival window and still have to be processed according to response time goals. Thus, the arrival window should form part of the service level objective to define acceptable burstiness. Compound objectives can be defined to ensure, that those requests, which do not have to meet the max response time goal, are not delayed forever. For example, TPC-C requires that the average response time of all requests is below the percentile response time. Accordingly, we define compound service level objectives (cSLOs) which require, that the average response time of all requests (avg_{rt}) is below the percentile response time goal (g_{rt}). But the average response time may depend on the time interval length for which the average response time of processed requests is computed (measurement window, w_m). A fixed number of delayed requests can be delayed much longer if the measurement window is longer as there are more compensating requests that are not delayed.

Thus, the measurement window should form part of the service level objective. Figure 6.8 summarizes the components of the described compound SLOs. We define that the service provider has to fulfill the cSLO always, with the only exception of service unavailability. Put differently, the service is available if requests are processed according to the cSLO, else it is unavailable. Apart from that, the SLA has to define a service availability goal, e.g. 99.9% (or "three nines"), and penalties if the service is unavailable more often than the acceptable downtime, e.g. 8.76 hours per year. We assume that the service availability goal has to be met by the given server and possibly cold standby servers using master-slave replication. Hot or warm standby could allow for separation of work between the different servers, but require additional resources and therefore do not fit our resource consolidation scenario. Multi-master replication is an interesting direction for future work.

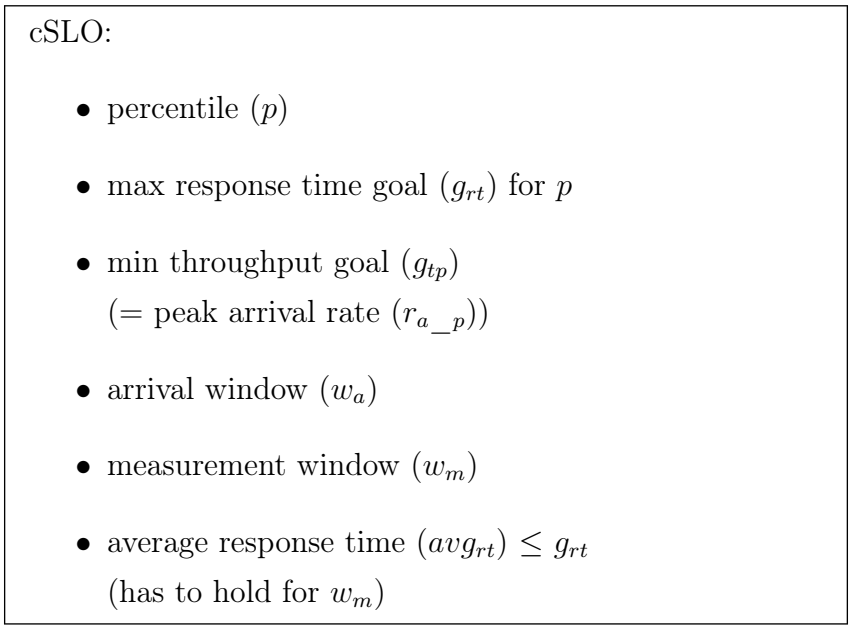


Figure 6.8: Compound Service Level Objective (cSLO)

6.3.2 Attractive cSLOs

We take the perspective of the service provider and try to find attractive cSLOs that can be met by a given service on a given physical server. In our scenario, the service is a main-memory DBMS, that processes mixed workloads consisting of transactions (OLTP) and analytical queries (OLAP), and the server is a commodity server with 32 CPU cores and 1 TB of main-memory. As discussed above, we assume

that OLTP transaction processing has very tight response time and throughput requirements. Therefore, we reserve a certain amount of resources for the OLTP part of the workload. These resources are never used for consolidating VMs onto the database server. In our running example, we reserve 7 CPU cores and around 900 GB of main-memory for OS, DATA, SNAP and OLTP (see Section 6.2.2), as main-memory DBMSs keep all data in main-memory all the time. Thus, around 100 GB of main-memory remain for OLAP. In contrast, we assume that the response time and throughput goals for OLAP are not that tight. Furthermore, we assume that arrival rates of OLAP request vary largely over time. Therefore, we use these resources for co-locating VMs onto the database server during phases of low OLAP load.

For the OLAP requests of our telemarketing example, we choose a reasonable max response time goal (g_{rt}) of 4 minutes on the database server (with network delays etc., this should allow end-to-end response time goals below 5 minutes) and a percentile of 99%. For simplicity, we use the same cSLO for all OLAP requests. For the given server, we estimate a worst-case execution time (e_m) of 2 minutes for in-memory processing of a single OLAP request and thus cover reasonable complex OLAP queries including join operations on large amounts of data. We have to make worst case assumptions, because the server needs to meet the derived cSLO always, with the only exception of hardware failures. The worst-case in-memory execution time estimate should be less than the max response time goal, as the difference between worst-case and average-case execution time is pretty small for main-memory DBMSs and some requests may be delayed due to co-location of VMs as described below. The server of our example fulfills this requirement. We reserved 7 CPU cores for OLTP and reserve one more CPU core for overhead introduced by elastic workload management. Thus 24 CPU cores (c) remain for OLAP processing. We choose the highest arrival rate that is supported by the given server as peak arrival rate (r_{a_p}), in order to find a cSLO that reflects the processing power of the server. In the following, we use formulas shown in Figures 6.9 and 6.10. According to Formula 6.1, the server supports a peak arrival rate of 12 requests per minute (720 requests per hour). As discussed above, the min throughput goal (g_{tp}) equals the peak arrival rate.

Based on these figures, we can derive the largest possible arrival window. In the worst case, we have peak arrival rate and all requests arrive at the beginning of

the given arrival window, thus g_{rt} time is available for processing all requests of the given arrival window (as discussed above g_{rt} does not include network delays etc.). Formula 6.2 describes that g_{rt} has to be large enough to process the percentile of these requests with the given number of CPU cores. From this inequality, we can derive an upper bound for the largest acceptable arrival window (w_a) as shown in Formula 6.3. The number of requests of the largest burst that can be processed with inter-query parallelization ($\sharp_{burst}$), can be computed as shown in Formula 6.4. Based on this figure, we find an attractive arrival window of 4 minutes according to Formula 6.5. Larger arrival windows are more attractive, because they allow for more burstiness. We only consider inter-query parallelization, but intra-query parallelization is an interesting direction for future work.

Furthermore, we need to determine the shortest possible measurement window (w_m) such that the average response time of all requests is still below g_{rt}. If all requests are processed in-memory, the average response time should be well below g_{rt}, but requests may be delayed due to co-location of VMs. The maximal number of delayed requests depends on the time it takes to give resources that were used for VMs back to the database. We refer to this as the *Elasticity Delay* (e_d). VMs can either be migrated to another server or be frozen and saved to disk. In the former case, the elasticity delay depends on the available network bandwidth and in the latter case on the available disk I/O bandwidth. In both cases, the entire state of the VMs (RAM, registers, etc.) has to be transferred. We decided for the latter approach, because we did not have a second server with similar capacity and connected with a high speed network, like InfiniBand. On our server, it takes less than 2 minutes to freeze and store 24 VMs with around 4 GB memory each on disk. These VMs correspond to the amount of OLAP resources that can be used for co-locating VMs. To limit the number of delayed requests per measurement window, we define that the resource allocation of the database is reduced at most once per measurement window. Thus, the shorter the measurement window, the more often the resource allocation of the database can be reduced, e.g. per day. In the worst case, load jumps from zero to peak after all resources have been taken away from the main-memory DBMS and load stays at peak until after resources are given back. Until then, all requests arriving during this elasticity delay — at most at peak arrival rate — are delayed, because all physical resources are needed to handle peak throughput as the server is sized for peak load. Thus, the maximal

$$r_{a\_p} = c \times \frac{1}{e_m} \tag{6.1}$$

$$g_{rt} \geq \frac{r_{a\_p} \times w_a \times p}{c} \times e_m \tag{6.2}$$

$$w_a \leq {w_a}^* = \frac{g_{rt} \times c}{r_{a\_p} \times p \times e_m} \tag{6.3}$$

$$\sharp_{burst} = c \times \lfloor \frac{\lfloor r_{a\_p} \times {w_a}^* \rfloor}{c} \rfloor \tag{6.4}$$

$$w_a = \frac{\sharp_{burst}}{r_{a\_p}} \tag{6.5}$$

$$\sharp_{delayed} = c \times \lceil \frac{\lceil r_{a\_p} \times d_e \rceil}{c} \rceil \tag{6.6}$$

$$\sharp_{delayed} \leq r_{a\_p} \times w_m \times (100\% - p)) \tag{6.7}$$

$$w_m \geq \frac{\sharp_{delayed}}{r_{a\_p} \times (100\% - p))} \tag{6.8}$$

$$\delta_{rt} = (w_a - d_e) + d_e = w_a \tag{6.9}$$

$$max_{rt} \leq \delta_{rt} + (\lceil \frac{\sharp_{delayed}}{m} \rceil \times e_d) \tag{6.10}$$

$$\sharp_{all\_m} = \lfloor r_{a\_p} \times w_m \rfloor \tag{6.11}$$

$$\sharp_{\neg delayed} = \sharp_{all\_m} - \sharp_{delayed} \tag{6.12}$$

$$\sum_{rt}^{delayed} = m \times \sum_{k=1}^{\lceil \frac{\sharp_{delayed}}{m} \rceil} (\delta_{rt} + (k \times e_d)) \tag{6.13}$$

$$\sharp_{all\_a} = \lfloor r_{a\_p} \times w_a \rfloor \tag{6.14}$$

Figure 6.9: Formulas

$$avg_{rt\_m} = \frac{c \times \sum_{k=1}^{\frac{\sharp_{all\_a}}{c}} (k \times e_m)}{\sharp_{all\_a}} \tag{6.15}$$

$$\sum_{rt}^{\neg delayed} = avg_{rt\_m} \times \sharp_{\neg delayed} \tag{6.16}$$

$$avg_{rt} \leq \frac{\sum_{rt}^{delayed} + \sum_{rt}^{\neg delayed}}{\sharp_{all\_m}} \tag{6.17}$$

