UNIVERSITY OF CALIFORNIA

Santa Barbara

Scalable and Elastic Transactional Data Stores for Cloud Computing Platforms

A Dissertation submitted in partial satisfaction

of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Sudipto Das

Committee in Charge:

Professor Divyakant Agrawal, Co-Chair

Professor Amr El Abbadi, Co-Chair

Dr. Philip A. Bernstein

Professor Timothy Sherwood

December 2011

The Dissertation of
Sudipto Das is approved:

Dr. Philip A. Bernstein

Professor Timothy Sherwood

Professor Divyakant Agrawal, Committee Co-Chair

Professor Amr El Abbadi, Committee Co-Chair

November 2011

To my family: Baba, Ma, Didibhai, Chhordi, and Pamela.

Acknowledgements

"Saying thank you is more than good manners. It is good spirituality."

– Alfred Painter.

I thank God for everything, including my own self. Words fall short here.

With immense gratitude, I acknowledge my advisors, Professors Divy Agrawal and Amr El Abbadi, for providing continuous support, guidance, mentoring, and technical advise over the course of my doctoral study. I still remember the day in the Winter quarter of 2007 when I met Divy and Amr discussing the possibilities of joining their research group. I had barely competed my first course on Distributed Systems and had superficial knowledge of Database Systems. On that day, I never imagined that four years down the line, I will be writing my dissertation on a topic that marries these two research areas. Divy and Amr's research insights have made this dissertation possible. I am fortunate to have been mentored by Divy and Amr, both of whom have been awarded the prestigious Outstanding Graduate Mentor Award by UCSB's Senate. Besides their great guidance on the technical side, I will never forget their kind fatherly and friendly attitude. Their characters will continue to inspire me.

I cannot find words to express my gratitude to Dr. Phil Bernstein for being such a wonderful mentor, providing in-depth feedback on various research projects, and serving on my dissertation committee. When I started work on transaction processing, Divy and Amr gave me Phil's book on "Concurrency Control and Recovery in Database Systems," a classical text in transaction processing. And in 2010, when Phil invited me to work with him as a summer intern, it was my opportunity to learn from him many important skills of the trade. His continued guidance and mentoring has helped me improve on various aspects, such as writing and presentation, ability to critically evaluate my own work, and the ability to think deeper on problems.

I would also thank Professor Tim Sherwood for serving on my dissertation committee and for providing his valuable feedback, Professors Ömer Eğecioğlu and Xifeng Yan for their collaborations and useful feedback, and the Computer Science Department and UCSB for providing me the opportunity to pursue my doctoral study and also providing the necessary support infrastructure. I immensely enjoyed my interactions with Professor Janet Kayfetz, and I thank her for her inputs on technical writing and presentations. I am also indebted to the National Science Foundation for funding my research through grants IIS 0744539, III 1018637 and CNS 1053594, NEC Labs America for providing a generous gift to support my research, and to Amazon web services for providing credits to use their clusters for large scale experiments. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author and do not necessarily reflect the views of the sponsors.

I am also thankful to all my friends for the fun moments in my PhD student life. Special thanks to Aaron, Ceren, Shashank, Shiyuan, and Shoji for the wonderful moments we shared in the lab. I am also thankful to Aaron, Shashank, Shoji, and Shyam for the collaborations and contributions to some of the projects in this dissertation. I also thank my other past and present DSL colleagues: Ahmed, Arsany, Hatem, Lin, Ping, Stacy, Terri, and Zhengkui. I will also cherish the good times spent with my friends during my stay in Santa Barbara. Special thanks to Hero (Sayan), Barda (Pratim), Sau (Rajdeep), Dada (Anindya) and the rest of the UCSB gang for the wonderful moments.

Most importantly, my deepest gratitude is for my family for their constant support, inspiration, guidance, and sacrifices. My father and mother were constant source of motivation and inspiration. I wish my father was here today to witness this moment and share the happiness—may his soul rest in peace. My sisters, Didibhai and Chhordi, were my first teachers at home. Their affection and guidance was instrumental in me choosing Engineering and eventually continuing on to my PhD. Pamela, my better half, has been an angel in my life. She has been my best friend, on whom I can rely on getting support on any and every aspect of my life. I cannot imagine life without her constant support, dedication, and advice. Words are not enough for thanking her.

Curriculum Vitæ

Sudipto Das

November, 2011

Education

December 2011	**Doctor of Philosophy** in Computer Science, University of California, Santa Barbara, CA, USA.
June 2011	**Master of Science** in Computer Science, University of California, Santa Barbara, CA, USA.
June 2006	**Bachelor of Engineering** in Computer Science & Engineering, Jadavpur University, Kolkata, India.

Research Interests

Scalable data management systems, elastic and self-managing systems, large scale distributed systems, transaction processing, cloud computing.

Work Experience

- **UC Santa Barbara**, Santa Barbara, CA, USA
  (October 2006 - December 2011)
  Graduate Research and Teaching Assistant.
- **Microsoft Research**, Redmond, WA, USA
  (June 2010 - September 2010)
  Research Intern.
- **IBM Almaden Research Center**, San Jose, CA, USA
  (June 2009 - September 2009)
  Research Intern.
- **Google**, Mountain View, CA
  (June 2007 - September 2007)
  Software Engineering Intern.
- **IBM India Pvt. Limited**, Kolkata, India.
  (June 2005 - July 2005)
  Summer Intern.

Selected Publications

- *"Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration,"* **Sudipto Das**, Shoji Nishimura, Divyakant Agrawal, Amr El Abbadi. *In the 37th International Conference on Very Large Databases (**VLDB**) 2011.*

- *"Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms,"* Aaron Elmore, **Sudipto Das**, Divyakant Agrawal, Amr El Abbadi. *In the ACM International Conference on Management of Data (**SIGMOD**) 2011.*

- *"Hyder – A Transactional Record Manager for Shared Flash,"* Philip Bernstein, Colin Reid, **Sudipto Das**. *In the 5th Biennial Conference on Innovative Data Research (**CIDR**) 2011 **[Recipient of Best Paper Award]**.*

- *"$\mathcal{MD}$-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services,"* Shoji Nishimura, **Sudipto Das**, Divyakant Agrawal, Amr El Abbadi. *In 12th International Conference on Mobile Data Management (**MDM**) 2011**[Recipient of Best Paper Runner-up Award]**.*

- *"G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud,"* **Sudipto Das**, Divyakant Agrawal, Amr El Abbadi. *In the 1st ACM International Symposium on Cloud Computing (**SoCC**) 2010.*

- *"Ricardo: Integrating R and Hadoop,"* **Sudipto Das**, Yannis Simanis, Kevin S. Beyer, Rainer Gemulla, Peter J. Haas, John McPherson. *In the ACM International Conference on Management of Data (**SIGMOD**) 2010.*

- *"Big Data and Cloud Computing: Current State and Future Opportunities,"* Divyakant Agrawal, **Sudipto Das**, Amr El Abbadi. *Tutorial presentation at the 14th International Conference on Extending Database Technology (**EDBT**) 2011.*

- *"Big Data and Cloud Computing: New Wine or just New Bottles?,"* Divyakant Agrawal, **Sudipto Das**, Amr El Abbadi. *Tutorial presentation at the 36th International Conference on Very Large Databases (**VLDB**) 2010.*

- *"ElasTraS: An Elastic, Scalable, and Self Managing Transactional Database for the Cloud,"* **Sudipto Das**, Shashank Agarwal, Divyakant Agrawal, Amr El Abbadi. *UCSB Technical Report CS 2010-04.*

- *"ElasTraS: An Elastic Transactional Data Store in the Cloud,"* **Sudipto Das**, Divyakant Agrawal, Amr El Abbadi. *In the* 1st *Usenix Workshop on Hot topics on Cloud Computing (**HotCloud**) 2009.*

- *"Towards an Elastic and Autonomic Multitenant Database,"* Aaron Elmore, **Sudipto Das**, Divyakant Agrawal, Amr El Abbadi. *In the* 6th *International Workshop on Networking Meets Databases (**NetDB**) 2011.*

- *"Database Scalability, Elasticity, and Autonomy in the Cloud,"* Divyakant Agrawal, Amr El Abbadi, **Sudipto Das**, Aaron Elmore. *In the* 16th *International Conference on Database Systems for Advanced Applications (**DASFAA**) 2011* (Invited Paper).

- *"Data Management Challenges in Cloud Computing Infrastructures,"* Divyakant Agrawal, Amr El Abbadi, Shyam Antony, **Sudipto Das**. *In the* 6th *International Workshop on Databases in Networked Information Systems (**DNIS**) 2010* (Invited Paper).

- *"Anónimos: An LP based Approach for Anonymizing Weighted Social Network Graphs,"* **Sudipto Das**, Ömer Eğecioğlu, Amr El Abbadi. *To appear in the IEEE Transactions on Knowledge and Data Engineering (**TKDE**).*

- *"Anonymizing Weighted Social Network Graphs,"* **Sudipto Das**, Ömer Eğecioğlu, Amr El Abbadi. *In the* 26th *International Conference on Data Engineering (**ICDE**) 2010* (Short Paper).

- *"Thread Cooperation in Multicore Architectures for Frequency Counting Over Multiple Data Streams,"* **Sudipto Das**, Shyam Antony, Divyakant Agrawal, Amr El Abbadi. *In the* 35th *International Conference on Very Large Databases (**VLDB**) 2009.*

- *"CoTS: A Scalable Framework for Parallelizing Frequency Counting over Data Streams,"* **Sudipto Das**, Shyam Antony, Divyakant Agrawal, Amr El Abbadi. *In the* 25th *International Conference on Data Engineering (**ICDE**) 2009* (Short Paper).

◦ *"QUORUM: Quality of Service in Wireless Mesh Networks,"* Vinod Kone, **Sudipto Das**, Ben Y. Zhao, Haitao Zheng. *In Mobile Networks and Applications (**MONET**) Journal 12(5-6), April 2008.*

Awards and Honors

◦ 2011 Outstanding Student Award in Computer Science, UC Santa Barbara.

◦ Best Paper award at the 5^{th} biennial Conference on Innovative Data Systems Research (CIDR) 2011.

◦ Best Paper Runner-up award at the 12^{th} International Conference on Mobile Data Management (MDM) 2011.

◦ TCS-JU Best Graduating Student Award for 2006, Computer Science Dept., Jadavpur University, Kolkata.

◦ UCSB Graduate Division Dissertation Fellowship, Winter 2011.

◦ Outstanding Teaching Assistant for Graduate Course on Distributed Systems, Fall '08.

◦ UC Senate Travel Award for VLDB 2009 & ACM SoCC Student Travel Award 2010.

◦ UC President's Work Study award.

Professional Service and Activities

◦ Reviewer for VLDB Journal, IEEE Trans. of Know. and Data Engg., Journal of Dist. and Par. Databases, IEEE Internet Computing, IEEE Computer.

◦ Program Committee member for Workshop on Data Management in the Cloud 2012, UCSB Graduate Students Workshop 2010 & 2011.

◦ External Reviewer for SIGMOD 2008, CIKM 2008, ICDE 2009 & 2010, EDBT 2009, SSDBM 2009 & 2011, DaMoN 2009, SIGSPATIAL 2009, LADIS 2010, TODS, VLDB 2011 & 2012, EuroPar 2011, SoCC 2011.

- Helped organize the NSF Workshop “Science of Cloud” held in March 2011.
- Served as Graduate Student Association representative in departmental Colloquium committee 2009 & 2010, and Faculty search committee ’11.
- Student member of the ACM.
- Mentored four undergraduate student projects, one Masters projects, and one high school student with UCSB Summer Sessions.

Abstract

Scalable and Elastic Transactional Data Stores for Cloud Computing Platforms

by

Sudipto Das

Cloud computing has emerged as a multi-billion dollar industry and as a successful paradigm for web application deployment. Economies-of-scale, elasticity, and pay-per-use pricing are the biggest promises of cloud. Database management systems (DBMSs) serving these web applications form a critical component of the cloud software stack. In order to serve thousands of applications and their huge amounts of data, these DBMSs must scale-out to clusters of commodity servers. Moreover, to minimize their operating costs, such DBMSs must also be elastic, i.e., possess the ability to increase and decrease the cluster size in a live system. This is in addition to serving a variety of applications (i.e., supporting multitenancy) while being self-managing, fault-tolerant, and highly available.

The overarching goal of this dissertation is to propose abstractions, protocols, and paradigms to architect efficient, scalable, and practical DBMSs that address the unique set of challenges posed by cloud platforms. This dissertation shows that *with careful choice of design and features, it is possible to architect scalable DBMSs that efficiently support transactional semantics to ease application design and elastically adapt to fluctuating operational demands to optimize the operating cost.* This dissertation advances the state-of-the-art by improving two critical facets of transaction processing systems. First, we propose architectures and abstractions to support efficient and scalable transaction processing in DBMSs scaling-out using clusters of commodity servers. The key insight is to co-locate data items frequently accessed together within a database partition and limit transactions to access only a single partition. We propose systems where the partitions—the granules for efficient transactional access—can be statically defined based on the applications' access patterns or dynamically specified on-demand by the application. Second, we propose techniques to migrate database partitions in a live system to allow lightweight elastic load balancing, enable dynamic resource orchestration, and improve the overall resource utilization. The key insight is to leverage the semantics of the DBMS internals to migrate a partition with minimal disruption and performance overhead while ensuring the transactional guarantees and correctness even in the presence of failures. We propose two different techniques to migrate partitions in decoupled storage and shared nothing DBMS architectures.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"A journey of a thousand miles must begin with a single step."

– Lao Tzu.

1.1 Cloud Computing

Cloud computing has emerged as a successful and ubiquitous paradigm for service oriented computing where computing infrastructure and solutions are delivered as a service. The cloud has revolutionized the way computing infrastructure is abstracted and used. Analysts project the global cloud computing services revenue is worth hundreds of billion dollars and is growing [52]. The major features that make cloud computing an attractive service oriented architecture are: **elasticity**, i.e., the ability to scale the resources and capacity on-demand; **pay-per-use** pricing resulting in *low upfront investment* and *low time to market* for trying out novel application ideas; and the **transfer of risks** from the small application developers to the large infrastructure providers. Many novel application ideas can therefore be tried out with minimal risks, a model that was not economically feasible in the era of traditional enterprise infrastructures. This has resulted in large numbers of applications—of various types, sizes, and requirements—being deployed across the various cloud service providers.

Three cloud abstractions have gained popularity over the years. Infrastructure as a service (**IaaS**) is the lowest level of abstraction where raw compute infrastructure (such as CPU, memory, storage, network etc.) is provided as a service. Amazon web services (`http://aws.amazon.com/`), Rackspace (`http://www.rackspace.com/`) and GoGrid (`http://www.gogrid.com/`) are example IaaS providers. Platform as a service (**PaaS**) is the next higher level of service abstraction where an application deployment platform is provided as a service. Applications are custom built for a

PaaS provider's platform and the provider deploys, serves, and scales the applications. Microsoft Azure (`http://www.microsoft.com/windowsazure/`), Google AppEngine (`http://code.google.com/appengine/`), Force.com (`http://www.force.com/`), and Facebook's developer platform (`https://developers.facebook.com/`) are example PaaS providers. Software as a Service (**SaaS**) is the highest level of abstraction where a complete application is provided as a service. A SaaS provider typically offers a generic application software targeting a specific domain (such as customer relationship management, property management, payment processing and checkout, etc.) with the ability to support minor customizations to meet customer requirements. Salesforce.com (`http://www.salesforce.com/`), Google Apps for Business and Enterprises (`http://www.google.com/apps/intl/en/business/index.html`), and Microsoft Dynamics CRM (`http://crm.dynamics.com/en-us/home`) are example SaaS providers. The concept of service oriented computing abstractions can also be extended to Database as a Service, Storage as a Service, and many more.

1.2 Motivation and Challenges

Irrespective of the cloud abstraction, data is central to applications deployed in the cloud. Data drives knowledge which engenders innovation. Be it personalizing search results, recommending movies or friends, determining which advertisements to display or which coupon to deliver, data is central in improving customer satisfaction and providing a competitive edge. Data, therefore, generates wealth and many modern enterprises are collecting data at the most detailed level possible, resulting in massive and ever-growing data repositories. Database management systems (**DBMSs**) therefore form a critical component of the cloud software stack.

The data needs of applications deployed in the cloud can be broadly divided into two categories: (i) online transaction processing (**OLTP**) databases responsible for storing and serving the data and optimized for short low latency and high throughput transaction processing; and (ii) decision support systems (**DSS**) responsible for gleaning intelligence from the stored data and optimized for complex analysis over large amounts of data with often long-running queries. In this dissertation, we focus on OLTP DBMSs designed to be deployed on a pay-per-use cloud infrastructure.

A diverse class of applications rely on a back end OLTP database engine. These applications include social gaming (such as FarmVille, CityVille, and similar massively multi-player stateful online games), rich social media (such as FourSquare, Uber Sense, etc.), managed applications for various domains (such as Mint, Intuit, AppFolio, etc.), all the way up to the various cloud application platforms (such as Windows Azure, AppEngine, Force.com, Rightscale, etc.).