Figure 6.10: Formulas

number of delayed requests ($\sharp_{delayed}$) is 24 and can be derived according to Formula 6.6. Of course, other reasons for missing response time goals, like network delays, also have to be considered, but this is orthogonal to our approach. $\sharp_{delayed}$ has to be in accordance with the percentile requirements, this requirement is expressed by Formula 6.7. From this requirement, we can derive a lower bound of a bit more than 3 hours (200 minutes) for the length of the measurement window (w_m), as described by formula 6.8.

cSLOs require that the average response time of all requests (avg_{rt}) is below the percentile response time goal (g_{rt}). Thus delayed requests cannot be delayed forever. We propose to process these requests on-disk (with only little main-memory) and to use the CPU core reserved for elastic workload management in order to avoid the processing overhead of on-disk processing from affecting in-memory processing of other requests. The response time of on-disk requests depends on the used multi programming level (MPL), which defines how many on-disk requests are processed concurrently — an MPL of one means that requests are processed sequentially. For our example server, we chose an MPL (m) of 4. Furthermore, we estimate a worst-case on-disk execution time (e_d) of 15 minutes for the given MPL. This estimation has to be much more conservative than the estimation of in-memory execution times, because we process on-disk requests concurrently on a single CPU core (masking delays caused by disk I/O) and due to the complexity of today's storage solutions with caches at several levels of the storage hierarchy.

Elastic workload management has to select which requests to process on disk. In the following, we analyze two promising strategies: Separate Queue and Common Queue. As discussed above, the average response time depends on the measurement

window. Thus we derive a second lower bound for the measurement window for each strategy. The higher one of the two lower bounds is the shortest possible measurement window. We derive the shortest possible measurement window and the max. response time for both strategies and choose the strategy with the shorter measurement window as long as the corresponding max. response time is acceptable, because the shorter the measurement window, the more often the resource allocation of the database can be reduced, e.g. per day.

Separate Queue Strategy

The Separate Queue Strategy puts delayed requests into a separate queue during the elasticity delay and processes them on-disk using the given MPL. The advantage of this approach is that at most $\sharp_{delayed}$ requests are delayed. The disadvantage of this strategy is that the delayed requests have to wait in queue which may result in high max response time. The max response time can be derived as follows: In the worst case, delayed requests arrived already at the beginning of the arrival window and the elasticity delay occurs at the end of the arrival window. Thus the delayed requests have already been waiting in queue for $w_a - d_e$ before on-disk execution even starts. Furthermore, if VMs are to be frozen and saved to disk, there is competition for disk I/O and on-disk requests may have to wait until the end of the elasticity delay until on-disk request processing starts. In the following, we use formulas shown in Figures 6.9 and 6.10. The response time delta (δ_{rt}) captures this, as shown in Formula 6.9. Delayed requests are delayed by the mentioned response time delta and then processed on disk with the given MPL. Thus, max response time (max_{rt}) is 94 minutes and can be computed according to Formula 6.10. As can be seen from this formula, max response time for this strategy highly depends on the on-disk MPL and thus the performance of the storage solution.

For this strategy, the average response time for the measurement window can be approximated conservatively as follows. Formula 6.11 shows how the peak number of requests per measurement window can be computed. Formula 6.12 shows how the number of requests that are not delayed (per measurement window) can be computed. The sum of response times of all delayed requests can be computed according to Formula 6.13. Formula 6.14 shows how the peak number of requests per arrival window can be computed. This figure is needed to compute the average response time for in-memory execution, according to Formula 6.15. In order

to compensate for requests that are delayed due to elasticity, we process all other requests in-memory. Thus, the sum of response times of all requests that are not delayed can be computed according to Formula 6.16. Based on these figures, an upper bound for the average response time of a given measurement window can be derived according to Formula 6.17. The measurement window has to be larger or equal the lower bound defined above and large enough such that the corresponding average response time is less than equal the percentile response time goal. In our example, the average response time for the lower bound measurement window is 3 minutes 33 seconds and thus meets this requirement. If the lower bound measurement window does not meet this requirement, it is more difficult to find an attractive measurement window length. Formula 6.17 defines a dependency between the average response time and the measurement window length. The average response time can be approximated conservatively as the percentile response time goal. Then the only remaining variable is the measurement window length and the inequality can be transformed to find an attractive measurement window length.

Common Queue Strategy

The second strategy is to keep all requests in the same queue and to keep track of how many requests need to be processed on disk (on-disk request counter). In the following, we use formulas shown in Figure 6.11. Whenever an on-disk request finishes, a fresh request can be taken from the common queue and be processed on-disk until the on-disk request counter has reached zero. The advantage of this approach is that the max response time is limited by Formula 6.18, as a given on-disk request waits at most one arrival window in-queue before being processed on-disk. The idea is that on-disk requests are replaced by newly arriving requests in each arrival window. If no new requests are available, peak load is over, the remaining requests can be processed in-memory and the on-disk request counter can be set to zero. The disadvantage of this strategy is that the average response time may be higher, because more requests than $\sharp_{delayed}$ may be delayed. For each group of concurrently running on-disk requests, a number of additional requests equal to the current value of the on-disk request counter is delayed per arrival window. To differentiate between the different kinds of delayed requests, we refer to the original requests that are delayed on-disk as $\sharp_{delayed_d}$ (see Formula 6.19) and to the additional request that are delayed in-memory as $\sharp_{delayed_m}$. On-disk requests are processed in

groups or "blocks" with MPL (m) and the number of "blocks" required to process all on-disk requests ($\sharp_{block}$) can be computed according to Formula 6.20, in our example 6. The measurement window has to be larger than the time required to process all on-disk requests, as described by Formula 6.21. Using this lower bound for the measurement window, a lower bound for $\sharp_{delayed_m}$ can be determined according to Formula 6.22 and Formula 6.23. Based on that Figure, a lower bound for the total number of delayed requests can be determined according to Formula 6.24. $\sharp_{delayed\_all}$ has to be in accordance with the percentile requirements and a second lower bound for the measurement window can be derived according to Formula 6.25 in the same way as for the other strategy. In our example, this second lower bound for w_m is 4100 minutes (or almost 3 days). This is very high in comparison to the measurement window of the *Separate Queue Strategy* (200 minutes). In other words, the resource allocation of the database system can be reduced at most twice per week with this strategy in comparison to 7 times per day with the other strategy. Therefore we choose the *Separate Queue Strategy*, as we deem the corresponding max response time of 94 minutes acceptable for our telemarketing OLAP scenario.

The analytical model is based on worst-case execution time estimates and employs conservative approximation, so there might be slightly better cSLOs that could still be met. But for the derived cSLO we can guarantee that it will always be met by the given server, with the only exception of hardware failures. This represents a guaranteed minimal performance goal. Furthermore, hardware failures are covered by the availability goal defined in the SLA. Moreover, we expect to derive attractive cSLOs, as we assume that the difference between worst-case and average-case is pretty small for emerging main-memory DBMSs. Figure 6.12 sums up the figures of our running example and shows the selected cSLO figures, characteristics of our example server, derived cSLO figures, the resulting elasticity and expected response times. In our example, a server with the given characteristics can fulfill the cSLO while the resource allocation of the DBMS may be reduced 7 times a day ($\sharp_{elasticity}$). The arrival window is maximal and the measurement window is minimal. Those figures can be derived from the other cSLO properties and the server characteristics as described above. This analysis is based on worst case assumptions, so it is probable that less requests miss their response time goals and that average response time is even lower. If the actual goal misses were monitored, it would be possible to reduce the resource allocation of the DBMS even more often.

$$max_{rt} \leq w_a + e_d \tag{6.18}$$

$$\sharp_{delayed\_d} = \sharp_{delayed} \tag{6.19}$$

$$\sharp_{block} = \lceil \frac{\sharp_{delayed\_d}}{m} \rceil \tag{6.20}$$

$$w_m \geq {w_m}^* = \sharp_{block} \times e_d \tag{6.21}$$

$$\sharp_{delayed\_m^*} = \sharp_{delayed\_d} \times \lfloor \frac{{w_m}^*}{w_a} \rfloor \tag{6.22}$$

$$\sharp_{delayed\_m} \geq \sharp_{delayed\_m^*} - (m \times \sum_{k=0}^{\sharp_{block}-1} k) \tag{6.23}$$

$$\sharp_{delayed\_all} \geq \sharp_{delayed\_m} + \sharp_{delayed\_d} \tag{6.24}$$

$$w_m \geq \frac{\sharp_{delayed\_all}}{r_{a\_p} \times (100\% - p))} \tag{6.25}$$

Figure 6.11: Formulas

Selected cSLO Figures	
p	99%
g_{rt}	4 min

Server Characteristics	
c	24
e_m	2 min
m	4
e_d	15 min
d_e	2 min

Derived cSLO Figures	
g_{tp}	12 per min
w_a	4 min
w_m	200 min

Resulting Elasticity	
$\sharp_{elasticity}$	7 per day

Expected Response Times	
avg_{rt}	$\leq$ 3 min 33 sec
max_{rt}	$\leq$ 94 min

Figure 6.12: Running Example

6.4 Experimental Evaluation

In the experimental evaluation of [96], we already showed that co-location of VMs with emerging main-memory database systems, like HyPer, works well for a continuous query workload without spikes that was modeled after the service demand of a SAP enterprise application used in a large business. Furthermore, we showed that spikes in demand may cause increased response times that may lead to SLO misses and SLA violations. The reason is that it takes time to give resources that are used for VMs back to the DBMS, as VMs need to be migrated to other servers or frozen to disk first. These delays cause increased response times for demand spikes, when demand was low long enough for co-locating VMs and then demand suddenly increases heavily. In Section 6.3, we proposed an analytical model for deriving an attractive cSLO that can be met by our test server even if there are spikes in demand.

For our evaluation, we use the following two workloads. The *Bursty Workload* includes a worst-case demand spike and is made up of five bursts ($B1$, $B2$, $B3$, $B4$, and $B5$) that occur in 4 minute intervals. Each burst consists of join queries from CH-benCHmark that join order and order-line tables. There are three query types ($Q1$, $Q2$ and $Q3$), corresponding to the same join query, but with different number of warehouses (750, 1500 and 3000). The estimated worst-case in-memory execution time of $Q1$ is 30 second, 1 minute for $Q2$ and 2 minutes for $Q3$. The estimated worst-case on-disk execution time of $Q1$ is 4 minutes, 8 minutes for $Q2$ and 15 minutes for $Q3$. Burst $B1$ consists of $48 \times Q3$ and requires all OLAP resources to meet the cSLO. Burst $B2$ consists of $24 \times Q3$ and requires half of the OLAP resources to meet the cSLO. Burst $B3$ consists of $24 \times Q2$ and requires one quarter of the OLAP resources to meet the cSLO. Burst $B4$ consists of $1 \times Q1$ and can be executed on-disk while still meeting the cSLO. The last burst represents a demand spike and there are three variants representing different kinds of spikes: $B5a$ consists of $96 \times Q1$, $B5b$ consists of $97 \times Q1$ and $B5c$ consists of $48 \times Q3$ that is repeated 50 times. In contrast, the *Steady Workload* represents continuous peak demand. Every 30 seconds, 24 query requests of type $Q1$ are sent to the DBMS for a duration of 11 minutes and 30 seconds.

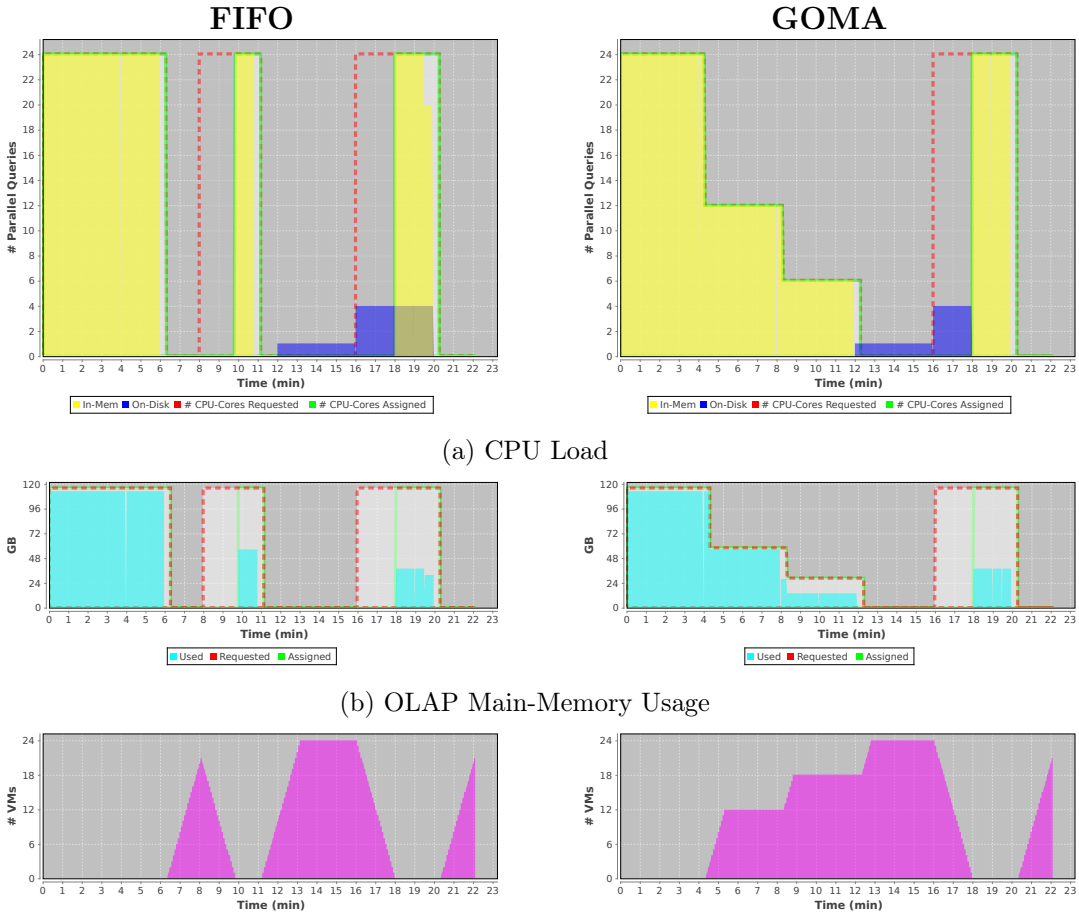
In the following experiments, we show the feasibility of elastic workload management and evaluate how much the progress of co-located VMs can be improved by elastic scheduling, while database service level objectives are met. In order to evalu-

ate progress of VMs, we use a statistical test program written in R, that runs inside each VM and that computes the well-known Mandelbrot set boundary which has a two-dimensional fractal shape. The size of the matrix was chosen such that around 4 GB are required to keep the results of the last 20 iterations in main-memory and the number of completed iterations indicates the progress made by the statistical test program. In our experiments, the VMs are only active while being co-located with the DBMS. Thus, we can sum up the number of completed iterations of all VMs to determine the progress of co-located VMs. The experiments were run on a commodity server with 1 TB of main memory (NUMA, 4 memory controllers) and four 2.27 GHz Processors with 8 CPU cores each (and two Hyper-Threading-Threads per CPU-Core). As discussed in Section 6.3.2, we assume that 24 CPU-Cores and around 100 GB of main-memory are available for OLAP, but actual OLAP load varies over time and spare OLAP resources may be used for co-locating VMs. We used Linux Control Groups for controlling the resource usage of the DBMS, Linux KVM for virtualization and the VMs were running Ubuntu Server 11.10 64 bit. [7]

We analyzed how co-location of VMs can be improved by elastic scheduling with GOMA using the *Bursty Workload* (with the first variant for the last burst). We compare GOMA with a simple FIFO scheduler that processes requests in arrival order. Figures 6.13a, 6.13b and 6.13c compare the CPU load, OLAP main-memory

[7]Because we did not have a second server with sufficient main-memory and high-speed network connection, we only suspended the VMs and later resumed them from main-memory instead of migrating VMs between different servers. To account for the expected delay for VM migration, we added a sleep duration of 4 seconds after suspending a VM, as we assume an elasticity delay of 2 minutes, which should be sufficient for migrating 24 VMs with 4-5 GB main-memory to another server. Furthermore, we only reserved enough memory to keep data etc. in memory, but we did not actually load corresponding data sets. We measured how long HyPer takes for executing a join with a given number of warehouses on our test server when using a given amount of main-memory. But during the experiments, our test program only slept for the corresponding amount of time. As part of his master thesis on the topic "Evaluation of Adaptive Workload Management Techniques for Main-memory Database Systems", Bernd Schultze developed a custom interface to Linux KVM, which allows to freeze 24 VMs with 4GB main-memory each and to store their state to the disks of our test server within 2 minutes. As his advisor, I (Michael Seibold) helped him integrating this new interface with the existing framework. This approach allowed to run the joins — representing the OLAP queries — during the experiments and to used spare OLAP resources for running VMs, although the available resources were limited using process control groups. He repeated the presented measurements and obtained similar results, which will be published in his master thesis.

usage and number of co-Located VMs of FIFO and GOMA. The charts on CPU load and OLAP main-memory usage also contain a dashed line indicating the number of CPU-cores and main-memory that are requested for OLAP by the *LocalController* respectively. Furthermore, a continuous line indicates the number of CPU-cores and main-memory that has been assigned for OLAP by the *GlobalController* respectively. With FIFO, co-located VMs completed a total of 6812 iterations and with GOMA 8835. This indicates that GOMA improved progress of co-located VMs by almost 30% (29,69%).