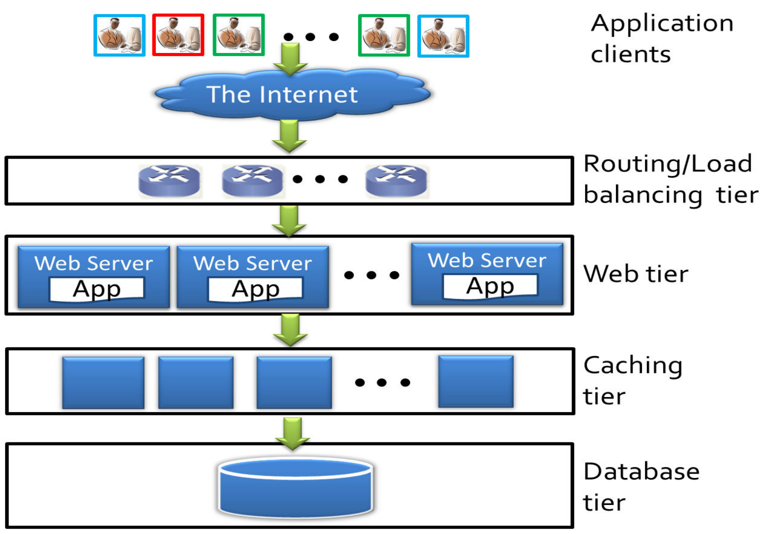


Figure 1.1: The classical software stack of a typical deployment of a web application. The clients connect through the Internet to the front tier servers (routing and web tiers) that interact with caching servers (optional, depending on performance requirements) and the database server(s) to serve client requests.

Relational database management systems (**RDBMSs**) are classical examples of OLTP systems; such systems include both commercial (such as Oracle RAC, IBM DB2 PureScale, Microsoft SQL Server, etc.) and open source (such as MySQL, Postgres, etc.) systems. These systems have been extremely successful in classical enterprise settings. Some of the key features of RDBMSs are: *rich functionality*, i.e., handling diverse application workloads using an intuitive relational data model and a declarative query language; *data consistency*, i.e., dealing with concurrent workloads while guaranteeing that data integrity is not lost; *high performance* by leveraging more than three decades of performance optimizations; and *high reliability and durability*, i.e., ensuring safety and persistence of data in the presence of different types of failures. Transactional access to data—i.e. guaranteeing *atomicity*, *consistency*, *isolation*, and *durability* (**ACID**) of data accesses—is one of the key features responsible for the widespread adoption of RDBMSs. In spite of the success of RDBMSs in classical enterprise infrastructures, they are often considered to be less *"cloud friendly"* [76]. This is because *scaling* the database layer *on-demand* while providing guarantees competitive with RDBMSs and ensuring high data availability in the presence of failures is a hard problem.

Consider a typical web application's software stack as shown in Figure 1.1. The clients access the application via the Internet through a tier of load balancers or proxy servers that eventually connect to the front-end tier comprising the web and application servers, i.e., the *web tier*. Data storage and querying is handled by the database servers in the *database tier*. Some frequent queries may be cached within the database tier and the web tier or, depending on the application's requirements, on a separate cluster of servers, i.e., the *caching tier*. Applications are typically deployed on commodity servers; servers in different tiers are interconnected via low latency and high throughput networks.

Most applications start on a small set of servers. As the number of clients and the request rates increase, every tier in the stack, except the database tier, can be scaled easily by adding more servers to distribute the load across a larger number of servers. Adding more database servers is, however, not as straightforward. This is primarily because the database servers store a lot of tightly coupled state while guaranteeing stringent properties and supporting concurrent access. Historically, there have been two approaches to scalability: scaling-up and scaling-out.

Scale-up, i.e., using larger and more powerful servers, has been the preferred approach to scale databases in enterprise infrastructures. This allowed RDBMSs to support a rich set of features and stringent guarantees without the need for expensive distributed synchronization. However, scaling-up is not viable in the cloud primarily because the cost of hardware grows non-linearly, thus failing to leverage the economies achieved from commodity servers.

Scale-out, i.e., increasing system's capacity by adding more (commodity) servers, is the preferred approach in the cloud. Scaling-out minimizes the total system cost by leveraging commodity hardware and the *pay-as-you-go* pricing. Scaling out RDBMSs, while supporting flexible functionality, is expensive due to distributed synchronization and the cost of data movement for transactions whose execution cannot be contained to a single node.[1] Moreover, managing partitioned RDBMS installations is a major engineering challenge with high administration costs [55].

In the absence of appropriate partitioning techniques or self-administered partitioned RDBMSs, many enterprises striving for large scale operations rely on a class of systems, called **Key-value stores**, that were designed to scale-out. Examples are Google's Bigtable [24], Yahoo!'s PNUTS [26], Amazon's Dynamo [35], and many other open source variants of the aforementioned systems. Data in all these systems is viewed as an independent collection of *key-value* pairs that are distributed over a cluster of servers. These Key-value stores were designed to scale to thousands of commodity servers, and replicate data across geographically remote locations, while supporting low latency and highly available accesses. These data stores, therefore, support simple

[1]We use the term **node** to represent a single server in a distributed system. These two terms, node and server, are used interchangeably throughout this dissertation.

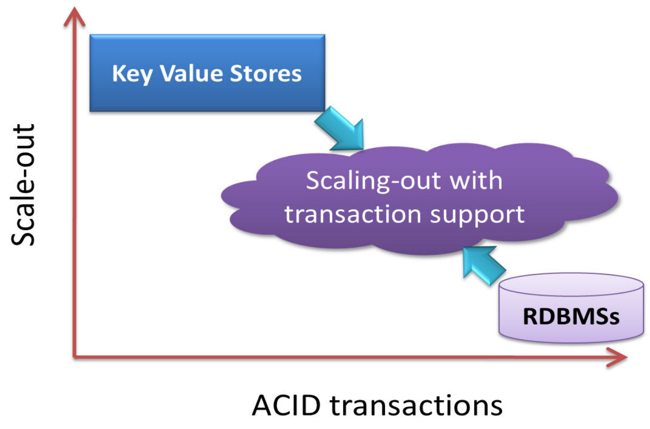


Figure 1.2: Scaling out while providing transaction access. Key-value stores are designed for large scale operations but support minimal transactional guarantees while RDBMSs support stringent transaction guarantees. This dissertation bridges this chasm.

functionality based on single-key operations. Most notable is the lack of transactional guarantees for accesses spanning multiple keys.

Single key accesses were enough to support the class of applications that the Key-value stores initially targeted [84]: web indexing for Bigtable [24], serving user profiles for PNUTS [26], and shopping cart maintenance for Dynamo [35]. However, as the class of applications using the Key-value stores diversified, applications accessing multiple data items within a single request was inevitable. The Key-value stores do not provide any atomicity and isolation guarantees for such accesses to multiple data items. In the absence of transactional guarantees, applications must either embrace the data inconsistencies, which considerably complicates reasoning about data freshness and application correctness [48], or implement transactional functionality in the application layer, a non-trivial and often inefficient implementation [72].

As a result, there exists a big chasm between RDBMSs that provide strong transactional guarantees but are hard to scale-out and Key-value stores that scale-out but support limited (or no) transactional semantics. Given this chasm, practitioners often resort to sub-optimal solutions, ad-hoc techniques, or retro-fitted features [48, 55, 72]. It is therefore critical to design DBMSs, either relational or key-value based, that *scale-out* while supporting *efficient transactional access*. Figure 1.2 depicts this balance between scale-out and transactional guarantees. Our goal is to combine the features of the Key-value stores and the RDBMSs to bridge the chasm.

In addition to scalability, the database tier must also be elastic, i.e., possess the ability to be *scaled on-demand*.[2] Elasticity allows the DBMSs to leverage variations in application's load patterns to consolidate to fewer servers during the period of low load and add more capacity when the load increases. This on-demand scaling minimizes the system's operating costs by leveraging the underlying pay-per-use cloud infrastructure. Of necessity, classical enterprise infrastructures were statically provisioned for peak expected load, so the focus was to efficiently utilize the available capacity and improve performance. However, with the advent of cloud computing, the need to optimize the system's operating cost has emerged as a critical design goal [41] and elasticity is a key contributor to cost minimization. **Lightweight**, i.e., low overhead, techniques for *elastic scalability* in DBMSs are, therefore, paramount.

Diverse applications are deployed in cloud infrastructures resulting in very different schemas, workload types, data access patterns, and resource requirements. Many of these applications (or **tenants**) are small in footprint and resource requirements. Allocating exclusive resources to these tenants is, therefore, wasteful in terms of the cost and the resource utilization. Sharing resources amongst multiple tenants, called **multitenancy**, allows effective resource utilization and further minimizes operating costs. When such a multitenant DBMS is deployed on a pay-per-use infrastructure, an added goal is to optimize the system's operating cost by aggressively consolidating the tenant databases while ensuring that the service level agreements (**SLA**) are met. Mechanisms to load balance tenant databases are therefore essential.

As the scale of a system grows, managing it is also challenging and expensive. For instance, detecting and recovering from failures, provisioning and capacity planning, and effective resource orchestration between the tenants are some common administration task in such systems. To further minimize the operating cost, it is also important to minimize the need for human intervention in system management.

Therefore, scalability, while supporting the guarantees application developers can reason about, and elasticity, with minimal overhead, are two fundamental requirements for the success of DBMSs in cloud infrastructures. These are in addition to the features common to any mission critical system, i.e., fault-tolerance, self-manageability, and high availability.

[2]There is a subtle difference between scalability and elasticity. Scalability is a *static* property that describes the system's ability to reach a certain scale (such as a thousand servers or a million requests per minute). Elasticity is a *dynamic* property that allows the system to scale *on-demand* (such as scaling to ten to hundred servers) in an operational system.

1.3 Dissertation Overview

The overarching goal of this dissertation is to propose abstractions, protocols, and paradigms to architect efficient and practical database management systems that meet the unique set of challenges encountered by DBMSs designed for cloud infrastructures. The underlying thesis of this dissertation is that with careful choice of design and features, *it is possible to architect scalable DBMSs that efficiently support transactional semantics to ease application design and elastically adapt to fluctuating operational demands to optimize the operating cost.* Using this principle as the cornerstone, this dissertation advances the state-of-the-art by improving two critical facets of OLTP DBMSs for the cloud. First, we propose architectures and abstractions to support efficient and scalable transaction processing in database systems spanning clusters of commodity servers. Second, we propose techniques to migrate databases for lightweight elastic load balancing in an operational system allowing the dynamic orchestration of system's resources.

1.3.1 Scale-out transaction processing

Classical RDBMSs allow complete flexibility in the scope and semantics of the transactions. For instance, an application can access the entire database within a single transaction and the RDBMS guarantees ACID properties on such a transaction. The RDBMSs abstracts the application's data as a cohesive granule of transactional access. As a result, when an RDBMS is scaled-out and distributed over a cluster of servers, the flexible transactional semantics lead to expensive distributed transactions. This makes efficiently scaling-out RDBMSs to large clusters a major challenge.

However, as the size of data grows, applications seldom access all the data within a single transaction; applications' accesses are typically localized to a small number of related data items. We propose to leverage this locality in the access patterns for access driven database partitioning where data items that are accessed together frequently are co-located within a single **database partition**. Co-locating data items based on access patterns allows us to support rich functionality while supporting transactional semantics at the granule of a database partition. Distributing the partitions on a cluster of servers and using the partitions as the granule of transactional access allows efficient transaction processing while scaling-out. Using this rationale, we present the design of two large partitioned DBMSs where the partitions can be statically or dynamically defined.

Many enterprise applications have static data access patterns. For such applications, rich functionality can be supported even when limiting most transactions to a single partition or node. Once the database is split into a set of partitions, the challenge in designing such a system lies in making it scalable, fault-tolerant, highly available, and self managing. In other words, the major research questions are: how to concurrently execute transactions on thousands of partitions, automatically detect and recover from node failures, dynamically maintain partition-to-node mappings in the presence of partition re-assignments, and guarantee correctness? We propose **ElasTraS** [31,32], an elastically scalable transaction processing system. The unique architectural aspect of ElasTraS is that it uses *logically* separate clusters of nodes for transaction processing and for data storage. In ElasTraS, the storage layer is abstracted as a scalable and fault-tolerant network-addressable storage. The transaction processing layer, executing on a cluster of nodes *logically* separate from the storage layer, focusses on efficient caching, transaction execution, and recovery of the partitions. Every transaction manager node has exclusive access to the data it is serving. ElasTraS, therefore, has a **decoupled storage** architecture which is different from the classical *shared storage* architectures where the storage layer is used for transaction synchronization. A self-managing system controller monitors nodes, detects failures, and automatically recovers from these failures. Stringent protocols guarantee strongly consistent and highly available management of system metadata.

In some applications, such as online games or collaboration-based applications, data items frequently accessed within a transaction change dynamically with time. For instance, in an online game, transactions frequently access player profiles participating in the same game instance which changes with time. In a statically partitioned database, profiles of players in a game may belong to different partitions. Transactions will, therefore, be distributed across multiple nodes resulting in inefficiencies and scalability bottlenecks. We propose the **Key Group** abstraction [33] that allows applications to dynamically specify the groups of data items (or keys) on which it will require transactional access. To allow efficient execution of transactions on a group, we propose the **Key Grouping protocol**, which co-locates read-write access of keys in a Key Group at a single node. The Key Group abstraction leverages the application semantics and is equivalent to a dynamically formed database partition. The group creation and deletion cost is paid for by more efficient transaction execution during the lifetime of the group. The challenges to implement transactions in such a system are: how to ensure the safe formation and deletion of groups in the presence of failures, how to ensure efficient and durable transaction execution, and how to guarantee safe maintenance of system metadata corresponding to the dynamically formed groups? We propose rigorous protocols for group formation, transaction execution, and recovery.

1.3.2 Lightweight elasticity

Elasticity in the database tier entails dynamically adapting the size of the database cluster depending on the workload's demands. In other words, when the load on the database increases, the system should be able to add new capacity and move some database partitions to the newly added servers to distribute the load, thus ensuring good performance. Similarly, when the load decreases, the system should also be able to consolidate the database partitions to fewer servers, thus reducing the operating cost. The ability to migrate database partitions in a live system executing transactions, with low overhead and minimal impact on the transactions executing during the migration, is critical to elastic load balancing and dynamic resource orchestration.

In this dissertation, we formulate the problem of **live database migration** for elastic scaling and load balancing in the database tier. Live migration enables a database partition to be migrated from a source node to a destination node while clients continue to execute transactions on the partition. The major challenges for live database migration are how to minimize the migration overhead, how to ensure serializable isolation of the transactions, and how to ensure correctness in the presence of failures that might occur during migration? We propose two live database migration techniques for two commonly used database architectures: decoupled storage and classical shared nothing.

In the decoupled storage architecture, the persistent data is stored in a network-addressable storage abstraction accessible from all the database nodes. Therefore, the partition's persistent image need not be migrated. An added challenge for this architecture is the live migration of the database cache and the state of active transactions. We propose a new protocol, **Albatross** [34], that migrates a partition with no aborted transactions and minimal performance impact. In Albatross, the source takes a quick snapshot of a partition's cache and the destination warms up its cache starting with this snapshot. While the destination initializes its cache, the source continues executing transactions; the destination therefore lags the source. Changes made to the source node's cache are iteratively copied to the destination. Once the destination has sufficiently caught up, transactions are blocked at the source and migration is completed. The state of active transactions is copied in this final handover phase.

In a shared nothing database architecture, the persistent data is stored on disks locally attached to the nodes. An added challenge for this architecture is the live migration of the persistent data (which can be in the order of gigabytes) with no unavailability. We propose a new protocol, **Zephyr** [39], that divides migration into multiple phases. In the first phase, Zephyr freezes all the indices, i.e., prevents structural changes, and copies a wireframe of the database to the destination node. This wireframe consists of the minimal information needed for the destination to start executing transactions but does not include the actual application data stored in the partition. Once the destination initializes the wireframe, migration enters the second phase where both the source and the destination nodes concurrently execute transactions on the partition. The source completes execution of the transactions that were active at the start of migration, while the destination executes new transactions. Database pages are used as granules for

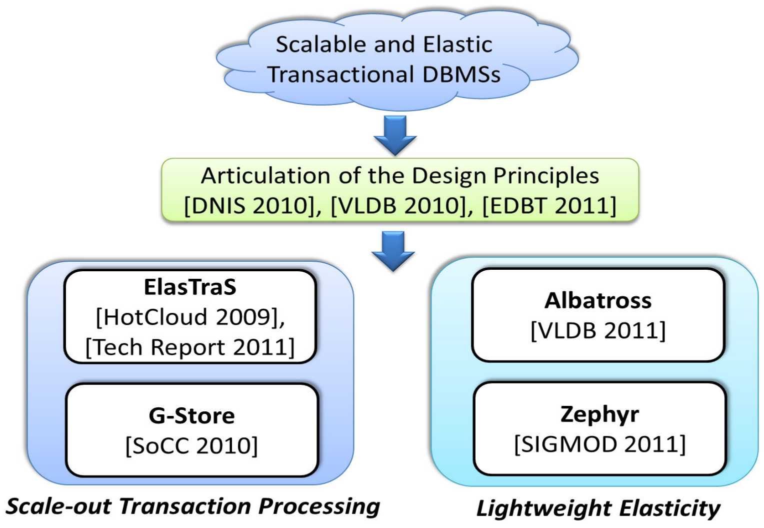

Scalable and Elastic Transactional DBMSs
Articulation of the Design Principles [DNIS 2010], [VLDB 2010], [EDBT 2011]
ElasTraS [HotCloud 2009], [Tech Report 2011]
G-Store [SoCC 2010]
Albatross [VLDB 2011]
Zephyr [SIGMOD 2011]
Scale-out Transaction Processing
Lightweight Elasticity

Figure 1.3: Overview of the dissertation's contributions classified into the two thrust areas for this dissertation: scale-out transaction processing and lightweight elasticity.

migration; pages are pulled from the source by the destination as transactions at the destination access them. Once transactions at the source complete, migration enters the final phase where the remaining pages are pushed to the destination. Minimal synchronization and handshaking between the source and the destination ensures correctness.

1.4 Contributions and Impact

This dissertation makes several fundamental contributions towards realizing our vision of building scalable and elastic OLTP DBMSs for cloud platforms. Our contributions significantly advance the state-of-the-art by supporting *scale-out transaction processing* and promoting *elasticity* as a first class concept in database systems. Our technical contributions are in access driven and dynamically specified database partitioning, large self-managing DBMSs installations, virtualization in the database tier, and live database migration for elastic load balancing. These technologies are critical to ensure the success of the next generation of DBMSs in cloud computing infrastructures.