(a) CPU Load

(b) OLAP Main-Memory Usage

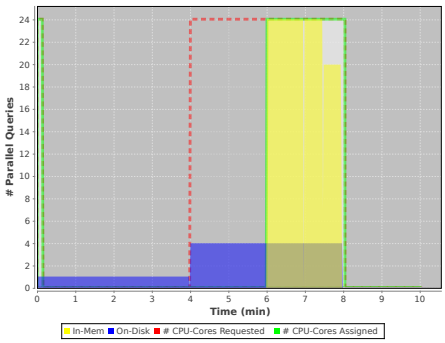
(c) Co-Located VMs

Figure 6.13: Bursty Workload

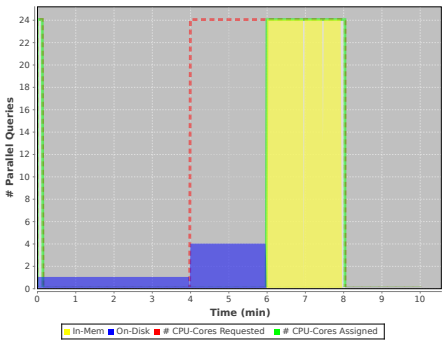
The table in Figure 6.16 presents the monitored statistics for the analyzed workloads and shows that GOMA always meets the cSLO, as the 99% percentile of response times (p_{rt}) is below the max response time goal (g_{rt}) of 4 minutes and the average response time of all requests (avg_{rt}) is also below the percentile response

time goal (g_{rt}). FIFO violates the cSLO for the *Bursty Workload*. For the *Bursty Workload*, the statistics for variant three (with $B5c$ as last burst) is shown. Moreover, Figure 6.16 lists min response time (min_{rt}), max response time (max_{rt}), total number of processed requests ($\sharp_{all}$) and number of delayed requests ($\sharp_{delayed}$).

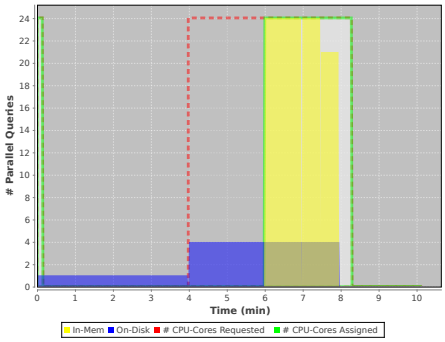
Furthermore, we analyzed how GOMA handles different kinds of spikes using the *Bursty Workload* and different variants for the demand spike of the last burst. $B5a$ represents a demand spike to half of peak. GOMA requests all resources, because of the expected delay until resources are returned to the DBMS, and starts to process four requests on-disk according the on-disk MPL chosen for our server. Once the resources are returned to the DBMS, the on-disk requests can be aborted and will still meet the response time goal after being restarted in-memory according to execution time estimates. GOMA would abort on-disk requests in this case and restart them in-memory to reduce average response time. But this Stop+Restart feature can be disabled with a configuration option. Figure 6.14a shows the CPU load without Stop+Restart and Figure 6.14b with Stop+Restart. In the former case, average response time is 194.727 sec and in the latter case it is 67 milliseconds lower (194.660 sec). $B5b$ represents a demand spike with one more request of $Q1$ than $B5a$. In this variant, GOMA would not abort on-disk requests even if Stop+Restart is enabled, because after restart in-memory, the response time goal will not be met anymore according to execution time estimates. This behavior is shown in Figure 6.14c. In all three cases, all requests meet the response time goal according to execution time estimates and the DBMS now has all resources back. As future work, GOMA could tell the *GlobalController* to ignore the last reduction of DBMS resources in these cases and thus the *GlobalController* could reduce DBMS resources more often than $\sharp_{elasticity}$ per day. The last variant $B5c$, represents a demand spike to peak load followed by peak load until the end of the measurement interval. This is the worst-case scenario and several requests miss the response time goal. We ran this experiment ten times faster than real-time and without VMs. As shown in Figure 6.14d, GOMA processes 24 requests on-disk (6 blocks of 4 parallel requests over a duration of 90 minutes) and the other requests in-memory. To meet the cSLO, the percentile of response times and the average response time of all requests have to be below the response time goal. This is the case, as shown in the second column of the table in Figure 6.16.



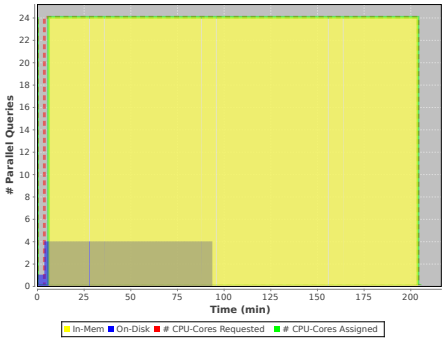
(a) Half Peak, No Disk Required, Without Stop+Restart



(b) Half Peak, No Disk Required, With Stop+Restart

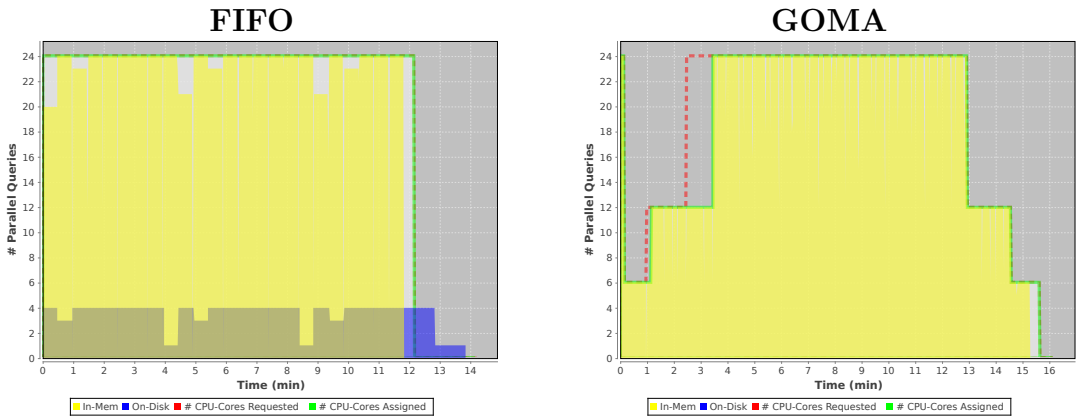


(c) Half Peak plus One, Disk Required

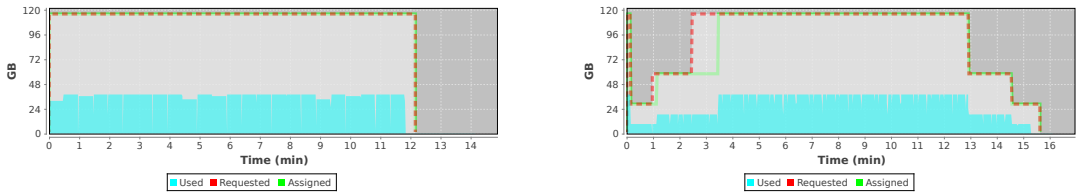


(d) Peak

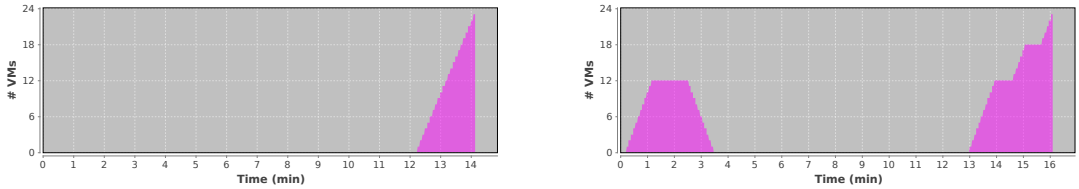
Figure 6.14: CPU Load for Different Kinds of Spikes with GOMA



(a) CPU Load



(b) OLAP Main-Memory Usage



(c) Co-Located VMs

Figure 6.15: Steady Workload

Moreover, we analyzed how GOMA handles workloads that do not contain bursts using the *Steady Workload*. Figures 6.15a and 6.15b compare the CPU load and OLAP main-memory usage of FIFO and GOMA. FIFO processes the requests in arrival order. When there are sufficient requests, 24 parallel requests are processed in-memory and four on-disk. CPU load is at peak and OLAP main-memory usage is low because in the *Steady Workload* all requests are of query type $Q1$, which requires less than 2 GB of main-memory per parallel query for in-memory processing. In the beginning, GOMA only requests a quarter of the OLAP resources and starts co-locating VMS as shown in Figure 6.15c, because there are only few requests and those can be delayed while still meeting the cSLO. As more requests are queued, GOMA requests more resources. After less than 4 minutes, GOMA uses the same amount of resources as FIFO. GOMA processes all requests in-memory to keep average response time low, as there are no demand spikes. After 12 minutes, there are no new requests and FIFO is done. At this time, GOMA still has queued requests and processes them according to the cSLO while signaling reduced resource requirements. As shown in the last two columns of the table in Figure 6.16, both meet the cSLO for the *Steady Workload*.

	Bursty		Steady	
	FIFO	GOMA	FIFO	GOMA
avg_{rt}	276 sec	209 sec	35 sec	149 sec
min_{rt}	120 sec	60 sec	30 sec	30 sec
max_{rt}	360 sec	5398 sec	240 sec	238 sec
p_{rt}	359 sec	240 sec	240 sec	238 sec
$\sharp_{all}$	2497	2497	576	576
$\sharp_{delayed}$	1200	24	0	0

Figure 6.16: Monitored cSLO Fulfillment for Bursty and Steady Workload

6.5 Conclusions

The proposed elastic workload management approach for emerging main-memory DBMSs allows to make use of spare resources on database servers by temporarily running other applications on the database server using virtual machines and ensures that the DBMS meets service level objectives of mixed workloads despite co-located VMs. The presented experimental results show the feasibility of the proposed approach and demonstrate that resource utilization of database servers can be improved as co-located VMs make significant progress. Furthermore, the presented experimental results show that elastic scheduling according to the GOMA approach can help to make spare resource better usable for co-locating VMs. As future work, elastic workload management could also be used for intelligent power management (e.g. hibernate CPU cores or deactivate unused memory banks etc.). Elastic workload management could help to identify spare resources, that can be put into power-save mode and to find out when to power them on again. Thereby, energy costs could be reduced without co-locating VMs. But, intelligent power management does not improve overall resource utilization. In contrast, consolidation of applications on formerly dedicated database servers may allow to reduce the total number of required servers, at least for large data centers.