Figure 1.3 summarizes these contributions into the two major thrust areas of this dissertation: scale-out transaction processing and lightweight elasticity. We now highlight these contributions and their impact.

- We present a thorough analysis of the state-of-the-art systems and distill the important aspects in the design of different systems and analyze their applicability and scope [2, 3, 5]. We then articulate some basic design principles for designing new database systems for the cloud. A thorough understanding and a precise characterization of the design space are essential to carry forward the lessons learned from the rich literature in scalable and distributed database management.

- We propose access driven database partitioning and self-administering DBMS installations designed for scaling out using commodity servers. Leveraging access patterns to drive database partitioning and co-locate data items is fundamental to support rich transactional semantics and efficient transaction execution. Minimizing the need for human intervention in common system management and administration tasks is important to reduce the overall operating cost in addition to facilitating adoption.
- We present the architecture and implementation of ElasTraS [31, 32], a design to efficiently execute transactions on a set of partitions distributed across a cluster of commodity servers. ElasTraS is one of the first systems to allow scale-out transaction processing on statically partitioned databases. We present mechanisms for automated system management to detect and recover from failures and load balance the partitions depending on changes in the load patterns.
- We present the Key Group abstraction [33], a mechanism for applications to dynamically specify a set of data items on which it requires transactional access. We also present the Key Grouping protocol that is a lightweight mechanism to dynamically reorganize the read/write access rights to the data items in a distributed data management system, thus allowing efficient transaction execution. This is the first technique to provide the benefits of partitioning even when accesses do not statically partition. We demonstrate the practicality of the proposed Key Grouping protocol by implementing a prototype, called **G-Store**, that guarantees transactional multi-key access over Key-value stores.
- We formulate the problem of live database migration as an important feature to build elastic DBMSs for the cloud. We envision embedding virtualization into the database tier in order to effectively support database multitenancy. Live database migration is a critical enabler for dynamic resource orchestration between the tenant databases in virtualized multitenant DBMSs.
- We present Albatross [34], the first published end-to-end solution for live migration in decoupled storage database architectures with no transaction aborts during migration. Albatross migrates the database cache and the state of active transactions to ensure minimal performance impact as a result of migration.
- We present Zephyr [39], the first published end-to-end solution for live migration in shared nothing database architectures with no downtime as a result of migration. Zephyr guarantees no downtime by allowing the source and the destination of migration to concurrently execute transactions during a phase in migration. We

demonstrate how a lightweight synchronization mechanism can be used in such a scenario to guarantee serializability of transactions executing during migration.
- We present a rigorous analysis of the guarantees and correctness in the presence of various failure scenarios for each of the four proposed systems and techniques. Such a rigorous analysis is important to reason about overall correctness while providing confidence in the system's behavior. In addition, all four techniques have been prototyped in real distributed database systems to demonstrate feasibility and the benefits of the proposed techniques. A detailed analysis of the trade-offs of each design allows future systems to make informed decisions based on insights from this dissertation.

1.5 Organization

In Chapter 2, we provide a systematic survey and analysis of the state-of-the-art in scalable and distributed database systems. The rest of the dissertation is organized into two parts focussing on the two thrust areas of this dissertation.

Part I focusses on systems designed to support efficient transaction processing while scaling-out to large clusters of commodity servers. Chapter 3 presents the design principles that form the foundation of the two systems described in the two subsequent chapters. Chapter 4 presents the design of ElasTraS, a system that scales-out using statically defined database partitions. Chapter 5 presents the Key Grouping protocol, a technique to efficiently execute transactions on dynamically defined partitions.

Part II focusses on techniques to enable lightweight elasticity using live database migration. Chapter 6 analyzes the problem space for elastic load balancing in the database tier and formulates the problem of live database migration. Chapter 7 presents Albatross, a technique for lightweight database migration in a decoupled storage database architecture where the persistent data is stored in a network-addressable storage abstraction. Chapter 8 presents Zephyr, a technique for lightweight database migration in shared nothing database architectures where the persistent data is stored on disks locally attached to the nodes.

Chapter 9 concludes this dissertation and outlines some open challenges.

Chapter 2

State of the Art

"Stand on the shoulders of giants."

– Bernard of Chartres and Isaac Newton.

Scalable distributed data management has been the vision of the computer science research community for more than three decades. This chapter surveys the related works in this area in light of the cloud infrastructures and their requirements. Our goal is to distill the key concepts and analyze their applicability and scope. A thorough understanding and a precise characterization of the design space are essential to carry forward the lessons learned from the rich literature in scalable and distributed database management.

2.1 History of Distributed Database Systems

Early efforts targeting the design space of scalable database systems resulted in two different types of systems: distributed DBMSs (**DDBMS**) such as R^* [66] and SDD-1 [77] and parallel DBMSs (**PDBMS**) such as Gamma [37] and Grace [42]. DeWitt and Gray [36] and Ozsu and Valduriez [73] provide thorough surveys of the design space, principles, and properties of these systems. The goal of both classes of systems was to distribute data and processing over a set of database servers while providing the abstractions and semantics similar to centralized systems. While the DDBMSs were designed for update intensive workloads, the PDBMSs allowed updates but were predominantly used for analytical workloads. The goal of DDBMSs was to continue providing the semantics of a centralized DBMS while the DBMS nodes can be distributed either within an enterprise or potentially across multiple enterprises and inter-connected using wide area networks.

Even though PDBMSs have been widely commercialized, the success of DDBMSs was limited primarily due to the overhead of distributed transactions. First, being distributed across high latency networks meant considerably higher response times for transactions spanning multiple database servers. Second, guaranteeing stringent transactional properties in the presence of failures limited the availability of these systems. In addition, a number of other technical and non-technical reasons also limited the practicality of these distributed DBMSs. As a result, enterprises resorted to scaling-up the DBMSs while using workflows and business processes [15] to replace complex distributed transactions.

In spite of the limited success, many important technologies were invented while building the DDBMSs and PDBMSs that remain relevant in the current context. An example is the two phase commit protocol (**2PC**) for atomic transaction commitment across multiple sites [44] which is commonly used in many systems currently used in production. Many important advances were also made in the distributed systems community. For instance, the Paxos consensus protocols [63], notions of causality and vector clocks [62], and fault-tolerant broadcast protocols [16, 47] are fundamental technologies that have influenced the design of contemporary systems.

Different from the distributed and parallel DBMSs, another approach to scaling DBMSs while preserving the semantics of a single node RDBMS is through **data sharing**. In such a model, a common database storage is shared by multiple processors that concurrently execute transactions on the shared data. Examples of such systems are Oracle Real Application Clusters [23], Oracle Rdb (formerly DEC Rdb) [68], and IBM DB2 data sharing [57]. A common aspect of all these designs is a shared lock manager responsible for concurrency control. Even though many commercial systems based on this architecture are still used in production, the scalability of such systems is limited by the shared lock manager and the complex recovery mechanisms resulting in longer unavailability periods as a result of a failure.

2.2 Cloud Data Management: Early Trends

With the growing popularity of the Internet, many applications were delivered over the Internet and the scale of these applications also increased rapidly. As a result, many Internet companies, such as Google, Yahoo!, and Amazon, faced the challenge of serving hundreds of thousands to millions of concurrent users. Classical RDBMS technologies could not scale to these workloads while using commodity hardware to be cost-effective. The need for low cost scalable DBMSs resulted in the advent of Key-value stores such as Google's Bigtable [24], Yahoo!'s PNUTS [26], and Amazon's

Dynamo [35].[1] These systems were designed to scale out to thousands of commodity servers, replicate data across geographically remote data centers, and ensure high availability of user data in the presence of failures which is the norm in such large infrastructures of commodity hardware. These requirements were a higher priority for the designers of the Key-value stores than rich functionality. Key-value stores support a simple *key-value* based data model and single key access guarantees, which were enough for their initial target applications [84]. In this section, we discuss the design of these three systems and analyze the implications of the various design choices made by these systems.

2.2.1 Key-value Stores

Bigtable [24] was designed to support Google's crawl and indexing infrastructure. A Bigtable cluster consists of a set of servers that serve the data; each such server (called a *tablet server*) is responsible for parts of the tables (known as a *tablet*). A tablet is logically represented as a key range and physically represented as a set of *SSTables*. A tablet is the unit of distribution and load balancing. At most one tablet server has read and write access to each tablet. Data from the tables is persistently stored in the Google File System (GFS) [43] which provides the abstraction of scalable, consistent, fault-tolerant storage. There is no replication of user data inside Bigtable; all replication is handled by the underlying GFS layer. Coordination and synchronization between the tablet servers and metadata management is handled by a *master* and a Chubby cluster [20]. Chubby provides the abstraction of a synchronization service via exclusive timed leases. Chubby guarantees fault-tolerance through log-based replication and consistency amongst the replicas is guaranteed through a Paxos protocol [22]. The Paxos protocol [63] guarantees safety in the presence of different types of failures and ensures that the replicas are all consistent even when some replicas fail. But the high consistency comes at a cost: the limited scalability of Chubby due to the high cost of the Paxos protocol. Bigtable, therefore, limits interactions with Chubby to only the metadata operations.

PNUTS [26] was designed by Yahoo! with the goal of providing efficient read access to geographically distributed clients. Data organization in PNUTS is also in terms of range-partitions tables. PNUTS performs explicit replication across different data centers. This replication is handled by a guaranteed ordered delivery publish/subscribe system called the Yahoo! Message Broker (**YMB**). PNUTS uses per record mastering and the master is responsible for processing the updates; the master is the publisher to

[1] At the time of writing, various other Key-value stores (such as HBase, Cassandra, Voldemort, MongoDB etc.) exist in the open-source domain. However, most of these systems are variants of the three in-house systems.

YMB and the replicas are the subscribers. An update is first published to the YMB associated to the record's master. YMB ensures that updates to a record are delivered to the replicas in the order they were executed at the master, thus guaranteeing *single object time line consistency*. PNUTS allows clients to specify the freshness requirements for reads. A read that does not have freshness constraints can be satisfied from any replica copy. Any read request that requires data that is more up-to-date than that of a local replica must be forwarded to the master.

Dynamo [35] was designed by Amazon to support the shopping carts for Amazon's e-commerce business. In addition to scalability, high write availability, even in the presence of network partitions, is a key requirement for Amazon's shopping cart application. Dynamo therefore explicitly replicates data and a write request can be processed by any of the replicas. It uses a quorum of servers for serving the read and writes. A write request is acknowledged to the client when a quorum of replicas has acknowledged the write. To support high availability, the write quorum size can be set to one. Since updates are propagated asynchronously without any ordering guarantees, Dynamo only supports eventual replica consistency [85] with the possibility that the replicas might diverge. Dynamo relies on application level reconciliation based on vector clocks [62].

2.2.2 Design Choices and their Implications

Even though all three Key-value stores share some common goals, they also differ in some fundamental aspects of their designs. We now discuss these differences, the rationale for these decisions, and their implications. We focus on the design aspects; Cooper et al. [27] discusses the performance implications.

Data model

The distinguishing feature of the Key-value stores is their simple data model. The primary abstraction is a table of items where each item is a *key-value* pair or a row. The value can either have structure (as in Bigtable and PNUTS), or can be an uninterpreted string or blob (as in Dynamo). Bigtable's data model is a sparse multi-dimensional sorted map where a single data item is identified by a row identifier, a column family, a column, and a timestamp. The column families are the unit of data co-location at the storage layer. PNUTS provides a more traditional flat row-like structure similar to the relational model. Atomicity and isolation are supported at the granularity of a single key-value pair, i.e., an atomic read-modify-write operation is supported only for individual key-value pairs. Accesses spanning multiple key-value pairs are best-effort without guaranteed atomicity and isolation from concurrent accesses. These systems allow large rows, thus allowing a logical entity to be represented as a single row.

Restricting data accesses to a *single-key* provides designers the flexibility of operating at a much finer granularity. Since a single key-value pair is never split across compute nodes, application level data manipulation is restricted to a single compute node boundary and thus obviates the need for multi-node coordination and synchronization [51]. As a result, these systems can scale to billions of key-value pairs using horizontal partitioning. The rationale is that even though there can be potentially millions of requests, the requests are generally distributed throughout the data set. Moreover, the single key operation semantics limits the impact of failure to only the data that was being served by the failed node; the rest of the nodes in the system can continue to serve requests. Furthermore, single-key operation semantics allows fine-grained partitioning and load-balancing. This is different from RDBMSs that consider data as a cohesive whole and a failure in one component results in overall system unavailability.

Data distribution and request routing

All the systems partition data to distribute it over a cluster of servers. Bigtable supports range partitioning, Dynamo supports hash partitioning, and PNUTS supports both hash and range partitioning. The systems also require a routing mechanism to determine which node is serving a particular key-value pair. Bigtable and PNUTS use centralized solutions to maintain the mapping of key intervals to the servers. Bigtable uses a three level B+-tree (called the `ROOT` and `META` tables) that stores the interval mapping. PNUTS stores the mappings on specialized servers, called *tablet controllers*, dedicated for this purpose. Dynamo, on the other hand, uses a distributed peer-to-peer (**P2P**) approach using consistent hashing [59]. Due to the centralized nature of the system metadata in Bigtable and PNUTS, specialized mechanisms are needed to route the client requests: Bigtable uses a client library that encapsulates the routing logic while PNUTS uses a separate set of routing servers that cache the interval mappings. In Dynamo, the clients use consistent hashing to route the client requests, thus obviating the need for explicit routing mechanisms.

Cluster management

As the scale of the system increases, managing such systems without human intervention becomes a challenge. Specifically, detecting and recovering from failures and basic load balancing functionalities are critical to the system's proper operation. In Bigtable, the master server, in close cooperation with the Chubby service, performs cluster management. The master and every tablet server in the system obtains a timed lease with Chubby that must be periodically renewed. A server in a Bigtable cluster can carry out its responsibilities only if it has an active lease from Chubby. Every tablet server periodically reports to the master using heartbeat messages that also contain the load statistics. These heartbeat messages and the leases with Chubby form the basis for failure detection and recovery. PNUTS also relies on a similar mechanism where the tablet controller detects failures as well as load balances the tablets. Dynamo, on the other hand, relies on P2P techniques. It uses gossip-based protocols to detect failures and sloppy quorum and hinted handoff mechanisms to deal with temporary failures. The somewhat centralized nature of the control mechanisms in Bigtable and PNUTS make them vulnerable to unavailability in the presence of failures of these critical components. For instance, if the Chubby service in Bigtable or the tablet controller in PNUTS are unavailable, the system becomes unavailable. The decentralized P2P approaches in Dynamo allow the system to be less susceptible to such failures and unavailability.

Fault-tolerance and data replication

Failures are common at the scale of operation the Key-value stores target. They were, therefore, designed to handle failures gracefully to ensure high data availability. In Bigtable, data replication is handled in the underlying storage layer, i.e., by GFS. Bigtable uses GFS as a strongly consistent replicated storage abstraction. The persistent data and the write ahead logs are stored in GFS, thus allowing Bigtable to recover from tablet server failures. After a tablet server failure is detected by the master, the state of the failed server can be recovered at another live tablet server. The GFS design, however, is optimized for replication within a data center. As a result, a large scale data center level outage results in data unavailability in Bigtable.

PNUTS uses the YMB for replication and fault-tolerance. An update is acknowledged to the client only after it has been replicated within YMB. PNUTS uses YMB as a fault-tolerant replicated log and leverages YMB's guaranteed ordered delivery for replication; PNUTS guarantees single record timeline consistency for the replicas. If a record's master fails, another replica can be elected as the master once all updates from the original master's YMB have been applied to the replica, a mechanism called *re-mastering*. However, in the event of a data center outage resulting in YMB unavailability, similar to any asynchronous log-based replication protocol, the tail of the log that has not propagated to other data centers will be lost if the records are re-mastered. Therefore, PNUTS presents a trade-off between data loss and data unavailability in the event of a catastrophic failure.

Dynamo uses asynchronous quorum based replication where any replica can process an update. Freshness guarantees can be provided if the read and write quorums overlap; though at the cost of increased latency. However, to guarantee high availability, Dynamo uses *sloppy quorums*, i.e., the read and write quorums do not overlap. Dynamo does not provide any update propagation guarantees, thus only guaranteeing eventual replica consistency. Since any replica can handle updates, failure handling is straightforward. As noted earlier, failures can lead to divergent replicas whose reconciliation must be handled at the application layer.

For these Key-value stores, scalability and high availability are the foremost requirements. It is well known that a distributed system can only choose two of consistency, availability, and partition tolerance (**CAP**) [19]. For these systems spanning large infrastructures or geographically separated data centers, network partitions are inevitable. In the event of a network partition, these systems choose availability over replica consistency.

2.3 Transaction Support in the Cloud

A large class of web-applications exhibit *single key* access patterns which drove the design of the Key-value stores [35,84]. However, as the class of applications broadened, applications often access multiple data items within a single request. These applications range from online social applications such as multi-player games and collaboration based applications to enterprise class applications deployed in the cloud. Providing transactional support in the cloud has therefore been an active area of research.

Classical multi-step transactions guarantee the ACID semantics: either all operations in a transaction execute or none of them execute (*atomicity*), a transaction takes the database from one consistent state to another consistent state (*consistency*), concurrently executing transactions do not interfere (*isolation*), and updates made from a committed transaction are persistent (*durability*) (**ACID**). **Serializability**, the strongest form of isolation, ensures that transactions execute in an order equivalent to serial order, thus making concurrency transparent to the application developers. The ACID semantics, therefore, considerably simplifies application logic and helps reasoning about correctness and data consistency using sequential execution semantics.

In addition to the two approaches that this dissertation presents, various other approaches were also developed concurrently in both academia and industry. In this section we provide a survey of this evolving landscape of scalable transaction processing in the cloud. We classify these approaches into two broad categories: approaches that rely of database partitioning to limit most transactions to a single server and approaches that do not partition the database.