Chapter 7

Conclusions

Operational Business Intelligence is complementary to an important class of business applications and therefore represents an interesting kind of service to be provided in the cloud. Cloud providers typically employ multi-tenant architectures in order to reduce costs by consolidating several customers onto the same infrastructure. Multi-tenancy features should be integrated into next-generation multi-tenant DBMSs in order to reduce administration and maintenance costs, i.a. by enabling seamless upgrades. The proposed data model can be used to capture the evolution and extensibility of a SaaS application explicitly, including data sharing. By supporting branching, the evolution can be captured along the development history of the corresponding application. Cloud services with automated administration procedures could be deployed on a large farm of commodity servers running independent DBMS instances. These DBMS instances have to efficiently process mixed workloads that result from Operational Business Intelligence, and a special purpose main-memory database system similar to MobiDB may be used. The proposed SaaS architecture may enable service providers to offer business applications with more powerful analytical features at competitive prices according to the SaaS model. Furthermore, MobiDB enables service providers to provide strict SLAs with stringent response time and throughput guarantees. With stricter SLAs it would be easier to compare different cloud offerings with on-premise solutions and thus cloud computing could become more attractive for potential customers. Moreover, the presented mixed workload benchmark could be used for analyzing the suitability of database systems for Operational Business Intelligence. Finally, cloud computing may foster the development of special purpose data management platforms and techniques like elastic workload management, for improving resource utilization. Improved resource utilization helps to reduce operational costs - including energy costs - and thereby may promote green computing.

Appendix: CH-benCHmark Queries

In the following, the SQL code of all 22 analytic queries of the CH-benCHmark is listed.[1]

Q1: Generate orderline overview

```
select ol_number,
       sum(ol_quantity) as sum_qty,
       sum(ol_amount) as sum_amount,
       avg(ol_quantity) as avg_qty,
       avg(ol_amount) as avg_amount,
        count(*) as count_order
from orderline where ol_delivery_d > '2007-01-02 00:00:00.000000'
group by ol_number order by ol_number
```

Q2: Most important supplier/item-combinations (those that have the lowest stock level for certain parts in a certain region)

```
select su_suppkey, su_name, n_name, i_id, i_name, su_address, su_phone,
    su_comment
from item, supplier, stock, nation, region,
     (select s_i_id as m_i_id,min(s_quantity) as m_s_quantity
      from stock, supplier, nation, region
      where mod((s_w_id*s_i_id),10000)=su_suppkey
      and su_nationkey=n_nationkey
      and n_regionkey=r_regionkey
      and r_name like 'Europ%'
      group by s_i_id) m
where i_id = s_i_id
and mod((s_w_id * s_i_id), 10000) = su_suppkey
and su_nationkey = n_nationkey
and n_regionkey = r_regionkey
and i_data like '%b'
and r_name like 'Europ%'
and i_id=m_i_id
and s_quantity = m_s_quantity
order by n_name, su_name, i_id
```

Q3: Unshipped orders with highest value for customers within a certain state

[1] The SQL code of all 22 queries can also be found at `http://www-db.in.tum.de/research/projects/CH-benCHmark/` (retrieved 07/31/2012).

```
select ol_o_id, ol_w_id, ol_d_id, sum(ol_amount) as revenue, o_entry_d
from customer, neworder, orders, orderline
where c_state like 'A%'
and c_id = o_c_id
and c_w_id = o_w_id
and c_d_id = o_d_id
and no_w_id = o_w_id
and no_d_id = o_d_id
and no_o_id = o_id
and ol_w_id = o_w_id
and ol_d_id = o_d_id
and ol_o_id = o_id
and o_entry_d > '2007-01-02 00:00:00.000000'
group by ol_o_id, ol_w_id, ol_d_id, o_entry_d
order by revenue desc, o_entry_d
```

Q4: Orders that were partially shipped late

```
select o_ol_cnt, count(*) as order_count
from orders
where o_entry_d >= '2007-01-02 00:00:00.000000'
and o_entry_d < '2012-01-02 00:00:00.000000'
and exists (select *
            from orderline
            where o_id = ol_o_id
            and o_w_id = ol_w_id
            and o_d_id = ol_d_id
            and ol_delivery_d >= o_entry_d)
group by o_ol_cnt
order by o_ol_cnt
```

Q5: Revenue volume achieved through local suppliers

```
select n_name, sum(ol_amount) as revenue
from customer, orders, orderline, stock, supplier, nation, region
where c_id = o_c_id
and c_w_id = o_w_id
and c_d_id = o_d_id
and ol_o_id = o_id
and ol_w_id = o_w_id
and ol_d_id=o_d_id
and ol_w_id = s_w_id
and ol_i_id = s_i_id
```

```
and mod((s_w_id * s_i_id),10000) = su_suppkey
and ascii(substr(c_state,1,1)) = su_nationkey
and su_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = 'Europe'
and o_entry_d >= '2007-01-02 00:00:00.000000'
group by n_name
order by revenue desc
```

Q6: Revenue generated by orderlines of a certain quantity

```
select sum(ol_amount) as revenue
from orderline
where ol_delivery_d >= '1999-01-01 00:00:00.000000'
and ol_delivery_d < '2020-01-01 00:00:00.000000'
and ol_quantity between 1 and 100000
```

Q7: Bi-directional trade volume between two nations

```
select su_nationkey as supp_nation,
       substr(c_state,1,1) as cust_nation,
       extract(year from o_entry_d) as l_year,
       sum(ol_amount) as revenue
from supplier, stock, orderline, orders, customer, nation n1, nation n2
where ol_supply_w_id = s_w_id
and ol_i_id = s_i_id
and mod((s_w_id * s_i_id), 10000) = su_suppkey

and ol_w_id = o_w_id
and ol_d_id = o_d_id
and ol_o_id = o_id
and c_id = o_c_id
and c_w_id = o_w_id
and c_d_id = o_d_id
and su_nationkey = n1.n_nationkey
and ascii(substr(c_state,1,1)) = n2.n_nationkey
and (
      (n1.n_name = 'Germany' and n2.n_name = 'Cambodia')
      or
      (n1.n_name = 'Cambodia' and n2.n_name = 'Germany'))
and ol_delivery_d between '2007-01-02 00:00:00.000000' and '2012-01-02
    00:00:00.000000'
```

```
group by su_nationkey, substr(c_state,1,1), extract(year from o_entry_d
    )
order by su_nationkey, cust_nation, l_year
```

Q8: Market share of a given nation for customers of a given region for a given part type

```
select extract(year from o_entry_d) as l_year,
    sum(case when n2.n_name = 'Germany' then ol_amount else 0 end) /
        sum(ol_amount) as mkt_share
from item, supplier, stock, orderline, orders, customer, nation n1,
    nation n2, region
where i_id = s_i_id
and ol_i_id = s_i_id
and ol_supply_w_id = s_w_id
and mod((s_w_id * s_i_id),10000) = su_suppkey
and ol_w_id = o_w_id
and ol_d_id = o_d_id
and ol_o_id = o_id
and c_id = o_c_id
and c_w_id = o_w_id
and c_d_id = o_d_id
and n1.n_nationkey = ascii(substr(c_state,1,1))
and n1.n_regionkey = r_regionkey
and ol_i_id < 1000

and r_name = 'Europe'
and su_nationkey = n2.n_nationkey
and o_entry_d between '2007-01-02 00:00:00.000000' and '2012-01-02
    00:00:00.000000'
and i_data like '%b'
and i_id = ol_i_id
group by extract(year from o_entry_d)
order by l_year
```

Q9: Profit made on a given line of parts, broken out by supplier nation and year

```
select n_name, extract(year from o_entry_d) as l_year, sum(ol_amount)
    as sum_profit
from item, stock, supplier, orderline, orders, nation
where ol_i_id = s_i_id
and ol_supply_w_id = s_w_id
and mod((s_w_id * s_i_id), 10000) = su_suppkey
and ol_w_id = o_w_id
```

```
and ol_d_id = o_d_id
and ol_o_id = o_id
and ol_i_id = i_id
and su_nationkey = n_nationkey
and i_data like '%BB'
group by n_name, extract(year from o_entry_d)
order by n_name, l_year desc
```

Q10: Customers who received their ordered products late

```
select c_id, c_last, sum(ol_amount) as revenue, c_city, c_phone, n_name
from customer, orders, orderline, nation
where c_id = o_c_id
and c_w_id = o_w_id
and c_d_id = o_d_id
and ol_w_id = o_w_id
and ol_d_id = o_d_id
and ol_o_id = o_id
and o_entry_d >= '2007-01-02 00:00:00.000000'
and o_entry_d <= ol_delivery_d
and n_nationkey = ascii(substr(c_state,1,1))
group by c_id, c_last, c_city, c_phone, n_name
order by revenue desc
```