2.3.1 Partitioning based approaches

We first discuss the class of systems that provide ACID guarantees by limiting the scope of the transactions supported. These systems are driven by a common observation that application access patterns can be partitioned so that data items frequently accessed together within a transaction can be co-located within a database partition. The systems however differ in how the database is partitioned and how the partitioned database is served.

Cloud SQL Server [14] from Microsoft uses a partitioned database on a shared-nothing architecture leveraging an RDBMS engine, Microsoft SQL Server, at its core. Cloud SQL server is the underlying storage abstraction for Microsoft SQL Azure and Exchange Hosted Archive. Transactions are constrained to execute on one partition and the database is replicated for high availability using a custom primary-copy replication scheme. In Cloud SQL Server, a logical database is called a *table group* which may be keyless or keyed. If it is keyed, then all of the tables in the table group must have a common column called the partitioning key. A *row group* is the set of all rows in a table group that have the same partition key value. For keyless tables, ACID guarantees are provided on a table group while for keyed tables, acid guarantees are provided on a row group. The transaction commitment protocol requires that only a quorum of replicas acknowledge a transaction commit. A Paxos-like consensus algorithm is used to maintain a set of replicas to deal with replica failures and recoveries. Dynamic quorums are used to improve availability in the face of multiple failures.

Megastore [11], the database supporting Google AppEngine, provides transactional multi-key accesses on top of Bigtable. Megastore provides both strong consistency guarantees and high availability of data supporting fully serializable ACID semantics within fine-grained partitions of data. It uses a statically defined abstraction, called *entity-groups*, that represents the application specified granule of transactional access as well as partitioning. An entity group is essentially a hierarchical key structure. An important aspect of the design of Megastore is the use of a Paxos-like protocol for synchronous replication of each write to an entity group across a wide area network with reasonable latency. This synchronous cross data center replication allows seamless fail-over between data centers but comes at the cost of increased transaction latencies.

Relational Cloud [28] also proposes a design with similar goals as the above systems and hence various aspects of the design reflect choices similar to that of the other systems. The key aspects of Relational Cloud include a workload-aware approach to multi-tenancy that identifies the workloads that can be co-located on a database server [29] and the use of a graph-based data partitioning algorithm [30].

The three systems described above, and ElasTraS presented in this dissertation, have similar goals and hence share various common design decisions such as: limiting interactions to a single node, limited and prudent use of distributed synchronization only when needed and so on. In spite of the similarity, there are some key differences that distinguish these systems from one another. First, ElasTraS, Cloud SQL Server, and Megastore use variants of a hierarchical schema pattern while Relational Cloud uses a custom graph-based database partitioning mechanism. Second, Cloud SQL Server and Relational Cloud use the classical shared nothing architecture where the persistent data is stored in locally attached disks. As a result, to ensure high availability, Cloud SQL Server has a custom replication protocol. On the other hand, ElasTraS and Megastore use a *decoupled* storage architecture where the transaction execution logic is *decoupled* from the storage logic and the persistent data is stored in a fault-tolerant shared network addressable storage abstraction. On top of the fault-tolerant storage, Megastore uses its own custom wide-area replication mechanism to support seamless data center fail-over for disaster recovery.

As noted earlier, the design of the systems described above is based on the assumption that application accesses partition statically. For applications who access patterns change dynamically, using such a statically partitioned system would result in distributed transactions. No technique is known that leverages the benefits of partitioning when accesses do not statically partition. We present the first such technique. Our technique allows applications to dynamically specify the set of data items on which transactional access is sought while allowing the system to co-locate accesses to these data items to provide performance similar to the statically partitioned systems.

2.3.2 Approaches without explicit partitioning

A different class of systems do not require the database to be partitioned. As a result, transactions span multiple servers, thus making them expensive. These systems either provide weaker transactional guarantees or leverage application semantics or custom optimizations to minimize the cost.

Brantner et al. [18] propose techniques to build higher level database functionality on top of storage utility services in the cloud, such as Amazon S3, which exposes an interface similar to Key-value stores. The proposed system's goal is to preserve the scalability and availability of a distributed system such as S3. In order to allow for high availability, the authors focus on maximizing the level of consistency that can be achieved without providing stringent consistency and isolation guarantees as supported by transactions in classical RDBMSs. Rather, the focus is on supporting atomicity and durability of a single object stored in S3. Since stronger guarantees make operations expensive, Kraska et al. [60] propose a mechanism allowing applications to declare data items with variable consistency. The rationale of this approach, called *consistency rationing*, is to pay the cost of strong guarantees only for the data items that need strong consistency.

Percolator [74] from Google targets an application domain where data co-location is not possible. As a result, Percolator must use distributed transactions. The focus is on the ACID properties of transactions and not on the low latency requirements of these transactions. The primary target application of Percolator is the indexing infrastructure of Google that allows incremental index maintenance. When a new web-page is crawled, it results in an update to multiple parts of the index that must be transactionally updated. Such transactions, however, do not have stringent latency requirements. As a result, percolator uses proven protocols for distributed transaction execution over a key-value store such as Bigtable. Some transactions might block in the presence of failures. Such blocking is acceptable in Percolator's target application domain that does not require low latency transactions.

Aguilera et al. [7] presents Sinfonia which is an efficient platform for building distributed systems. Sinfonia can be used to efficiently design and implement systems such as distributed file systems. The metadata of such systems (e.g. the inodes) need to be maintained as well as manipulated in a distributed setting; Sinfonia provides efficient means for guaranteeing the consistency of these critical operations. It provides the *minitransaction* abstraction that guarantees transactional semantics on only a small set of operations such as *atomic* and distributed *compare-and-swap*. The idea is to use the two phases of message exchange in 2PC to execute some simple operations. The operations are piggy-backed on the messages sent out during the first phase of 2PC. Operations should be such that each participating site can perform the operation and reply with a commit or abort vote. The lightweight nature of a *minitransaction* allows the system to scale to hundreds of nodes.

Vo et al. [83] propose ecStore, an elastic cloud storage system that supports automated data partitioning and replication while supporting transactional access. In ecStore, data objects are distributed and replicated in a cluster of commodity nodes in the cloud. Transactional access to multiple data items is provided through the execution of distributed transactions; the focus is primarily on atomicity and durability with weaker isolation levels supported through optimistic multi-version concurrency control.

The Deuteronomy system [65] supports efficient and scalable ACID transactions in the cloud by decomposing the functions of a database storage engine kernel into a transactional component (TC) and a data component (DC). The TC manages transactions and their logical concurrency control and recovery, but knows nothing about physical data location. The DC maintains a data cache and uses access methods to support a record-oriented interface with atomic operations, but knows nothing about transactions. The key idea of Deuteronomy is that the TC can be applied to data anywhere (in the cloud, local, etc.) with a variety of deployments for both the TC and DC. Since the TC and DC are completely decoupled, the focus of the system is on the design of the interface protocol, efficient locking, and efficient recovery.

Hyder [13] aims at providing scale-out without the need to partition the application or the database while allowing high throughput and low latency transaction execution. Hyder proposes a data sharing architecture, radically different from classical data sharing architectures [23, 68], that leverages novel advances in the overall hardware infrastructure in data centers: low latency and high throughput networks, abundant I/O operations in new storage-class memories (such as flash memory), and powerful multicore processors. Hyder supports reads and writes on indexed records within classical multi-step transactions. It is designed to run on a cluster of servers that have shared access to a large pool of network-addressable raw flash memory chips. The key aspects in Hyder's design are the shared log that is also the database, the use of multi-version optimistic concurrency control for transaction execution, and an efficient algorithm for deterministic conflict detection for the optimistic execution of transactions.

2.4 Multitenant Database Systems

In addition to supporting transactional semantics, with the growing number of applications being deployed in the cloud, another important requirement is the support of multitenancy in the database tier. As the number of applications deployed in the cloud grows, so does the number of databases. Many of these databases are small and it is imperative that multiple small databases share the system resources, a model commonly referred to as multitenancy. Classical examples of multitenancy are in the SaaS paradigm where multiple different customizations of the same application share an underlying multitenant database. However, as a wider class of applications is now deployed on shared cloud platforms, various other multitenancy models are also being explored to meet varying application requirements. In this section, we review some commonly used multitenancy models and analyze the trade-off in light of virtualization in the database tier.

2.4.1 Multitenancy Models

Sharing resources at different levels of abstraction and distinct isolation levels results in different multitenancy models in the database tier.[2] The three models explored in the past [56] consist of: **shared hardware**, **shared process**, and **shared table**. SaaS providers, such as Salesforce.com [88], typically use the shared table model. The shared process model is used in a number of database systems for the cloud, such as RelationalCloud [28], Cloud SQL Server [14], and ElasTraS [31]. Soror et al. [78] and Xiong et al. [89] propose systems using the *shared machine* model. Figure 2.1 depicts the three multitenancy models and the level of sharing.

[2]The term isolation in the context of multitenancy refers to performance isolation or access control isolation between tenants sharing the same multitenant DBMS. This is different from the use of isolation in the context of concurrent transactions.

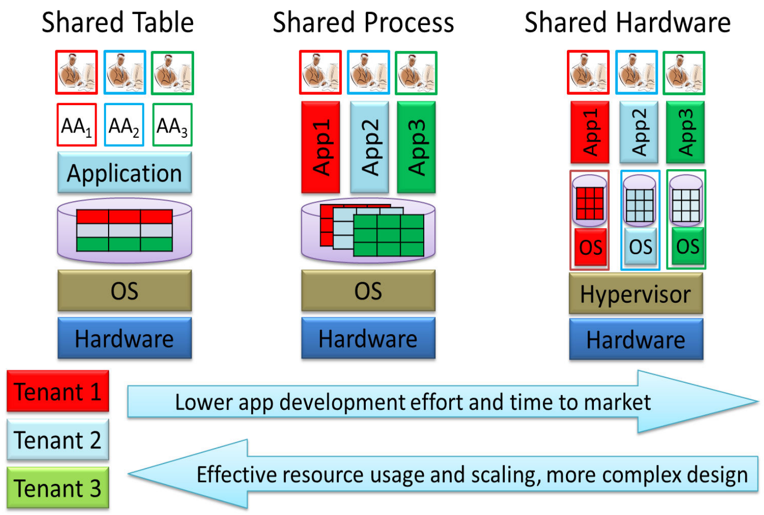


Figure 2.1: The different multitenancy models and their associated trade-offs.

Shared Hardware

In this model, tenants share resources at the same server using multiple VMs. Each tenant is assigned its own VM and an exclusive database process that serves the tenant's database. A virtual machine monitor or a hypervisor orchestrates resource sharing between the co-located VMs. While this model offers strong isolation between tenants, it comes at the cost of increased overhead due to redundant components and a lack of coordination using limited machine resources in a non-optimal way. Consider the instance of disk sharing between the tenants. A VM provides an abstraction of a virtualized disk which might be shared by multiple VMs co-located at the same node. The co-located database processes make un-coordinated accesses to the disk. This results in high contention for the disk that can considerably impact performance in a consolidated setting. A recent experimental study by Curino et al. [29] shows that this performance overhead can be up to an order of magnitude. This model might therefore be useful when only a small number of tenants are executing at any server. In this case, multitenancy can be supported without any changes in the database layer.

Shared Process

In this model, tenants share resources within a single database process running at each server. This sharing can happen at various isolation levels—from sharing only some database resources such as the logging infrastructure, to sharing all resources such as the buffer pool, transaction manager etc. This model allows for effective resource sharing between the tenants while allowing the database to intelligently managing some critical resources such as the disk bandwidth. This will allow more tenants to be consolidated at a single server while ensuring good performance. Tenant isolation is typically provided using authentication credentials.

Shared Table

In the shared table model, the tenants' data is stored in a shared table called the *heap table*. To support flexibility in schema and data types across the different tenants, the heap table does not contain the tenant's schema or column information. Additional metadata structures, such as *pivot tables* [9, 88], provide the rich database functionality such as the relational schema, indexing, key constraints etc. The reliance on consolidated and specialized pivot and heap tables implies re-architecting the query processing and execution functionality, in addition to performance implications due to low tenant isolation. Additionally, the shared table model requires that all tenants reside on the same database engine and release (or version). This limits specialized database functionality, such as spatial or object based, and requires that all tenants use a limited subset of functionality. This multitenancy model is ideal when multiple tenants have similar schema and access patterns with minimal customizations, thus providing effective sharing of resources. Such similarity is observed in SaaS where a generic application tenant is customized to meet specific customer requirements.

2.4.2 Analyzing the models

The different multitenancy models provide different trade-offs; Figure 2.1 depicts some of these trade-offs as we move from the shared hardware model to the shared table model. At one extreme, the *shared hardware* model uses virtualization to multiplex multiple VMs on the same machine. Each VM has only a single database process serving the database of a single tenant. As mentioned earlier, this strong tenant isolation comes at the cost of reduced performance [29]. At the other extreme is the *shared table* model which stores multiple tenants' data on shared tables and provides the least amount of isolation, which in-turn requires changes to the database engine while limiting schema flexibility across the tenants. The shared process model allows independent schemas for tenants while sharing the database process amongst multiple tenants, thus providing better isolation compared to the shared table model while allowing effective sharing and consolidation of multiple tenants in the same database process. The shared process model therefore strikes the middle ground.

The overall vision of multitenancy in cloud computing platforms is to develop an architecture of a multitenant DBMS that is *consistent, scalable, fault-tolerant, elastic* and *self-managing*. We envision multitenancy as analogous to virtualization in the database tier for sharing the DBMS resources. Similar to virtual machine (VM) migration [25], efficient techniques for live database migration is an integral component to provide elastic load balancing. Live database migration should therefore be a first class feature in the system having the same stature as scalability, consistency, and fault-tolerance. However, no prior work exists in the area of live database migration for elastic load balancing. This dissertation presents the first two published solutions to this problem [34, 39].

Part I

Scale-out Transaction Processing

Chapter 3

Scaling-out with Database Partitioning

"We have not wings we cannot soar; but, we have feet to scale and climb, by slow degrees, by more and more, the cloudy summits of our time."

– Henry Wadsworth Longfellow.

Scaling out while processing transactions efficiently is an important requirement for databases supporting applications in the cloud. We analyzed multiple scalable DBMSs to distill some design principles for building systems that scale out to clusters of commodity servers while efficiently executing transactions. In this chapter, we first highlight these design principles and then show how these design principles can be used in building practical systems that effectively realize the goal of scale-out transaction processing.

3.1 Design Principles

RDBMSs and Key-value stores are the two most popular alternatives for managing large amounts of data. Even though very different in their origins and architectures, a careful analysis of their design allows us to distill some design principles that can be carried over in designing database systems for cloud platforms. The following is an articulation of the distilled design principles.

- **Separate system state from application state:** Abstractly, a distributed database system can be modeled as a composition of two different states: the **system state** and the **application state**. The system state is the meta data critical for the system's proper operation. It also represents the current state of different components that collectively form the distributed system. Examples of system state

are: node membership information in a distributed system, mapping of partitions to the nodes serving the partition, and the location of master and replicas in a replicated system. This state requires stringent consistency guarantees, fault-tolerance, and high availability to ensure the proper functioning of the system in the presence of different types of failures. However, scalability is not a primary requirement for the system state since it is typically small and not frequently updated. On the other hand, the application state is the application's data which the system stores. Consistency, scalability, and availability of the application state is dependent on the application's requirements. Different systems provide varying trade-offs among the different guarantees provided for application state.
A clean separation between the two states allows the use of different protocols, with different guarantees and associated costs, to maintain the two types of states. For instance, the system state can be made highly available and fault-tolerant by synchronously replicating this state using distributed consensus protocols such as Paxos [63] while the application state might be maintained using less stringent protocols.

- **Decouple data storage from ownership: Ownership** refers to the read/write access rights to data items. Separating (or **decoupling**) the data ownership and transaction processing logic from that of data storage has multiple benefits: (*i*) it results in a simplified design allowing the storage layer to focus on fault-tolerance while the ownership layer can guarantee higher level guarantees such as transactional access without worrying about the need for replication; (*ii*) depending on the application's requirements it allows independent scaling of the ownership layer and the data storage layer; and (*iii*) it allows for lightweight control migration for elastic scaling and load balancing, it is enough to safely migrate only the ownership without the need to migrate data.
- **Limit common operations to a single node:** Limiting the frequently executed operations to a single node allows efficient execution of the operations without the need for distributed synchronization. Additionally, it allows the system to horizontally partition and scale-out. It also limits the effect of a failure to only the data served by the failed component and does not affect the operation of the remaining components, thus allowing graceful performance degradation in the event of failures. As a rule of thumb, operations manipulating the application state must be limited to a single node in the database tier. Once transaction execution is limited to a single node, techniques from RDBMS literature for efficient transaction execution and performance optimization can be potentially applied [15, 87].
- **Limited distributed synchronization can be practical.** Distributed synchronization, if used in a prudent manner, can also be used in a scalable data manage-

ment system. The systems should limit distributed synchronization to the minimum and use it only when needed. Eliminating them altogether is not necessary for a scalable design.

3.2 Problem Formulation

Partitioning (or **sharding**) the database is one of the most common techniques to scale-out a database to a cluster of nodes. Typically a database is partitioned by splitting the individual tables within the database. Common partitioning techniques used are range partitioning or hash partitioning. Range partitioning involves splitting the tables into non-overlapping ranges of their keys and then mapping the ranges to a set of nodes. In hash partitioning, the keys are hashed to the nodes serving them. These partitioning techniques are simple and are supported by most common DBMSs. However, the main drawback of such techniques is that they result in the need for large numbers of distributed transactions to access data partitioned across multiple servers. As a result, the scalability of such systems is limited due to the cost of distributed transactions. Therefore, the first challenge is to partition the database such that most accesses are limited to a single partition.