Q11: Most important (high order count compared to the sum of all order counts) parts supplied by suppliers of a particular nation

```
select s_i_id, sum(s_order_cnt) as ordercount
from stock, supplier, nation
where mod((s_w_id * s_i_id),10000) = su_suppkey
and su_nationkey = n_nationkey
and n_name = 'Germany'
group by s_i_id
having sum(s_order_cnt) >
    (select sum(s_order_cnt) * .005
     from stock, supplier, nation
     where mod((s_w_id * s_i_id),10000) = su_suppkey
     and su_nationkey = n_nationkey
     and n_name = 'Germany')
order by ordercount desc
```

Q12: Determine whether selecting less expensive modes of shipping is negatively affecting the critical-priority orders by causing more parts to be received late by customers

```
select o_ol_cnt,
       sum(case when o_carrier_id = 1 or o_carrier_id = 2 then 1 else 0
           end) as high_line_count,
       sum(case when o_carrier_id <> 1 and o_carrier_id <> 2 then 1
           else 0 end) as low_line_count
from orders, orderline
where ol_w_id = o_w_id
and ol_d_id = o_d_id
and ol_o_id = o_id
and o_entry_d <= ol_delivery_d
and ol_delivery_d < '2020-01-01 00:00:00.000000'
group by o_ol_cnt
order by o_ol_cnt
```

Q13: Relationships between customers and the size of their orders

```
select c_count, count(*) as custdist
from (select c_id, count(o_id)
      from customer left outer join orders on (
           c_w_id = o_w_id
           and c_d_id = o_d_id
           and c_id = o_c_id
           and o_carrier_id > 8)
      group by c_id) as c_orders (c_id, c_count)
group by c_count
order by custdist desc, c_count desc
```

Q14: Market response to a promotion campaign

```
select 100.00 *
    sum(case when i_data like 'PR%' then ol_amount else 0 end) / 1+sum(
        ol_amount) as promo_revenue
from orderline, item
where ol_i_id = i_id and ol_delivery_d >= '2007-01-02 00:00:00.000000'
and ol_delivery_d < '2020-01-02 00:00:00.000000'
```

Q15: Determines the top supplier

```
with revenue (supplier_no, total_revenue) as (
    select mod((s_w_id * s_i_id),10000) as supplier_no,
           sum(ol_amount) as total_revenue
    from orderline, stock
    where ol_i_id = s_i_id and ol_supply_w_id = s_w_id
    and ol_delivery_d >= '2007-01-02 00:00:00.000000'
```

```
    group by mod((s_w_id * s_i_id),10000))
select su_suppkey, su_name, su_address, su_phone, total_revenue
from supplier, revenue
where su_suppkey = supplier_no
and total_revenue = (select max(total_revenue) from revenue)
order by su_suppkey
```

Q16: Number of suppliers that can supply parts with given attributes

```
select i_name,
       substr(i_data, 1, 3) as brand,
       i_price,
       count(distinct (mod((s_w_id * s_i_id),10000))) as supplier_cnt
from stock, item
where i_id = s_i_id
and i_data not like 'zz%'
and (mod((s_w_id * s_i_id),10000)) not in
    (select su_suppkey
     from supplier
     where su_comment like '%bad%')
group by i_name, substr(i_data, 1, 3), i_price
order by supplier_cnt desc
```

Q17: Average yearly revenue that would be lost if orders were no longer filled for small quantities of certain parts

```
select sum(ol_amount) / 2.0 as avg_yearly
from orderline, (select i_id, avg(ol_quantity) as a
                 from item, orderline
                 where i_data like '%b'
                 and ol_i_id = i_id
                 group by i_id) t
where ol_i_id = t.i_id
and ol_quantity < t.a
```

Q18: Rank customers based on their placement of a large quantity order

```
select c_last, c_id o_id, o_entry_d, o_ol_cnt, sum(ol_amount)
from customer, orders, orderline
where c_id = o_c_id
and c_w_id = o_w_id
and c_d_id = o_d_id
and ol_w_id = o_w_id
and ol_d_id = o_d_id
```

```
and ol_o_id = o_id
group by o_id, o_w_id, o_d_id, c_id, c_last, o_entry_d, o_ol_cnt
having sum(ol_amount) > 200
order by sum(ol_amount) desc, o_entry_d
```

Q19: Machine generated data mining (revenue report for disjunctive predicate)

```
select sum(ol_amount) as revenue
from orderline, item
where ( ol_i_id = i_id
          and i_data like '%a'
          and ol_quantity >= 1
          and ol_quantity <= 10
          and i_price between 1 and 400000
          and ol_w_id in (1,2,3))
or ( ol_i_id = i_id
      and i_data like '%b'
      and ol_quantity >= 1
      and ol_quantity <= 10
      and i_price between 1 and 400000
      and ol_w_id in (1,2,4))
or ( ol_i_id = i_id
      and i_data like '%c'
      and ol_quantity >= 1
      and ol_quantity <= 10
      and i_price between 1 and 400000
      and ol_w_id in (1,5,3))
```

Q20: Suppliers in a particular nation having selected parts that may be candidates for a promotional offer

```
select su_name, su_address
from supplier, nation
where su_suppkey in
    (select mod(s_i_id * s_w_id, 10000)
     from stock, orderline
     where s_i_id in
         (select i_id
          from item
          where i_data like 'co%')
     and ol_i_id=s_i_id
     and ol_delivery_d > '2010-05-23 12:00:00'
     group by s_i_id, s_w_id, s_quantity
```

```
      having 2*s_quantity > sum(ol_quantity))
and su_nationkey = n_nationkey
and n_name = 'Germany'
order by su_name
```

Q21: Suppliers who were not able to ship required parts in a timely manner

```
select su_name, count(*) as numwait
from supplier, orderline l1, orders, stock, nation
where ol_o_id = o_id
and ol_w_id = o_w_id
and ol_d_id = o_d_id
and ol_w_id = s_w_id
and ol_i_id = s_i_id
and mod((s_w_id * s_i_id),10000) = su_suppkey
and l1.ol_delivery_d > o_entry_d
and not exists (select *
                from orderline l2
                where l2.ol_o_id = l1.ol_o_id
                and l2.ol_w_id = l1.ol_w_id
                and l2.ol_d_id = l1.ol_d_id
                and l2.ol_delivery_d > l1.ol_delivery_d)
and su_nationkey = n_nationkey
and n_name = 'Germany'
group by su_name
order by numwait desc, su_name
```

Q22: Geographies with customers who may be likely to make a purchase

```
select substr(c_state,1,1) as country,
       count(*) as numcust,
       sum(c_balance) as totacctbal
from  customer
where substr(c_phone,1,1) in ('1','2','3','4','5','6','7')
and c_balance > (select avg(c_BALANCE)
                 from customer
                 where c_balance > 0.00
                 and substr(c_phone,1,1) in ('1','2','3','4','5','6','7
                     '))
and not exists (select *
                from orders
                where o_c_id = c_id
                and o_w_id = c_w_id
```

```
                    and o_d_id = c_d_id)
group by substr(c_state,1,1)
order by substr(c_state,1,1)
```

List of Figures

List of Tables

Bibliography

[1] Srini Acharya, Peter Carlin, Cesar Galindo-Legaria, Krzysztof Kozielczyk, Pawel Terlecki, and Peter Zabback. Relational Support for Flexible Schema Scenarios. *PVLDB*, 1(2):1289–1300, 2008.

[2] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13. ACM, 2006.

[3] Rakesh Agrawal, Amit Somani, and Yirong Xu. Storage and Querying of E-Commerce Data. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB)*, pages 149–158. Morgan Kaufmann, 2001.

[4] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *PVLDB*, 5(10):968–979, 2012.

[5] Yanif Ahmad and Christoph Koch. DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases. *PVLDB*, 2(2):1566–1569, 2009.

[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[7] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques. In *Proceedings of the 2008 SIGMOD International Conference on Management of Data*, pages 1195–1206. ACM, 2008.

[8] Stefan Aulbach, Dean Jacobs, Alfons Kemper, and Michael Seibold. A Comparison of Flexible Schemas for Software as a Service. In *Proceedings of the*

2009 SIGMOD International Conference on Management of Data, pages 881–888. ACM, 2009.

[9] Stefan Aulbach, Michael Seibold, Dean Jacobs, and Alfons Kemper. Extensibility and Data Sharing in Evolving Multi-Tenant Databases. In *Proceedings of the 27th International Conference on Data Engineering (ICDE)*, pages 99–110. IEEE, 2011.

[10] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüs, and Prashant Shenoy. "Cut Me Some Slack": Latency-Aware Live Migration for Databases. In *Proceedings of the 15th International Conference on Extending Database Technology (EDBT)*, pages 432–443. ACM, 2012.

[11] Luiz André Barroso and Urs Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37, 2007.

[12] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2009.

[13] Jennifer L. Beckmann, Alan Halverson, Rajasekar Krishnamurthy, and Jeffrey F. Naughton. Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, pages 58:1–58:10. IEEE, 2006.

[14] Hal Berenson, Philip Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 SIGMOD International Conference on Management of Data*, pages 1–10. ACM, 1995.

[15] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[16] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *Proceedings of the 2011 SIGMOD International Conference on Management of Data*, pages 37–48. ACM, 2011.

[17] Steve Bobrowski. Optimal Multitenant Designs for Cloud Apps. In *4th International Conference on Cloud Computing (CLOUD)*, pages 654–659. IEEE, 2011.

[18] Anja Bog, Hasso Plattner, and Alexander Zeier. A mixed transaction processing and operational reporting benchmark. *Information Systems Frontiers*, 13(3):321–335, 2011.