Furthermore, managing large RDBMS installations with large numbers of partitions poses huge administrative overhead. First, partitioning itself is a tedious task, where the database administrator has to decide on the number of partitions, bring the database offline, partition the data on multiple nodes, and then bring it back up online [55]. Second, the partitioning and mapping of partitions to nodes are often static, and when the load characteristics change or nodes fail, the database needs re-partitioning or migration to a new partitioned layout, resulting in administrative complexity and downtime. Third, partitioning is often not transparent to the application, and applications often need modification whenever data is re-partitioned.

In this dissertation, we propose designs with fundamental advances beyond state-of-the-art RDBMSs to overcome the above-mentioned challenges of database partitioning.

- Instead of partitioning the tables independent of each other, we propose to partition the database based on access patterns with the goal to co-locate data items that are frequently accessed together within a transaction. This minimizes the number of cross-partition transactions, thus minimizing distributed transactions. The partitions are the granules of co-location and can be statically or dynamically defined.
- Instead of viewing the application state as a tightly-coupled cohesive granule, our proposed designs view the application state as a set of loosely coupled database partitions. These partitions form the granule of transactional access. Such a

design has two implications. For large application databases, this approach to database partitioning limits transactions to a single node, thus ensuring scalability without considerably limiting the set of operations on which transactional guarantees are provided. In the case of small databases, typically observed in multitenant platforms [9, 88, 90], the systems can provide full transactional support for the small tenant databases that are self-contained partitions.

- Our designs dynamically map the partitions to the nodes serving the partitions while abstracting this mapping from the applications. This allows application transparent partition re-assignments due to failures or load balancing.

For applications whose database can be statically partitioned based on the access patterns, we propose ElasTraS, a system to provide transactional support to large numbers of these partitions distributed across a cluster of servers; Chapter 4 presents the design of ElasTraS in detail. For applications whose access patterns change dynamically, thus negating the benefits of static partitioning, we propose the Key Group abstraction for applications to dynamically define the data items that will form a partition. We propose the Key Grouping protocol to dynamically co-locate ownership of the data items within a partition, thus allowing efficient transaction execution on the dynamically formed partitions. Chapter 5 presents the details of the Key Grouping protocol and G-Store, a prototype implementation of the Key Grouping protocol to guarantee transactional multi-key access on top of a Key-value store.

Chapter 4

Statically Defined Partitions

"Geography has made us neighbors. History has made us friends. Economics has made us partners, and necessity has made us allies. Those whom God has so joined together, let no man put asunder."

– John F. Kennedy.

In this chapter, we present the detailed design and implementation of ElasTraS [31, 32], an elastically scalable transaction processing system. ElasTraS views the database as a set of database partitions. The partitions form the granule of distribution, transactional access, and load balancing. For small application (or tenant) databases, as observed in multitenant platforms serving large numbers of small applications [88, 90], a tenant's database can be contained entirely within a partition. For applications whose data requirements grow beyond a single partition, ElasTraS supports partitioning at the schema level by co-locating data items frequently accessed together. ElasTraS is designed to serve thousands of *small* tenants as well as tenants that *grow big*.[1]

ElasTraS is a culmination of two major design philosophies and embodies a unique combination of their design principles: traditional RDBMSs for efficient transaction execution on small databases and the Key-value stores for scale, elasticity, and high availability. ElasTraS allows effective resource sharing among tenants while supporting low latency transaction processing and low overhead live database migration for elastic load balancing. This lends ElasTraS the unique set of features critical for a multitenant DBMS serving a cloud platform. ElasTraS assumes an underlying pay-per-use infrastructure and is designed to serve on-line transaction processing (OLTP) style application workloads that use a relational data model. ElasTraS's target deployment is a

[1]An earlier abridged version of the work reported in this chapter was published as the paper entitled "ElasTraS: An Elastic Transactional Data Store in the Cloud" in the proceedings of the 2009 USENIX workshop on Hot topics in cloud computing (HotCloud).

cluster of few tens of nodes co-located within a data center and processing billions of transactions per day.

At the microscopic scale, ElasTraS consolidates multiple tenants within the same database process (shared process multitenancy) allowing effective resource sharing among small tenants. It achieves high transaction throughput by limiting tenant databases to a single process, thus obviating distributed transactions. For tenants with sporadic changes in loads, ElasTraS leverages low-cost live database migration for elastic scaling and load balancing. This allows it to aggressively consolidate tenants to a small set of nodes while still being able to scale-out on-demand to meet tenant SLAs in the event of unpredictable load bursts.

At the macroscopic scale, ElasTraS uses loose synchronization between the nodes for coordinating operations, rigorous fault-detection and recovery algorithms to ensure safety during failures, and system models that automate load balancing and elasticity. ElasTraS significantly advances the state-of-the-art by *presenting a unique combination of multitenancy and elasticity in a single database system, while being self-managing, scalable, fault-tolerant, and highly available.*

4.1 The Design of ElasTraS

4.1.1 ElasTraS Architecture

We explain the ElasTraS architecture in terms of the four layers shown in Figure 4.1 from bottom-up: the distributed fault-tolerant storage layer, the transaction management layer, the control layer, and the routing layer.

The Distributed Fault-tolerant Storage Layer

The storage layer, or the Distributed Fault-tolerant Storage (**DFS**), is a network-addressable storage abstraction that stores the persistent data. This layer is a replicated storage manager that guarantees durable writes and strong replica consistency while ensuring high data availability in the presence of failures. Such storage abstractions are common in current data centers in the form of commercial products (such as storage area networks), scalable distributed file systems (such as the Hadoop distributed file system [50]), or custom solutions (such as Amazon elastic block storage or the storage layer of Hyder [13]). High-throughput and low-latency data center networks provide low cost reads from the storage layer; however, strong replica consistency make writes expensive. ElasTraS minimizes the number of DFS accesses to reduce network communication and improve the overall system performance. We use a multi-version

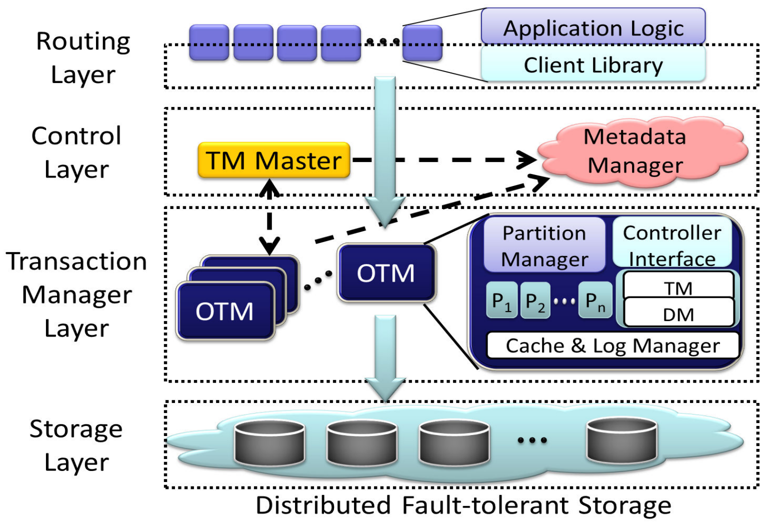


Figure 4.1: ElasTraS architecture. Each box represents a server, a cloud represents a service abstraction, dashed arrows represent control flow, block arrows represent data flow, and dotted blocks represent conceptual layers in the system.

append-only storage layout that supports more concurrency for reads and considerably simplifies live migration for elastic scaling.

Transaction Management Layer

This layer consists of a cluster of servers called Owning Transaction Managers (**OTM**). An OTM is analogous to the transaction manager in a classical RDBMS. Each OTM serves tens to hundreds of partitions for which it has unique **ownership**, i.e., exclusive read/write access to these partitions. The number of partitions an OTM serves depends on the overall load. The exclusive ownership of a partition allows an OTM to cache the contents of a partition without violating data consistency while limiting transaction execution within a single OTM and allowing optimizations such as *fast commit* [87]. Each partition has its own transaction manager (**TM**) and shared data manager (**DM**). All partitions share the OTM's log manager which maintains the transactions' commit log. This sharing of the log minimizes the number of competing accesses to the shared storage while allowing further optimizations such as group commit [15, 87]. To allow fast recovery from OTM failures and to guarantee high availability, an OTM's commit log is stored in the DFS. This allows an OTM's state to be recovered even if it fails completely.

Control Layer

This layer consists of two components: the **TM Master** and the Metadata Manager (**MM**). The TM Master monitors the status of the OTMs and maintains overall system load and usage statistics for performance modeling. The TM Master is responsible for assigning partitions to OTMs, detecting and recovering from OTM failures, and controlling elastic load balancing. On the other hand, the MM is responsible for maintaining the system state to ensure correct operation. This metadata consists of **leases** that are granted to every OTM and the TM Master, **watches**, a mechanism to notify changes to a lease's state, and (a pointer to) the **system catalog**, an authoritative mapping of a partition to the OTM currently serving the partition. Leases are uniquely granted to a server for a fixed time period and must be periodically renewed. Since the control layer stores only meta information and performs system maintenance, it is not in the data path for the clients. The state of the MM is critical for ElasTraS's operation and is replicated for high availability; the TM Master is stateless.

Routing Layer

ElasTraS dynamically assigns partitions to OTMs. Moreover, for elastic load balancing, a database partition can be migrated on-demand in a live system. The routing layer, the **ElasTraS client library** which the applications link to, hides the logic of connection management and routing, and abstracts the system's dynamics from the application clients while maintaining un-interrupted connections to the tenant databases.

4.1.2 Design Rationales

Before we present the details of ElasTraS's implementation, we explain the rationale that has driven this design. We relate these design choices to the principles outlined in Section 3.1.

ElasTraS uses a decoupled storage architecture where the persistent data is stored on a cluster of servers that is logically separate from the servers executing the transactions. This architecture distinguishes ElasTraS from systems such as Cloud SQL Server [14] and RelationalCloud [28] that resemble the classical shared nothing architecture where the persistent data is stored in disks locally attached to the transaction managers. The rationale for decoupling the storage layer is that for high performance OLTP workloads, the active working set must fit in the cache or else the peak performance is limited by the disk I/O. OLTP workloads, therefore, result in infrequent disk accesses. Hence, decoupling the storage from transaction execution will result in negligible performance since the storage layer is accessed infrequently. Moreover, since the storage layer is connected to the TM layer via a low latency and high throughput data center network, remote data access is fast. Furthermore, this decoupling of the storage layer allows independent scaling of the storage and transaction management layers, simplifies the design of the transaction managers by abstracting fault-tolerance in the storage layer, and allows lightweight live database migration without migrating the persistent data. ElasTraS's use of decoupled storage is, however, different from traditional shared disk systems [87], Hyder [13], or Megastore [11] that use the shared storage as the point of arbitration between different servers. In ElasTraS, each server is allocated a storage location that does not overlap with any other server.

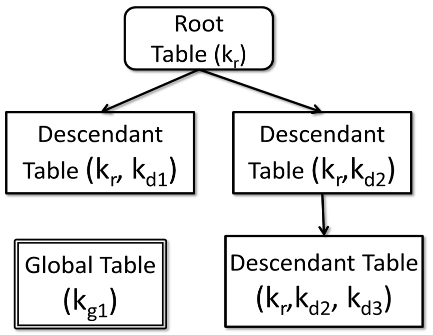
Co-locating a tenant's data into a single partition, serving a partition within a single database process, and limiting transactions to a partition allows ElasTraS to limit transactions to a single node, thus limiting the most frequent operations to a single node. This allows efficient transaction processing and ensures high availability during failures. For small tenants contained in a single partition, ElasTraS supports flexible multi-step transactions. The size and peak loads of a single-partition tenant is limited by the capacity of a single server.

ElasTraS separates the application state, stored in the storage layer and served by the transaction management layer, from the system state, managed by the control layer. ElasTraS replicates the system state, using protocols guaranteeing strong replica consistency, to ensure high availability. The clean separation between the two types of state allows ElasTraS to limit distributed synchronization to only the critical system state while continuing to use non-distributed transactions to manipulate application state.

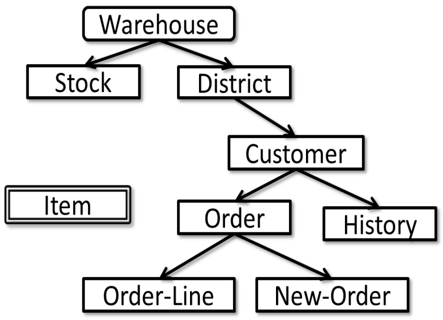
ElasTraS's use of the shared process multitenancy model is in contrast to the shared table model used by systems such as Salesforce.com [88] and Megastore [11]. In addition to providing effective consolidation (scaling to large numbers of small tenants) and low cost live migration (elastic scaling), the shared process model provides good isolation between tenants and provides for more flexible tenant schema.

4.1.3 Partitioning the Database Schema

ElasTraS supports partitioning the big databases. We now explain a particular instance of schema level partitioning. ElasTraS supports a tree-based schema for partitioned databases. Figure 4.2(a) provides an illustration of such a schema type. This schema supports three types of tables: the **Root Table**, **Descendant Tables**, and **Global Tables**. A schema has one root table whose primary key acts as the partitioning key. A schema can however have multiple descendant and global tables. Every descendant table in a database schema will have the root table's key as a foreign key. Referring to Figure 4.2(a), the key k_r of the root table appears as a foreign key in each of the descendant tables.


Root Table (k_r)
Descendant Table (k_r, k_{d1})
Descendant Table (k_r,k_{d2})
Global Table (k_{g1})
Descendant Table (k_r,k_{d2}, k_{d3})

(a) Tree Schema.



(b) TPC-C as a tree schema.

Figure 4.2: Schema level database partitioning.

This tree structure implies that corresponding to every row in the root table, there are a group of related rows in the descendant tables, a structure similar to row groups in Cloud SQL Server [14]. All rows in the same row group are guaranteed to be co-located. A transaction can only access rows in a particular row group. A database partition is a collection of such row groups. This schema structure also allows efficient dynamic splitting and merging of partitions. In contrast to these two table types, global tables are look-up tables that are mostly read-only. Since global tables are not updated frequently, these tables are replicated on all the nodes; decoupled storage allows ElasTraS to cache the global tables at the OTMs without actually replicating the data. In addition to accessing only one row group, an operation in a transaction can also read from a global table. Figure 4.2(b) shows a representation of the TPC-C schema [81] as a tree schema. The TATP benchmark [70] is another example of a schema conforming to the tree schema.

ElasTraS requires databases to conform to the tree schema to enable partitioning; small tenant databases contained within a single partition do not require to have a tree schema. For such a configuration, a transaction can access any data item within the partition and the partition is the granule of transactional access. This configuration is similar to table groups in Cloud SQL Server.

The "tree schema" has been demonstrated to be amenable to partitioning using a partition key shared by all the tables in the schema. Even though such a schema does not encompass the entire spectrum of OLTP applications, a survey of real applications within a commercial enterprise shows that a large number of applications either have such an inherent schema pattern or can be easily adapted to it [14]. A similar schema pattern is observed in two other concurrently developed systems, Cloud SQL Server and Megastore. Since ElasTraS operates at the granularity of the partitions, the architecture can also leverage other partitioning techniques, such as Schism [30].

4.2 Implementation Details

4.2.1 Storage layer

Many existing solutions provide the guarantees required by the ElasTraS storage layer; in our prototype, we use the Hadoop Distributed File System (HDFS) [50]. HDFS is a replicated append-only file system providing low-cost block-level replication with strong replica consistency when only one process is appending to a file. ElasTraS ensures that at any point in time, only a single OTM appends to an HDFS file. A write is acknowledged only when it has been replicated in memory at a configurable number of servers (called data nodes in HDFS). ElasTraS optimizes the number of writes to HDFS by batching DFS writes to amortize the cost.

4.2.2 Transaction manager layer

The transaction manager layer consists of a cluster of OTMs serving the partitions. Even though each ElasTraS OTM is analogous to a relational database, we implemented our own custom OTM for the following reasons. First, most existing open-source RDBMSs do not support dynamic partition re-assignment which is essential for live migration. Second, since traditional RDBMSs were not designed for multitenancy, they use a common transaction and data manager for all partitions sharing a database process. This makes tenant isolation and guaranteeing per tenant SLA more challenging. Third, ElasTraS uses advanced schema level partitioning for large tenants while most open-source databases only support simple table-level partitioning mechanisms off-the-shelf.

An ElasTraS OTM has two main components: the transaction manager responsible for concurrency control and recovery and the data manager responsible for storage and cache management.

Concurrency control and Recovery

Concurrency Control. There exists a rich literature of concurrency control techniques that can be used for transaction management in an OTM [15, 87]. In our prototype, we implement Optimistic Concurrency Control (**OCC**) [61] to ensure serializability. In OCC, transactions do not obtain locks when reading or writing data. They rely on the optimistic assumption that there are no conflicts with other concurrently executing transactions. Before a transaction commits, it is validated to guarantee that the optimistic assumption was indeed correct and the transaction did not conflict with any other concurrent transaction. In case a conflict is detected, the transaction is aborted. Writes made by a transaction are kept local and are applied to the database only if the transaction commits. We implemented parallel validation in OCC which results in a very short critical section for validation, thus allowing more concurrency [61].

Log Management. Transaction logging is critical to ensure the durability of committed transactions in case of OTM failures. Every OTM maintains a *transaction log* (or **log** for brevity) where updates from transactions executing at the OTM are appended. All partitions at an OTM share a common log which is stored in the DFS to allow quick recovery from OTM failures. This sharing minimizes the number of DFS accesses and allows effective batching. Each log entry has a partition identifier to allow a log entry to be associated to the corresponding partition during recovery. Log entries are buffered during transaction execution. Once a transaction has been successfully validated, and before it can commit, a `COMMIT` record for this transaction is appended to the log, and the log is *forced* (or flushed) to ensure durability. The following optimizations minimize the number of DFS accesses: (i) no log entry is written to record the start of a transaction, (ii) a `COMMIT` entry is appended only for update transactions, (iii) no log entry is made for aborted transactions, so the absence of a `COMMIT` record implies an abort, and (iv) group commits are implemented to commit transactions in groups and batch their log writes [87]. (i) and (ii) ensure that there are no unnecessary log writes for read-only transactions. Buffering and group commits allow the batching of updates and are optimizations to improve throughput.