[19] Dhruba Borthakur, Joydeep Sen Sarma, Jonathan Gray, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand S. Aiyer. Apache Hadoop Goes Realtime at Facebook. In *Proceedings of the 2011 SIGMOD International Conference on Management of Data*, pages 1071–1080. ACM, 2011.

[20] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a Database on S3. In *Proceedings of the 2008 SIGMOD International Conference on Management of Data*, pages 251–263. ACM, 2008.

[21] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems, Design and Implementation (OSDI)*, pages 335–350. USENIX, 2006.

[22] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable Isolation for Snapshot Databases. *ACM Transactions on Database Systems*, 34(4):20:1–20:42, 2009.

[23] David G. Campbell, Gopal Kakivaya, and Nigel Ellis. Extreme Scale with Full SQL Language Support in Microsoft SQL Azure. In *Proceedings of the 2010 SIGMOD International Conference on Management of Data*, pages 1021–1023. ACM, 2010.

[24] Tuan Cao, Marcos Vaz Salles, Benjamin Sowell, Yao Yue, Alan Demers, Johannes Gehrke, and Walker White. Fast Checkpoint Recovery Algorithms for Frequently Consistent Applications. In *Proceedings of the 2011 SIGMOD International Conference on Management of Data*, pages 265–276. ACM, 2011.

[25] Dorin Carstoiu, Elena Lepadatu, and Mihai Gaspar. Hbase - non SQL Database, Performances Evaluation. *International Journal of Advancements in Computing Technology*, 2(5):42–52, 2010.

[26] Rick Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Record*, 39(4):12–27, 2010.

[27] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):4:1–4:26, 2008.

[28] Yun Chi, Hyun Jin Moon, and Hakan Hacigümüs. iCBS: Incremental Cost-based Scheduling under Piecewise Linear SLAs. *PVLDB*, 4(9):563–574, 2011.

[29] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems, Design and Implementation (NSDI)*, pages 273–286. USENIX, 2005.

[30] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. The mixed workload CH-benCHmark. In *Proceedings of the 4th International Workshop on Testing Database Systems (DBTest)*, pages 8:1–8:6. ACM, 2011.

[31] George P. Copeland and Setrag N. Khoshafian. A Decomposition Storage Model. In *Proceedings of the 1985 SIGMOD International Conference on Management of Data*, pages 268–279. ACM, 1985.

[32] Conor Cunningham, César A. Galindo-Legaria, and Goetz Graefe. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 998–1009. Morgan Kaufmann, 2004.

[33] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *PVLDB*, 3(1):48–57, 2010.

[34] Carlo Curino, Evan P. C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-Aware Database Monitoring and Consolidation. In *Proceedings of the 2011 SIGMOD International Conference on Management of Data*, pages 313–324. ACM, 2011.

[35] Carlo Curino, Evan P. C. Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Sam Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational Cloud: A Database-as-a-Service for the Cloud. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 235–240. Online Proceedings, `www.cidrdb.org/cidr2011` (retrieved 08/28/2012), 2011.

[36] Carlo Curino, Hyun J. Moon, and Carlo Zaniolo. Automating Database Schema Evolution in Information System Upgrades. In *Proceedings of the 2nd Workshop on Hot Topics in Software Upgrades (HotSWUp)*, pages 5:1–5:5. ACM, 2009.

[37] Carlo A. Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *PVLDB*, 4(2):117–128, 2010.

[38] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration. *PVLDB*, 4(8):494–505, 2011.

[39] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, pages 205–220. ACM, 2007.

[40] Bill Devlin, Jim Gray, Bill Laing, and George Spix. Scalability Terminology: Farms, Clones, Partitions, and Packs: RACS and RAPS. *Computing Research Repository (CoRR)*, cs.AR/9912010, 1999.

[41] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation Techniques for

Main Memory Database Systems. In *Proceedings of the 1984 SIGMOD International Conference on Management of Data*, pages 1–8. ACM, 1984.

[42] Ramez Elmasri and Sham Navathe. *Fundamentals of Database Systems*. Pearson/Addison Wesley, 4th edition, 2004.

[43] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of the 2011 SIGMOD International Conference on Management of Data*, pages 301–312. ACM, 2011.

[44] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: Data Management for Modern Business Applications. *SIGMOD Record*, 40(4):45–51, 2011.

[45] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance Tradeoffs for Client-Server Query Processing. In *Proceedings of the 1996 SIGMOD International Conference on Management of Data*, pages 149–160. ACM, 1996.

[46] Florian Funke, Alfons Kemper, Stefan Krompass, Harumi Kuno, Thomas Neumann, Anisoara Nica, Meikel Poess, and Michael Seibold. Metrics for Measuring the Performance of the Mixed Workload CH-benCHmark. In Raghunath Othayoth Nambiar and Meikel Poess, editors, *TPCTC*, volume 7144 of *Lecture Notes in Computer Science*, pages 10–30. Springer, 2011.

[47] Florian Funke, Alfons Kemper, and Thomas Neumann. Benchmarking Hybrid OLTP&OLAP Database Systems. In Härder et al. [58], pages 390–409.

[48] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP)*, pages 29–43. ACM, 2003.

[49] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. SharedDB: Killing One Thousand Queries With One Stone. *PVLDB*, 5(6):526–537, 2012.

[50] Georgios Giannikis, Philipp Unterbrunner, Jeremy Meyer, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. Crescando. In *Proceedings of the 2010*

SIGMOD International Conference on Management of Data, pages 1227–1230. ACM, 2010.

[51] Goetz Graefe. A generalized join algorithm. In Härder et al. [58], pages 267–286.

[52] Pablo Graubner, Matthias Schmidt, and Bernd Freisleben. Energy-efficient Management of Virtual Machines in Eucalyptus. In *4th International Conference on Cloud Computing (CLOUD)*, pages 243–250. IEEE, 2011.

[53] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[54] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.

[55] Hakan Hacigümüs, Bala Iyer, and Sharad Mehrotra. Providing Database as a Service. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 29–38. IEEE, 2002.

[56] Sebastian Hagen, Weverton Luis da Costa Cordeiro, Luciano Paschoal Gaspary, Lisandro Zambenedetti Granville, Michael Seibold, and Alfons Kemper. Planning in the Large: Efficient Generation of IT Change Plans on Large Infrastructures. Accepted for publication in Proceedings of 8th International Conference on Network and Service Management (CNSM), 2012.

[57] Sebastian Hagen, Michael Seibold, and Alfons Kemper. Efficient Verification of IT Change Operations or: How We Could Have Prevented Amazon's Cloud Outage. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 368–376. IEEE, 2012.

[58] Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, and Holger Schwarz, editors. *14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme", Kaiserslautern, Germany, February 28 - March 4, 2011*. GI, 2011.

[59] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There. In

Proceedings of the 2008 SIGMOD International Conference on Management of Data, pages 981–992. ACM, 2008.

[60] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: A Scalable Approach to Logging. *PVLDB*, 3(1):681–692, 2010.

[61] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB*, 1(2):1496–1499, 2008.

[62] Alfons Kemper and André Eickler. *Datenbanksysteme: Eine Einführung*. Oldenbourg, 8th edition, 2011.

[63] Alfons Kemper and Thomas Neumann. One Size Fits all, Again! The Architecture of the Hybrid OLTP&OLAP Database Management System HyPer. In Malú Castellanos, Umeshwar Dayal, and Volker Markl, editors, *BIRTE*, volume 84 of *Lecture Notes in Business Information Processing*, pages 7–23. Springer, 2010.

[64] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 27th International Conference on Data Engineering (ICDE)*, pages 195–206. IEEE, 2011.

[65] Setrag Khoshafian and Razmik Abnous. *Object Orientation: Concepts, Analysis and Design, Languages, Databases, Graphical User Interfaces, Standards*. Wiley, 2nd edition, 1995.

[66] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[67] Stefan Krompass, Daniel Gmach, Andreas Scholz, Stefan Seltzsam, and Alfons Kemper. Quality of Service Enabled Database Applications. In Asit Dan

and Winfried Lamersdorf, editors, *ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 215–226. Springer, 2006.

[68] Stefan Krompass, Harumi Kuno, Janet L. Wiener, Kevin Wilkinson, Umeshwar Dayal, and Alfons Kemper. Managing Long-Running Queries. In *Proceedings of the 12th International Conference on Extending Database Technology (EDBT)*, pages 132–143. ACM, 2009.

[69] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB*, 5(1):61–72, 2011.

[70] Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Kevin Wilkinson, Archana Ganapathi, and Stefan Krompass. Managing Dynamic Mixed Workloads for Operational Business Intelligence. In Shinji Kikuchi, Shelly Sachdeva, and Subhash Bhalla, editors, *DNIS*, volume 5999 of *Lecture Notes in Computer Science*, pages 11–26. Springer, 2010.

[71] Willis Lang, Srinath Shankar, Jignesh M. Patel, and Ajay Kalhan. Towards Multi-tenant Performance SLOs. In *Proceedings of the 28th International Conference on Data Engineering (ICDE)*, pages 702–713. IEEE, 2012.

[72] Edmond Lau and Samuel Madden. An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 703–714. ACM, 2006.

[73] Jiexing Li, Rimma V. Nehme, and Jeffrey Naughton. GSLPI: a Cost-Based Query Progress Indicator. In *Proceedings of the 28th International Conference on Data Engineering (ICDE)*, pages 678–689. IEEE, 2012.