Recovery. Log records corresponding to a transaction's updates are forced to the disk before the transaction commits. This ensures durability and transaction recoverability. In the event of a failure, an OTM recovers the state of failed partitions. In OCC, uncommitted data is never flushed to the DFS, and hence, only redo recovery is needed. The OTM replays the log to recreate the state prior to failure and recovers updates from all the committed transactions. We use a standard two-pass recovery algorithm [87]. The first pass is the analysis pass that determines the transactions that committed. Only those transactions that have a `COMMIT` record in the log are considered committed. The second pass re-does the updates of all committed transactions. Once the redo pass over the log is complete, the cache is flushed to the DFS, old log files are deleted, and the OTM starts serving the partition. Version information in the storage layer is used to guarantee idempotence of updates to guarantee safety during repeated OTM failures.

Storage and Cache management

Storage Layout. The storage layer in ElasTraS is append-only and the DM is designed to leverage this append-only nature. Persistent data for a partition consists of a collection of *immutable* **segments** which store the rows of a table in sorted order of their primary keys. Internally, a segment is a collection of **blocks** with an index to map blocks to key ranges. This segment index is used to read directly from the block containing the requested row and thus avoiding an unnecessary scan through the entire segment. In traditional database terminology, a segment is equivalent to a clustered index on the primary key of a table, secondary or non-clustered indices are stored as separate segments pointing to the segments storing the clustered index. This storage layout allows efficient lookups and range scans on the clustered index while allowing a simple storage layout.

Cache Management. Each OTM caches recently accessed data; the cache at an OTM is shared by all partitions being served by the OTM. Due to the append-only nature of the storage layout, a DM in ElasTraS uses separate read and write caches. Updates are maintained in the write cache which is periodically flushed to the DFS as new segments; a flush of the write cache is asynchronous and does not block new updates. As in-memory updates are flushed to the DFS, the corresponding entries in the transaction log are marked for garbage collection. The append-only semantics of the storage layer adds two new design challenges: (i) fragmentation of the segments due to updates and deletions, and (ii) ensuring that reads see the updates from the latest writes in spite of the two separate caches. A separate garbage collection process addresses the first challenge. The second challenge is addressed by answering queries from a merged view of the read and write caches. A least recently used policy is used for cache maintenance. Since a segment is immutable, a cache eviction does not result in a DFS write.

Garbage Collection. The immutable segments get fragmented internally with updates and deletes. A periodic garbage collection process recovers unused space and reduces the number of segments. In the same vein as Bigtable [24], we refer to this process as **compaction**. This compaction process executes in the background and merges multiple small segments into one large segment, removing deleted or updated entries during this process. Old segments can still be used to process reads while the new compacted segments are being produced. Similarly, the log is also garbage collected as the write cache is flushed, thus reducing an OTM's recovery time.

4.2.3 Control layer

The control layer has two components: the metadata manager (MM) and the TM Master. We use Zookeeper [54] as the MM. The Zookeeper service provides low overhead distributed coordination and exposes an interface similar to a filesystem where each file is referred to as a **znode**. A Zookeeper client can request exclusive access to a znode, which is used to implement timed *leases* in the MM. Zookeeper also supports *watches* on the leases that notify the watchers when the state of a lease changes.

ElasTraS uses the Zookeeper primitives—leases and watches—to implement timed *leases* granted to the OTMs and the TM Master. Each OTM and the TM Master acquires exclusive write access to a znode corresponding to the server; an OTM or the TM Master is operational only if it owns its znode. The OTMs register a watch on the TM Master's znode, while the TM Master registers watches for each OTM's znode. This event-driven mechanism allows the MM to scale to large numbers of OTMs. Even in the presence of concurrent requests and arbitrary failures, Zookeeper ensures that only one client has exclusive access to a znode. Zookeeper's state is replicated on an ensemble of servers; an atomic broadcast protocol ensures replica consistency [58].

A system catalog maintains the mapping of a partition to the OTM serving the partition. Each partition is identified by a unique partition identifier; the catalog stores the partition-id to OTM mapping as well as the metadata corresponding to tenant partitions, such as schema, authentication information, etc. The catalog is stored as a database within ElasTraS and is served by one of the live OTMs. The MM maintains the address of the OTM serving the catalog database.

The TM Master automates failure detection using periodic heart-beat messages and the MM's notification mechanism. Every OTM in the system periodically sends a heart-beat to the TM Master. In case of an OTM failure, the TM Master times-out on the heart beats from the OTM and the OTM's lease with the MM also expires. The master then atomically deletes the znode corresponding to the failed OTM. This deletion makes the TM Master the *owner* of the partitions that the failed OTM was serving and ensures safety in the presence of a network partition.

Once the znode for an OTM is deleted, the TM Master reassigns the failed partitions to another OTM. In an infrastructure supporting dynamic provisioning, the TM Master can spawn a new node to replace the failed OTM. In statically provisioned settings, the TM Master assigns the partitions of the failed OTM to the set of live OTMs. After reassignment, a partition is first recovered before it comes online. Since the persistent database and the transaction log are stored in the DFS, partitions can be recovered without the failed OTM recovering. An OTM failure affects only the partitions that OTM was serving; the rest of the system remains available.

ElasTraS also tolerates failures of the TM Master. A deployment can have a standby TM Master which registers a watch on the active TM Master's znode. When the acting TM Master fails, its lease on the znode expires, and the standby TM Master is notified. The standby TM Master then acquires the master lease to become the acting TM Master. The MM's notification mechanism notifies all the OTMs about the new master which then initiates a connection with the new TM Master. This TM Master fail-over does not affect client transactions since the master is not in the data path of the clients. Therefore, there is no single point of failure in an ElasTraS installation.

4.2.4 Routing layer

The routing layer comprises of the ElasTraS client library that abstracts the complexity of looking up and initiating a connection with the OTM serving the partition being accessed. Application clients link directly to the ElasTraS client library and submit transactions using the client interface. The ElasTraS client is responsible for: (i) looking up the catalog information and routing the request to the appropriate OTM, (ii) retrying the requests in case of a failure, and (iii) re-routing requests to a new OTM when a partition is migrated. When compared to the traditional web-services architecture, the client library is equivalent to a proxy/load balancer, except that it is linked with the application server.

The ElasTraS client library determines the location of the OTM serving a specific partition by querying the system catalog. When a client makes the first connection to ElasTraS, it has to perform two lookups: a query to the MM to determine the address of the OTM serving the catalog, and then a query to the catalog to determine the location of the OTM serving the partition. However, a client caches this information for subsequent accesses to the same partition. The catalog tables are accessed again only if there is a cache miss or an attempt to connect to a cached location fails. Performance is further improved by pre-fetching rows of the catalog tables to speedup subsequent lookups for different partitions. Therefore, in the steady state, the clients directly connect to the OTM using cached catalog information. The distributed nature of the routing layer allows the system to scale to large numbers of tenants and clients.

4.2.5 Advanced implementation aspects

Multi-version data

The storage layer of ElasTraS is append-only and hence multi-version. This allows for a number of implementation optimizations. First, read-only queries that do not need the latest data can run on snapshots of the database. Second, for large tenants spanning multiple partitions, read-only analysis queries can be executed in parallel without the need for distributed transactions. Third, if a tenant experiences high transaction load, most of which is read-only, the read-only transactions can be processed by another OTM that executes transactions on a snapshot of the tenant's data read from the storage layer. Since such transactions do not need validation, they can execute without any coordination with the OTM currently owning the tenant's partition. The use of snapshot OTMs in ElasTraS is only on-demand to deal with high load. The snapshot OTMs also allow the co-existence of transaction processing and analysis workloads without the cost of data duplication.

Dynamic Partitioning

When using a tree schema, ElasTraS can be configured to dynamically split or merge partitions. In a tree schema configuration, a partition is a collection of row groups. A partition can therefore be dynamically split into two sets of row groups. To split a partition, an OTM first splits the root table and then uses the root table's key to split the descendant tables. Global tables are replicated, and hence are not split. The root table is physically organized using its key order and the rows of the descendant tables are first ordered by the root table's key and then by the descendant's key. This storage organization allows efficient splitting of partitions, since the key used to split the primary key can also be used to split the descendant tables into two parts. To ensure a quick split, an OTM performs a logical split; the two child partitions reference the top and bottom halves of the old parent partition. The OTM performing the split updates the system state to reflect the logical split. Thus, splitting does not involve data shuffling and is efficient. To ensure correctness of the split in the presence of a failure, this split is executed as a transaction, thus providing an atomic split. Once a partition is split, the TM Master can re-assign the two sub-partitions to different OTMs. Data is moved to the new partitions asynchronously and the references to the old partition are eventually cleaned up during garbage collection.

4.3 Correctness Guarantees

Correctness in the presence of different types of failure is critical. Our failure model assumes that the node failures are fail-stop and we do not tolerate Byzantine failures or malicious behavior. We do not consider scenarios of complete data loss or message failures. ElasTraS ensures safety in the presence of network partitions. Before we delve into the correctness arguments, we first define safety and liveness.

Definition 4.3.1. ***Safety:*** *At no instant of time should more than one OTM claim ownership of a database partition.*

Definition 4.3.2. ***Liveness:*** *In the absence of repeated failures, i.e., if the TM Master and OTMs can communicate for sufficiently long durations and the MM is live, no database partition is indefinitely left without an owner. The MM is live if it can successfully perform updates to its state, i.e., renew existing leases and grant new leases.*

Safety guarantees that nothing bad happens to the data in the event of failures, while liveness ensures that something good will happen in the absence of repeated failures. ElasTraS's correctness is dependent on certain guarantees by the storage and metadata management layers. We first list these guarantees and then use them to reason informally about the correctness of ElasTraS.

4.3.1 Guarantees provided by the MM and DFS

The MM provides the following guarantees.

Guarantee 4.3.3. *System state stored in the MM is not lost or left in an inconsistent state in the presence of arbitrary failures or network partitions.*

Guarantee 4.3.4. *The MM is live if a majority of replicas are non-faulty and can communicate.*

Guarantee 4.3.5. *At any instant of time, the MM will assign a lease to only a single node, i.e., a lease is mutually exclusive.*

If a lease (or znode) is available, the request to acquire a lease is granted only when a majority of MM replicas have acknowledged the request to grant the lease. This majority approval ensures that a lease is granted to only a single node and also ensures that any other concurrent request for the same lease is rejected, thus guaranteeing consistency of the MM's state. Our implementation uses Zookeeper which provides all the above guarantees [54].

The DFS layer provides the following guarantees.

Guarantee 4.3.6. ***Durability.*** *Appends flushed to the DFS are never lost.*

This is guaranteed by acknowledging a flush only after the appends have propagated to a configurable number of replicas. If all the replicas of a data block fails, then the data becomes unavailable, but is recovered when the replicas recover.

Guarantee 4.3.7. ***Strong replica consistency.*** *When only a single process is appending to a file, the replicas of the file are strongly consistent.*

The block level replication protocol in HDFS guarantees strong replica consistency with a single appender.

4.3.2 Safety Guarantees

We first prove that there can be at most one operational TM Master in an ElasTraS installation and then use it to prove that at most one OTM can own a database partition in ElasTraS.

Lemma 4.3.8. *At any instant of time there is only a single operational TM master in one ElasTraS installation.*

Proof. Assume for contradiction that an ElasTraS installation has more than one operational TM Masters. A TM Master is operational *only if* it owns the master lease with the MM. Since an ElasTraS installation contains a single master lease, multiple TM Masters imply that the master lease is concurrently owned by multiple nodes, a contradiction to Guarantee 4.3.5. □

Theorem 4.3.9. *At any instant of time, at most one OTM has write access to a database partition.*

Proof. Assume for contradiction that at least two OTMs have write access to a database partition ($\mathbb{P}$). Let us assume that $\mathbb{P}$ was assigned to OTM_1 and OTM_2. Only an operational TM Master can assign $\mathbb{P}$ to an OTM. Therefore, following Lemma 4.3.8, $\mathbb{P}$ cannot be assigned by two different TM Masters. Two scenarios are possible: (i) a single TM Master assigned $\mathbb{P}$ to both OTM_1 and OTM_2; or (ii) a TM Master assigned $\mathbb{P}$ to OTM_1, then the TM Master failed, and a subsequent TM Master assigned $\mathbb{P}$ to OTM_2.
Scenario (i): Without loss of generality, assume that $\mathbb{P}$ was assigned to OTM_1 first, and then to OTM_2 while OTM_1 still owns $\mathbb{P}$. Such a reassignment is impossible during normal operation, i.e., when the TM Master and OTM_1 can communicate. The TM Master will reassign $\mathbb{P}$ to OTM_2 if it mistakenly determined that OTM_1 has failed. This is possible when the network connecting the TM Master and OTM_1 has partitioned. The TM Master will reassign $\mathbb{P}$ *only if* it successfully obtained OTM_1's lease. OTM_1 owns $\mathbb{P}$ *only if* it owns its lease with the MM, i.e., OTM_1 can communicate with the MM. In case OTM_1 is also partitioned from the MM, OTM_1 will lose $\mathbb{P}$'s ownership. Therefore, before the TM Master reassigns $\mathbb{P}$, there is a time instant when both the TM Master and OTM_1 concurrently own the same lease with the MM, a contradiction to Guarantee 4.3.5.
Scenario (ii): Once the TM Master assigns $\mathbb{P}$ to OTM_1, it updates the metadata to persistently store this mapping with the MM. After this TM Master fails, a subsequent operational TM Master will reassign $\mathbb{P}$ to OTM_2, even though OTM_1 owns $\mathbb{P}$, *only if* the metadata reflects that $\mathbb{P}$ was unassigned. However, this is a contradiction to Guarantee 4.3.3. □

4.3.3 Transactional Guarantees

An OTM executes transactions on a partition and is responsible for ensuring the transactional guarantees.

Lemma 4.3.10. *Transactions on a single partition are guaranteed to be atomic, consistent, durable, and serializable.*

A transaction commit is acknowledged by an OTM only after the log records for a transaction, including the `COMMIT` record, has been forced to the DFS. Guarantee 4.3.6 ensures that the log records survive a failure of the OTM. This ensures that the transaction state is recoverable and guarantees durability. On failure of an OTM, the REDO recovery operation on a partition from the commit log applies all updates of a transaction whose `COMMIT` record is found in the commit log (i.e., the transaction successfully committed) and discards all updates of a transaction whose `COMMIT` record is missing (meaning either the transaction aborted and was rolled back, or it did not commit before the failure and hence is considered as aborted). This ensures atomicity of the transactions. Consistency and serializable isolation is guaranteed by the concurrency control protocol [61].

4.3.4 Liveness Guarantees

ElasTraS's liveness is contingent on the MM's liveness. As discussed in Section 4.2.3, if the MM is alive and is able to communicate with the rest of the nodes, then ElasTraS can also tolerate the failure of the TM Master and continue to serve the partitions without a functioning TM Master until an OTM fails.

Lemma 4.3.11. *The failure of an OTM does not leave its partitions without an owner for indefinite periods.*

If an ElasTraS cluster has an operational TM Master, then the TM Master detects that the OTM has failed, obtains the lease for the failed OTM, and reassigns ownership of the partitions owned by the failed OTM. If there is no operational TM Master. Once a new TM Master joins the cluster, the partitions are reassigned.

4.4 Elasticity and Load balancing

Elasticity is one of the major goals of ElasTraS's design. To be effective, elastic scaling and load balancing in a live system must result in minimal service interruption and negligible impact on performance. The TM Master in ElasTraS automates these decisions to minimize the need for human intervention. We now discuss the most important components for elastic load balancing.

System modeling. Modeling the system's performance and behavior plays an important role in determining when to scale up the system, when to consolidate to a fewer number of nodes, and how to assign partitions to the OTMs. Every live OTM periodically sends heartbeats to the TM Master which include statistics about the load on the

OTM, i.e., the number of partitions, the number of requests per partition, CPU, IO and other resource utilization metrics. The TM Master aggregates these statistics across all the OTMs to create a model for the overall system performance and incrementally maintains it to create a history of the system's behavior. This model is then used to predict future behavior, load balance partitions, or to decide on elastic scaling of the system.

A combination of modeling techniques can be used to improve the accuracy of predictions and decisions. The simplest model is rule based: if the system's load is above a max-threshold, then new OTM instances are added, while if it is below a min-threshold, partitions are consolidated into a smaller number of nodes. The threshold values are based on performance measurements of utilization levels above which a considerable impact on tenant transactions is observed, or under which the system's resources are being under-utilized. Historical information is used to predict future behavior and to detect transient spikes in load, while allowing proactive scaling when a high impending load is predicted. These simple modeling techniques work well for a class of workloads with predictable behavior. We plan to explore other advanced modeling techniques based on control theory, machine learning, and time series.

The partition assignment algorithm determines an assignment of partitions to OTMs ensuring that the tenant SLAs are met while consolidating the tenants into as few servers as possible. Therefore, the partition assignment problem is an optimization problem. In our current prototype, we use a greedy heuristic where the historical load and resource utilization statistics per partition are used to assign partitions to an OTM without overloading it.

Live database migration. Elastic load balancing mandates that migration of partitions between OTMs incurs minimal overhead on the system as well as negligible impact on the users. Since ElasTraS uses the shared process multitenancy model, live VM migration techniques cannot be directly applied. The decoupled storage architecture of ElasTraS allows a lightweight migration of *ownership* without moving the persistent data. We developed Albatross [34], a live migration technique for decoupled storage database architectures such as ElasTraS. Chapter 7 presents the details of Albatross.

4.5 Experimental Evaluation

We now experimentally evaluate our prototype implementation of ElasTraS to analyze the impact of the three major design choices of ElasTraS: the shared storage layer, shared process multitenancy, and co-locating tenant's data into a single partition and limiting transactions to a single partition.

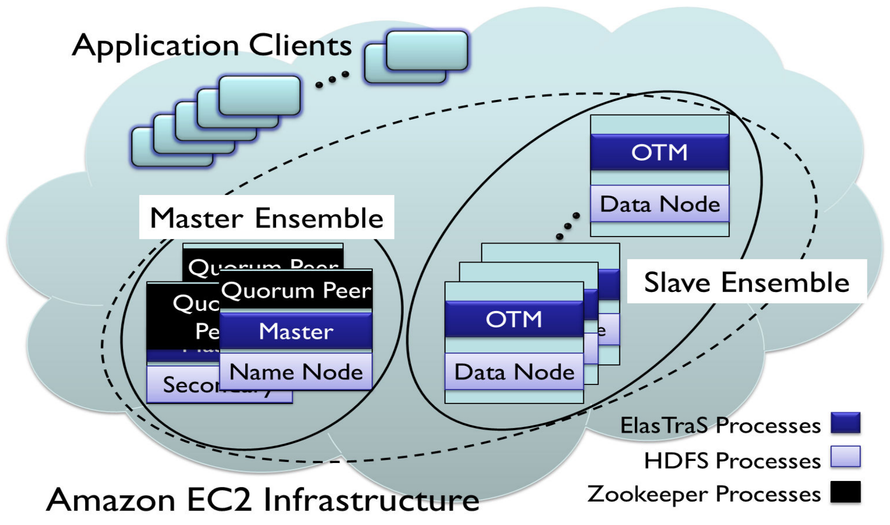


Figure 4.3: Cluster setup for experiments to evaluate ElasTraS's performance.

The performance of the storage layer (HDFS) impacts ElasTraS's performance. Several studies have reported on HDFS performance [80]. In our experimental configuration using three replicas, we observed an average read bandwidth of 40 MBps when reading sequentially from a 64 MB block in HDFS. When reading small amounts of data, the typical read latency was on the order of 2 to 4 ms. A latency on the order of 5 ms was observed for appends. In our implementation, log appends are the only synchronous writes, a flush of the cache is performed asynchronously. Group commit amortizes the cost of a log append over multiple transactions. The cost of accessing data from the storage layer is therefore comparable to that of a locally attached storage.

4.5.1 Experimental Setup

Our experiments were performed on a local cluster and on Amazon Web Services Elastic Compute Cloud (**EC2**). Servers in the local cluster had a quad core processor, 8 GB RAM, and 1 TB disk. On EC2, we use the `c1.xlarge` instances each of which is a virtual machine with eight virtual cores, 7 GB RAM, and 1.7 TB local storage. ElasTraS is implemented in Java and uses HDFS v0.21.0 [50] as the storage layer and Zookeeper v3.2.2 [54] as the metadata manager. We use a three node Zookeeper ensemble in our experiments. The nodes in the Zookeeper ensemble also host the ElasTraS TM Master and the HDFS Name node and Secondary Namenode. A separate ensemble of nodes host the slave processes of ElasTraS and HDFS: i.e., the ElasTraS OTMs which serve the partitions, and the Data nodes of HDFS which store the filesystem data. In the local cluster, the OTMs and the Data nodes were executed on different nodes, while in EC2, each node executed the Data node and the OTM processes. Figure 4.3 illustrates the experimental cluster set-up.

4.5.2 Workload Specification

Our evaluation uses two different OLTP benchmarks that have been appropriately adapted to a multitenant setting: (*i*) the Yahoo! cloud serving benchmark (YCSB) [27] adapted for transactional workloads to evaluate performance under different read/write loads and access patterns; and (*ii*) the TPC-C benchmark [81] representing a complex transactional workload for typical relational databases.

Yahoo! Cloud Serving Benchmark

YCSB [27] is a benchmark to evaluate DBMSs serving web applications. The benchmark was initially designed for Key-value stores and hence did not support transactions. We extended the benchmark to add support for multi-step transactions that access data only from a single database partition. We choose a tenant schema with three tables where each table has ten columns of type `VARCHAR` and 100 byte data per column; one of the tables is the root and the two remaining tables are descendant. The workload consists of a set of multi-step transactions parameterized by the number of operations, percentage of reads and updates, and the distribution (uniform, Zipfian, and hotspot distributions) used to select the data items accessed by a transaction. We also vary the transaction loads, database sizes, and cache sizes. To simulate a multitenant workload, we run one benchmark instance for each tenant. YCSB comprises small transactions whose logic executes at the clients.

TPC-C Benchmark

The TPC-C benchmark is an industry standard benchmark for evaluating the performance of OLTP systems [81]. The TPC-C benchmark suite consists of nine tables and five transactions that portray a wholesale supplier. The five transactions represent various business needs and workloads: (i) the NEWORDER transaction which models the placing of a new order; (ii) the PAYMENT transaction which simulates the payment of an order by a customer; (iii) the ORDERSTATUS transaction representing a customer query for checking the status of the customer's last order; (iv) the DELIVERY transaction representing deferred batched processing of orders for delivery; and (v) the STOCKLEVEL transaction which queries for the stock level of some recently sold items. A typical transaction mix consists of approximately 45% NEWORDER transactions, 43% PAYMENT transactions, and 4% each of the remaining three transaction

Parameter	**Default value**
Transaction size	10 operations
Read/Write distribution	9 reads 1 write
Database size	4 GB
Transaction load	200 transactions per second (TPS)
Hotspot distribution	90% operations accessing 20% of the database

Table 4.1: Default values for YCSB parameters used in ElasTraS's evaluation.

types, representing a good mix of read/write transactions. The number of warehouses in the database determines the scale of the system. Since more than 90% of transactions have at least one write operation (insert, update, or delete), TPC-C represents a write heavy workload. Each tenant is an instance of the benchmark. The TPC-C benchmark represents a class of complex transactions executed as stored procedures at an OTM.

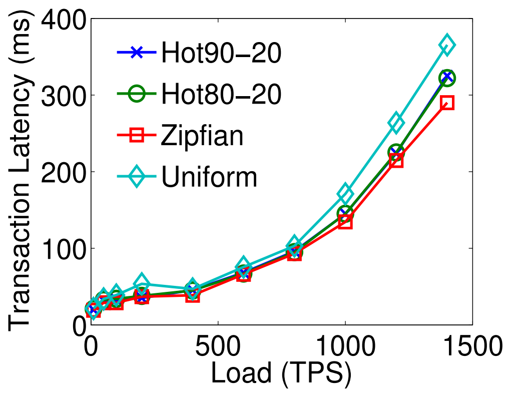
4.5.3 Single tenant behavior

We first evaluate the performance of ElasTraS for a single tenant in isolation, i.e., an OTM serving only one tenant whose database is contained within a partition. This allows us to analyze the performance of the different components in the design, i.e., the transaction manager, cache, and interactions with the storage layer.

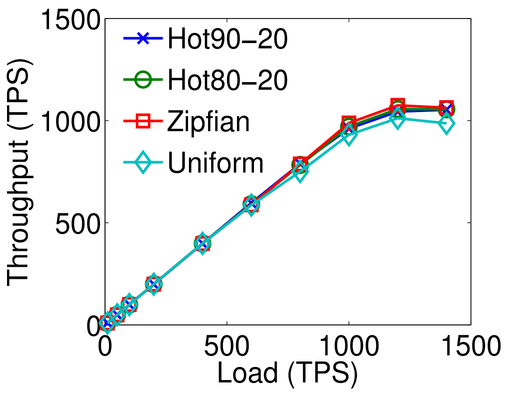
Yahoo! Cloud Serving Benchmark

Table 4.1 summarizes the default values of some YCSB parameters. To evaluate the performance of a single tenant in isolation, we vary different YCSB parameters while using the default values for the rest of the parameters. The YCSB parameters varied in these experiments are: number of operations in a transaction or **transaction size**, number of read operations in a transaction or **read/write ratio**, database size, number of transactions issued per second (TPS) or **offered load**, and cache size. The data items accessed by a transaction are chosen from a distribution; we used Zipfian (co-efficient 1.0), uniform, and hotspot. A hotspot distribution simulates a hot spot where $x\%$ of the operations access $y\%$ of the data items. We used a hot set comprising 20% of the database and two workload variants with 80% and 90% of the operations accessing the hot set.

Figure 4.4 plots the transaction latency and throughput as the load on the tenant varied from 20 to 1400 TPS. Until the peak capacity is reached, the throughput increases linearly with the load and is accompanied by a gradual increase in transaction latency; limiting transactions to a single partition allows this linear scaling. The throughput



(a) Transaction response time.



(b) Transaction throughput.

Figure 4.4: Evaluating the impact of offered load on throughput and latency. This experiment uses YCSB and runs a single tenant database in isolation at an OTM.

plateaus at about 1100 TPS and a sharp increase in latency is observed beyond this point. The system thrashes for a uniform distribution due to high contention in the network arising from high cache miss percentage. Thrashing is observed in OCC only in the presence of heavy resource contention. This behavior is confirmed for the skewed distributions where the working set is mostly served from the database cache. A higher cache miss rate also results in higher latency for the uniform distribution.

An ElasTraS OTM is similar to a single node RDBMS engine and should therefore exhibit similar behavior. To compare the behavior of ElasTraS with that of a traditional RDBMS, we repeated the above experiment using two other open-source RDBMSs, MySQL v5.1 and H2 v$1.3.148$, running on the same hardware with the same cache settings.[2] A behavior similar to that shown in Figure 4.4 was observed; the only difference was the peak transaction throughput: MySQL's peak was about 2400 TPS and H2's peak was about 600 TPS.

As observed from Figure 4.4, the behavior of all skewed workloads is similar as long as the working set fits in the database cache. We therefore choose a hotspot distribution with 90% operations accessing 20% data items (denoted by Hot90-20) as a representative skewed distribution for the rest of the experiments. Since our applied load for the remaining experiments (200 TPS) is much lower than the peak, the throughput is equal to the load. We therefore focus on transaction latency where interesting trends are observed.

Figure 4.5 plots the impact of transaction size; we vary the number of operations from 5 to 30. As expected, the transaction latency increases linearly with a linear

[2]MySQL: `www.mysql.com/`, H2: `www.h2database.com`.

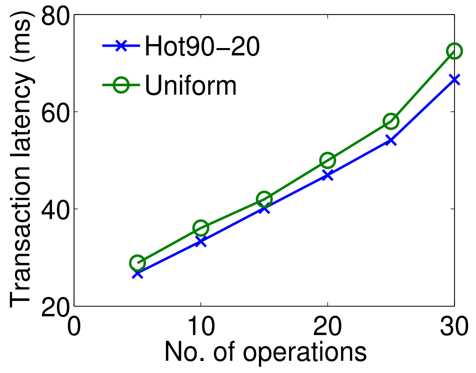


Figure 4.5: Impact of transaction size on response times.

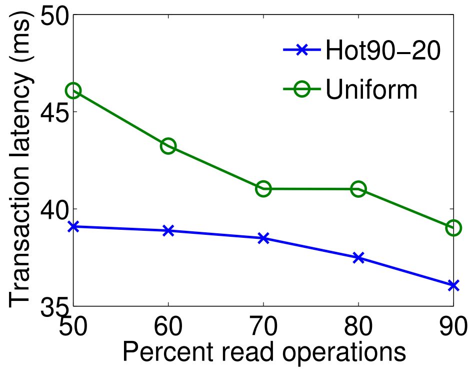


Figure 4.6: Impact of read percent on response times.

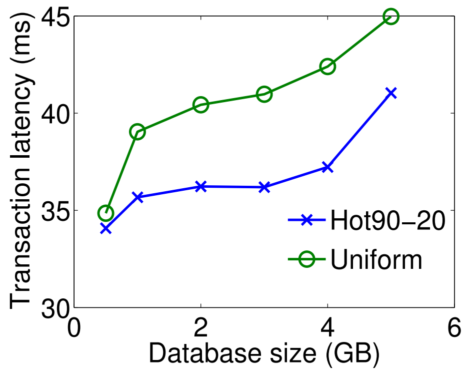


Figure 4.7: Impact of database size on response times

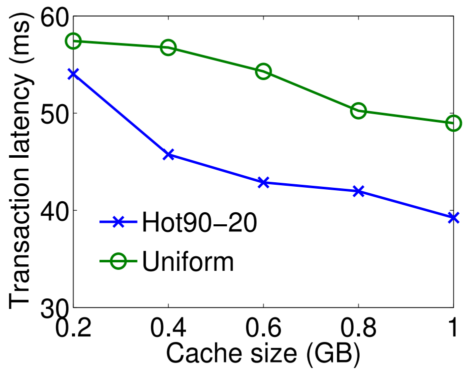


Figure 4.8: Impact of cache size on response times.

increase in the transaction size. Transaction latency also increases as the percentage of update operations in a transaction increases from 10% to 50% (see Figure 4.6). This increase in latency is caused by two factors: first, due to the append-only nature of the storage layer, more updates results in more fragmentation of the storage layer, resulting in subsequent read operations becoming more expensive. The latency increase is less significant for a skewed distribution since the more recent updates can often be found in the cache thus reducing the effect of fragmentation. Second, a higher fraction of write operations implies a more expensive validation phase for OCC, since a transaction must be validated against all write operations of concurrently executing transactions.

Figure 4.7 plots the effect of database size on transaction latency; we varied the database size from 500 MB to 5 GB. For a database size of 500 MB, the entire database fits into the cache resulting in comparable transaction latencies for both uniform and

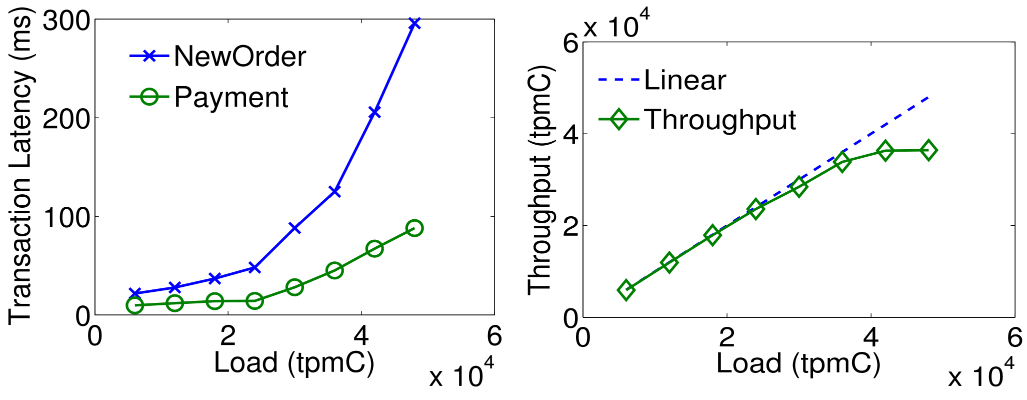


Figure 4.9: Evaluating the performance of a single tenant in isolation using TPC-C.

skewed distributions. As the database size increases, a uniform access distribution results in more cache misses and hence higher transaction latency. When the database size is 5 GB, the working set of the skewed distribution also does not fit in the result, resulting in a steep increase in latency. Similarly, when varying the cache size, transaction latency decreases with an increase in cache size resulting from higher cache hit ratios (see Figure 4.8 where we varied the cache size from 200 MB to 1 GB).

TPC-C Benchmark

In this experiment, we use a TPC-C database with ten warehouses resulting in a data size of 2.5 GB. The cache size is set to 1 GB. Figure 4.9 plots the average transaction latency and throughput as the TPC-C metric of transactions per minute (tpmC), where a behavior similar to that in YCSB is observed. We report the latency of NEWORDER and PAYMENT transactions that represent 90% of the TPC-C workload. The NEWORDER is a long read/write transaction accessing multiple tables while Payment is a short read-/write transaction accessing a small number of tables. Hence the payment transaction has a lower latency compared to NEWORDER. The throughput is measured for the entire benchmark workload. The gains from executing transactions as stored procedures is evident from the lower latency of the more complex NEWORDER transaction (executing close to a hundred operations per transaction) compared to that of the simpler transactions in YCSB (executing 10-20 operations).

4.5.4 Multitenancy at a single OTM

We now evaluate performance of multiple tenants hosted at the same OTM, thus evaluating the tenant isolation provided by the shared process multitenancy model. We

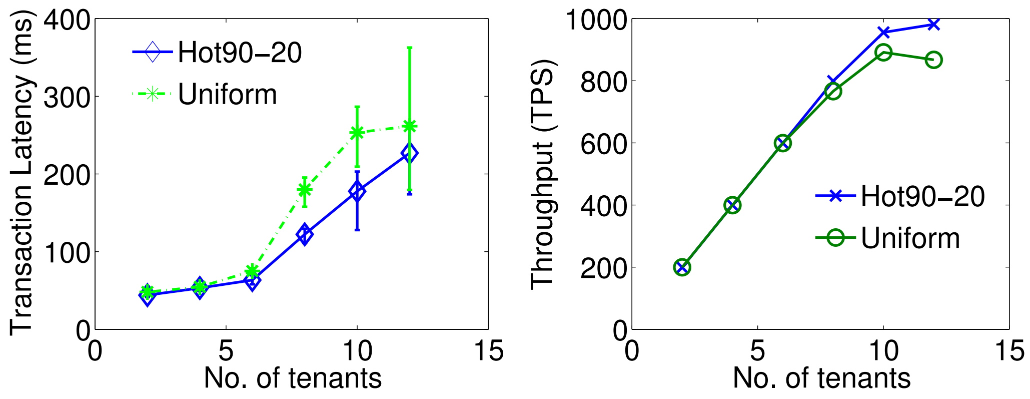


Figure 4.10: Evaluating performance when multiple tenants shared an OTM. In this experiment, each tenant's workload is an instance of YCSB. Multiple concurrent YCSB instances simulate a multitenant workload.

use YCSB for these experiments. Figure 4.10 reports transaction latency and throughput at one OTM. In the plot for transaction latency, the data points plotted correspond to the average latency across all tenants, while the error bars correspond to the minimum and maximum average latency observed for the tenants. Each tenant database is about 500 MB in size and the total cache size is set to 2.4 GB. The load on each tenant is set to 100 TPS. We increase the number of tenants per OTM, thus increasing the load on the OTM. For clarity of presentation, we use a hot spot distribution where 90% of the operations access 20% of the data items as a representative skewed distribution, and a uniform distribution. For skewed workloads when the working set fits in the cache, even though different tenants compete for the OTM resources, the performance is similar to that of a single tenant with the same load. The only observable difference is for uniform distribution at high loads where high contention for the network and storage results in a higher variance in transaction latency. Similarly, the peak throughput is also similar to that observed in the experiment for a single tenant (see Figure 4.4). This experiment demonstrates the good isolation provided by ElasTraS.