[74] Gang Luo, Jeffrey F. Naughton, and Philip S. Yu. Multi-query SQL Progress Indicators. In Yannis E. Ioannidis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Michael Hatzopoulos, Klemens Böhm, Alfons Kemper, Torsten Grust, and Christian Böhm, editors, *EDBT*, volume 3896 of *Lecture Notes in Computer Science*, pages 921–941. Springer, 2006.

[75] David Maier and Jeffrey D. Ullman. Maximal Objects and the Semantics of Universal Relation Databases. *ACM Transactions on Database Systems*, 8(1):1–14, 1983.

[76] Stefan Manegold, Martin L. Kersten, and Peter Boncz. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. *PVLDB*, 2(2):1648–1653, 2009.

[77] Todd McKinnon. Plug Your Code in Here: An Internet Application Platform. Published Online, `www.hpts.ws/papers/2007/hpts_conference_oct_2007.ppt` (retrieved 08/22/2011), 2007.

[78] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. Published Online, `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf` (retrieved 08/28/2012), 2011.

[79] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[80] Henrik Mühe, Alfons Kemper, and Thomas Neumann. How to Efficiently Snapshot Transactional Data: Hardware or Software Controlled? In *Proceedings of the 7th International Workshop on Data Management on New Hardware (DaMoN)*, pages 17–26. ACM, 2011.

[81] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[82] Norifumi Nishikawa, Miyuki Nakano, and Masaru Kitsuregawa. Energy Efficient Storage Management Cooperated with Large Data Intensive Applications. In *Proceedings of the 28th International Conference on Data Engineering (ICDE)*, pages 126–137. IEEE, 2012.

[83] William O'Mullane, Nolan Li, María Nieto-Santisteban, Alex Szalay, Ani Thakar, and Jim Gray. Batch is back: CasJobs, serving multi-TB data on the Web. In *Proceedings of the 3rd International Conference on Web Services (ICWS)*, pages 33–40. IEEE, 2005.

[84] Pat O'Neil, Betty O'Neil, and Xuedong Chen. The Star Schema Benchmark (SSB). Published Online, `http://www.cs.umb.edu/~xuedchen/research/publications/StarSchemaB.PDF` (retrieved 08/28/2012), 2007.

[85] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMCloud. *Communications of the ACM*, 54(7):121–130, 2011.

[86] Alok Pareek. Addressing BI transactional flows in the real-time enterprise using GoldenGate TDM. In Malú Castellanos, Umeshwar Dayal, and Renée J. Miller, editors, *BIRTE*, volume 41 of *Lecture Notes in Business Information Processing*, pages 118–141. Springer, 2009.

[87] Jignesh M. Patel, Michael J. Carey, and Mary K. Vernon. Accurate Modeling of The Hybrid Hash Join Algorithm. In *Proceedings of the 1994 SIGMETRICS conference on Measurement and modeling of computer systems*, pages 56–66. ACM, 1994.

[88] Hasso Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In *Proceedings of the 2009 SIGMOD International Conference on Management of Data*, pages 1–2. ACM, 2009.

[89] Edward Ray and Eugene Schultz. Virtualization Security. In *Proceedings of the 5th Annual Cyber Security and Information Intelligence Research Workshop (CSIIRW)*, pages 43:1–43:5. ACM, 2009.

[90] Jan Schaffner, Benjamin Eckart, Dean Jacobs, Christian Schwarz, Hasso Plattner, and Alexander Zeier. Predicting In-Memory Database Performance for Automating Cluster Management Tasks. In *Proceedings of the 27th International Conference on Data Engineering (ICDE)*, pages 1264–1275. IEEE, 2011.

[91] Oliver Schiller, Benjamin Schiller, Andreas Brodt, and Bernhard Mitschang. Native Support of Multi-tenancy in RDBMS for Software as a Service. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT)*, pages 117–128. ACM, 2011.

[92] Michael Seibold, Dean Jacobs, and Alfons Kemper. Operational Business Intelligence: Meeting Strict Service Level Objectives for Mixed Workloads. *Accepted for publication in IT Professional.*

[93] Michael Seibold and Alfons Kemper. Database as a Service. *Datenbank-Spektrum*, 12(1):59–62, 2012.

[94] Michael Seibold and Alfons Kemper. GOMA: Elastic Workload Management for Emerging Main-memory DBMSs. Submitted to 29th IEEE International Conference on Data Engineering (ICDE), 2013.

[95] Michael Seibold, Alfons Kemper, and Dean Jacobs. Strict SLAs for Operational Business Intelligence. In *4th International Conference on Cloud Computing (CLOUD)*, pages 25–32. IEEE, 2011.

[96] Michael Seibold, Andreas Wolke, Martina Albutiu, Martin Bichler, Alfons Kemper, and Thomas Setzer. Efficient Deployment of Main-memory DBMS in Virtualized Data Centers. In *5th International Conference on Cloud Computing (CLOUD)*, pages 311–318. IEEE, 2012.

[97] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems. In *Proceedings of the 2nd Symposium on Cloud Computing (SOCC)*, pages 5:1–5:14. ACM, 2011.

[98] Darius Sidlauskas, Christian S. Jensen, and Simonas Saltenis. A Comparison of the Use of Virtual Versus Physical Snapshots for Supporting Update-Intensive Workloads. In *Proceedings of the 8th International Workshop on Data Management on New Hardware (DaMoN)*, pages 1–8. ACM, 2012.

[99] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proceedings of the 2012 SIGMOD International Conference on Management of Data*, pages 731–741. ACM, 2012.

[100] Alkis Simitsis, Kevin Wilkinson, Umeshwar Dayal, and Malu Castellanos. Optimizing ETL Workflows for Fault-Tolerance. In *Proceedings of the 26th In-*

ternational Conference on Data Engineering (ICDE), pages 385–396. IEEE, 2010.

[101] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. Automatic Virtual Machine Configuration for Database Workloads. *ACM Transactions on Database Systems*, 35(1), 2010.

[102] Benjamin Sowell, Wojciech Golab, and Mehul A. Shah. Minuet: A Scalable Distributed Multiversion B-Tree. *PVLDB*, 5(9):884–895, 2012.

[103] Benjamin Speitkamp and Martin Bichler. A Mathematical Programming Approach for Server Consolidation Problems in Virtualized Data Centers. *IEEE Transactions on Services Computing*, 3(4):266–278, 2010.

[104] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 149–160. ACM, 2001.

[105] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 1150–1160. ACM, 2007.

[106] Michael Stonebraker and Lawrence A. Rowe. The Design of Postgres. In *Proceedings of the 1986 SIGMOD International Conference on Management of Data*, pages 340–355. ACM, 1986.

[107] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stanley Zdonik. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 553–564. ACM, 2005.

[108] The Times-Ten Team. In-Memory Data Management for Consumer Transactions: The Times-Ten Approach. In *Proceedings of the 1999 SIGMOD International Conference on Management of Data*, pages 528–529. ACM, 1999.

[109] Alexander Thomson and Daniel J. Abadi. The Case for Determinism in Database Systems. *PVLDB*, 3(1):70–80, 2010.

[110] Sean Tozer, Tim Brecht, and Ashraf Aboulnaga. Q-Cop: Avoiding Bad Query Mixes to Minimize Client Timeouts Under Heavy Loads. In *Proceedings of the 26th International Conference on Data Engineering (ICDE)*, pages 397–408. IEEE, 2010.

[111] Transaction Processing Performance Council. TPC BENCHMARK C: Standard Specification. Published Online, `www.tpc.org/tpcc/spec/TPC-C_v5-11.pdf` (retrieved 08/22/2011), 2010.

[112] Transaction Processing Performance Council. TPC BENCHMARK H: Standard Specification. Published Online, `www.tpc.org/tpch/spec/tpch2.14.0.pdf` (retrieved 08/22/2011), 2011.

[113] Philipp Unterbrunner, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. Predictable Performance for Unpredictable Workloads. *PVLDB*, 2(1):706–717, 2009.

[114] VoltDB. TPC-C-like Benchmark Comparison - Benchmark Description. Published Online, `http://community.voltdb.com/node/134` (retrieved 08/22/2011), 2010.

[115] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems, Design and Implementation (OSDI)*, pages 181–194. USENIX, 2002.

[116] Craig D. Weissman and Steve Bobrowski. The Design of the Force.com Multitenant Internet Application Development Platform. In *Proceedings of the 2009 SIGMOD International Conference on Management of Data*, pages 889–896. ACM, 2009.

[117] Kevin Wilkinson, Alkis Simitsis, Malu Castellanos, and Umeshwar Dayal. Leveraging Business Process Models for ETL Design. In Jeffrey Parsons, Motoshi Saeki, Peretz Shoval, Carson Woo, and Yair Wand, editors, *ER*, volume 6412 of *Lecture Notes in Computer Science*, pages 15–30. Springer, 2010.

[118] Martin Wimmer, Daniela Eberhardt, Pia Ehrnlechner, and Alfons Kemper. Reliable and Adaptable Security Engineering for Database-Web Services. In Nora Koch, Piero Fraternali, and Martin Wirsing, editors, *ICWE*, volume 3140 of *Lecture Notes in Computer Science*, pages 502–515. Springer, 2004.

[119] Yi Zhang, Zhihu Wang, Bo Gao, Changjie Guo, Wei Sun, and Xiaoping Li. An Effective Heuristic for On-line Tenant Placement Problem in SaaS. In *Proceedings of the 8th International Conference on Web Services (ICWS)*, pages 425–432. IEEE, 2010.