We now analyze the effectiveness of ElasTraS in handling large numbers of tenants with small data footprints. In this experiment, each OTM serves an average of 100 tenants each between 50 MB to 100 MB in size. We evaluate the impact on transaction latency and aggregate throughput per OTM as the load increases (see Figure 4.11). Even when serving large numbers of tenants that contend for the shared resources at the OTM, a similar behavior is observed when compared to the experiment with a single isolated tenant or one with a small number of tenants. Even though the peak throughput of an OTM is lower when compared to the other scenarios, transaction latencies remain comparable to the experiments with a much smaller number of tenants. The

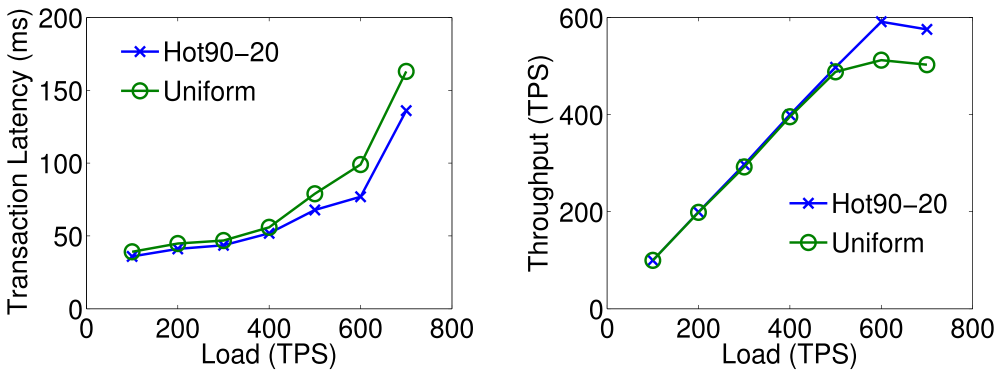


Figure 4.11: Performance evaluation using YCSB for large numbers of small tenants sharing resources at an OTM. In this experiment, each tenant's workload is an instance of YCSB. Multiple concurrent YCSB instances simulate a multitenant workload.

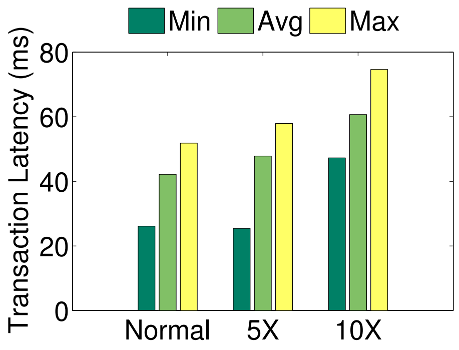
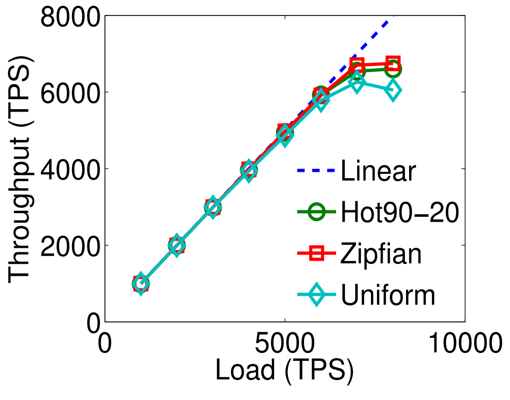


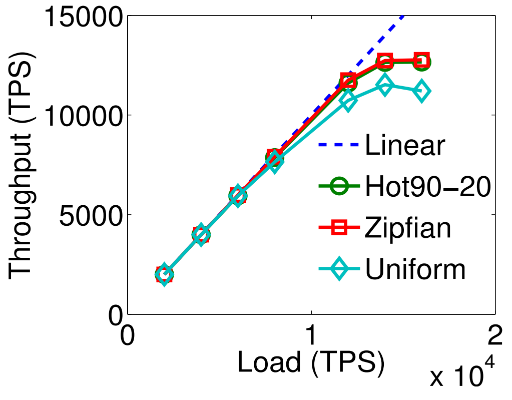
Figure 4.12: Impact of a heavily loaded tenant on other co-located tenants.

reduction in throughput arises from the overhead due to the large number of network connections, large number of transaction and data managers, and higher contention for OTM resources. This experiment demonstrates that ElasTraS allows considerable consolidation where hundreds of tenants, each with small resource requirement, can be consolidated at a single OTM.

When co-locating multiple tenants at an OTM, it is also important that the system effectively isolates a heavily loaded tenant from impacting other co-located tenant. Figure 4.12 shows the impact of an overloaded tenant on other co-located tenants. In this experiment, we increase the load on a tenant by $5\times$ and $10\times$ the load during normal operation and analyze the impact on the transaction latency of other co-located tenants. The OTM under consideration is serving 10 tenants each with an average load of 25



(a) 10 node cluster.



(b) 20 node cluster.

Figure 4.13: Aggregate system throughput using YCSB for different cluster sizes.

TPS during normal operation (an aggregate load of 250 TPS at the OTM). The first group of bars (labeled 'Normal') plots the minimum, average, and maximum transaction latency for the tenants during normal operation. The next set of bars plots the latencies when the load on one of the tenants is increased by $5\times$ from 25 TPS to 125 TPS. As is evident from Figure 4.12, this sudden increase in load results in less than 10% increase in the average transaction latency of other tenants co-located at the same OTM. A significant impact on other tenants is observed only when the load on a tenant increases by $10\times$ from 25 TPS to 250 TPS; even in this case, the increase in transaction latency is only about 40%. This experiment demonstrates the benefits of having an independent transaction and data manager per tenant; the impact on co-located tenants arise from contention for some shared resources like the CPU and the cache. ElasTraS's use of the shared process multitenancy model therefore provides a good balance between effective resource sharing and tenant isolation.

4.5.5 Overall system performance

Figure 4.13 plots the aggregate throughput for different cluster sizes and number of tenants served, as the load on the system increases. We use YCSB for this experiment. Each OTM on average serves 50 tenants; each tenant database is about 200 MB and the cache size is set to 2.4 GB. Experiments were performed on EC2 using `c1.xlarge` instances. Since there is no interaction between OTMs and there is no bottleneck, the system is expected to scale linearly with the number of nodes. This (almost) linear scaling is evident from Figure 4.13 where the peak capacity of the twenty node cluster is almost double that of the 10 node cluster; the 20 node cluster serves about 12K TPS compared to about 6.5K TPS for the 10 node cluster. The choice of co-locating a

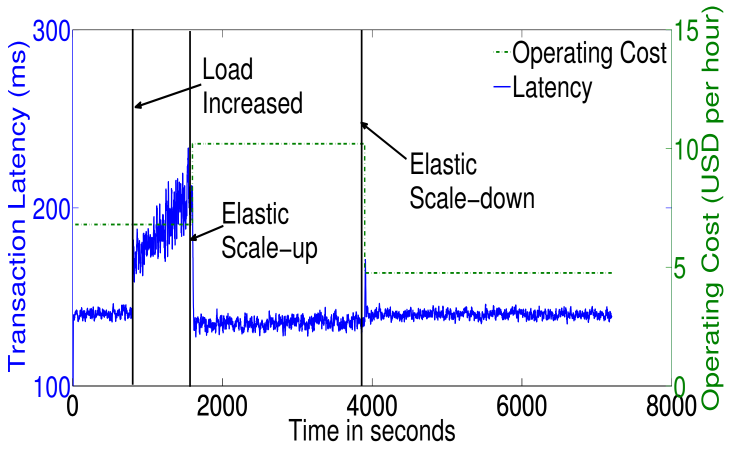


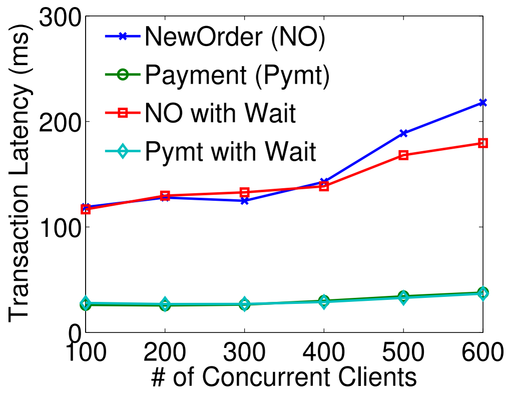
Figure 4.14: Evaluating the benefits of elastic scaling.

tenant's data within a single partition allows ElasTraS to scale linearly. The reason for the small reduction in per-node throughput in the 20 node cluster compared to the 10 node cluster is, in part, due to the storage layer being co-located with the OTMs, thus sharing some of the resources available at the servers.

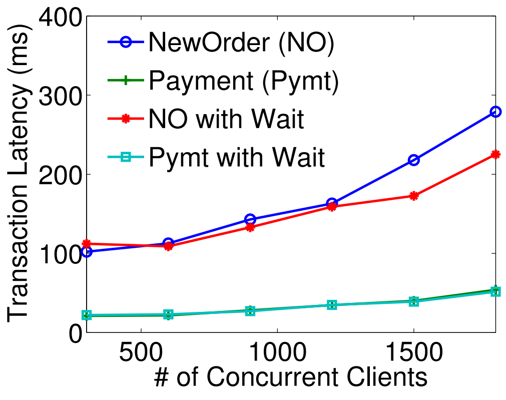
The peak throughput of 12K TPS in an ElasTraS cluster of 20 OTMs serving 1000 tenants corresponds to about a billion transactions per day. Though not directly comparable, putting it in perspective, Database.com (the "cloud database" serving the Salesforce.com application platform) serves 25 billion transactions per quarter hosting about $80,000$ tenants [75].

4.5.6 Elasticity and Operating Cost

ElasTraS uses Albatross for low cost live database migration. We now evaluate the effectiveness of Albatross in elastic scaling of the overall system; Figure 4.14 illustrates this effect as OTMs are added to (or removed from) the system. We vary the overall load on the system during a period of two hours using YCSB. Along the x-axis, we plot the time progress in seconds, the primary y-axis (left) plots the transaction latency (in ms) and the secondary y-axis (right) plots the operating cost of the system (as per the cost of `c1.xlarge` EC2 instances). We plot the moving average of latency averaged over a 10 second window. The number of tenants is kept constant at 200 and the load on the system is gradually increased. The experiment starts with a cluster of 10 OTMs. As the load on the system increases, the transaction latency increases (shown in Figure 4.14). When the load increases beyond a threshold, elastic scaling is triggered and five new OTMs are added to the cluster. The TM Master re-balances the tenants to the



(a) 10 nodes, 1000 warehouses



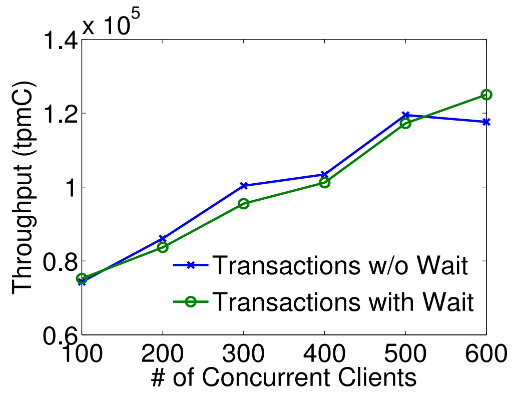
(b) 30 nodes, 3000 warehouses

Figure 4.15: Scaling-out a single tenant by partitioning and scaling to different cluster sizes and varying number of clients. This experiment reports the impact on transaction latency as the load on the system increases.

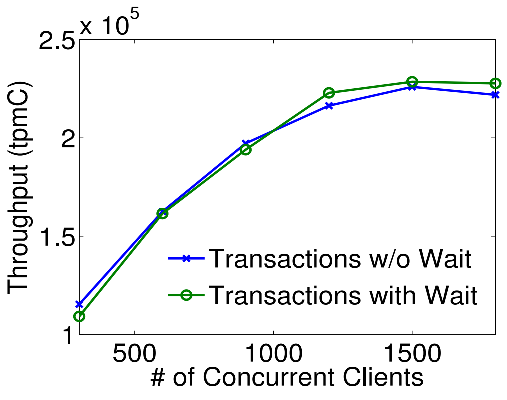
new partitions using Albatross. The ElasTraS client library ensures that the database connections are re-established after migration. After load-balancing completes, load is distributed on more OTMs resulting in reduced average transaction latency. When the load decreases below a threshold, elastic scaling consolidates tenants to 7 OTMs (also shown in Figure 4.14). Since the number of OTMs is reduced, the operating cost of the system also reduces proportionately. This experiment demonstrates that elastic scaling using low overhead migration techniques can effectively reduce the system's operating cost with minimal impact on tenant performance.

4.5.7 Scaling large tenants

In all our previous experiments, we assume that a tenant is small enough to be contained within a single partition and served by a single OTM. In this experiment, we evaluate the scale-out aspect of ElasTraS for a single tenant that grows big. We leverage schema level partitioning to partition a large tenant. The partitions are then distributed over a cluster of OTMs. In this experiment, we vary the number of OTMs from 10 to 30. As the number of OTMs is increased, the size of the database is also increased proportionally and so is the peak offered load. We use the TPC-C benchmark for this experiment and leverage the inherent tree schema for partitioning. As per the original TPC-C benchmark specification, about 15% transactions are not guaranteed to be limited to a single partition. In our evaluation, all TPC-C transactions access a single partition. For a cluster with 10 OTMs, the database is populated with 1000 warehouses and the number of clients is increased from 100 to 600 concurrent clients in steps of



(a) 10 nodes, 1000 warehouses



(b) 30 nodes, 3000 warehouses

Figure 4.16: Scaling-out a single tenant by partitioning and scaling to different cluster sizes and varying number of clients. This experiment reports the impact on transaction throughput as the load on the system increases.

100. For a cluster with 30 OTMs, the database size is set to 3000 warehouses (about 2 TB on disk), and the number of concurrent clients is varied from 300 to 1800.

Figure 4.15 plots the transaction latencies of the NewOrder and Payment transactions and the aggregate system throughput. Figure 4.16 plots the aggregate throughput of the system in terms of the TPC-C metric of transactions per minute C (tpmC). We used two variants of transactional workload: (i) the clients do not wait between submitting transaction requests, and (ii) the clients wait for a configurable amount of time ($10 - 50$ms) between two consecutive transactions. As per TPC-C specifications for generating the workload, each client is associated with a warehouse and issues requests only for that warehouse. As expected, transaction latency increases as the load increases, a behavior similar to the previous experiments. These experiments also demonstrate that in addition to serving large numbers of small tenants, ElasTraS can efficiently scale-out large tenants serving thousands of concurrent clients and sustaining throughput of more than 0.2 million transactions per minute. Putting it in perspective, the performance of ElasTraS is comparable to many commercial systems that have published TPC-C results (`http://www.tpc.org/tpcc/results/tpcc_results.asp`).

4.5.8 Analysis and Lessons Learned

Our evaluation of the ElasTraS prototype using a wide variety of workloads and scenarios show that when the working set fits in the cache, the high cost of accessing the de-coupled storage layer in ElasTraS is amortized by caching at the OTM. Since for most high performance OLTP systems, the working set must fit the cache such that the peak throughput is not limited by disk I/O, this architecture will suit most practical OLTP workloads. On the other hand, using a de-coupled storage layer allows migrating partitions with minimal performance impact. Using separate TMs for each tenant partition allows effective isolation between the tenants. Limiting small tenants to a single database process obviates distributed synchronization and lends good scalability to the design. On one hand, the shared process multitenancy model in ElasTraS provides effective resource sharing between tenants allowing consolidation of hundreds of small tenants at an OTM. On the other hand, schema level partitioning allows scaling-out large tenants.

Our experiments revealed an important aspect in the design—whether to share the cache among all the partitions (or tenants) at the OTM or provide an exclusive cache to each tenant. It is a trade-off between better resource sharing and good tenant isolation. In our current implementation, all OTMs share the cache at an OTM. This allows better sharing of the RAM while allowing the tenants to benefit from temporal distributions in access—when a tenant requires a larger cache, it can steal some cache space from other tenants that are not currently using it. However, when multiple tenants face a sudden increase in cache requirements, this lower isolation between tenants results in contention for the cache. In such a scenario, some tenants suffer due to a sudden change in behavior of a co-located tenant, which is not desirable in a multitenant system that must guarantee every tenant's SLAs. In the future, we plan to explore the choice of having an exclusive cache for each partition. We also plan to explore modeling tenant behavior to determine which cache sharing scheme to use for a given set of tenants co-located at an OTM.

Another important aspect is whether to store an OTM's transaction log on a disk locally attached to the OTM or in the DFS; our current implementation stores the log in the DFS. Storing the log in the DFS allows quick recovery after an OTM failure, since the partitions of the failed OTM can be recovered independent of the failed OTM. However, in spite of using group commit, DFS appends are slower compared to that of a local disk which limits the peak OTM throughput and places a higher load on the network. It is, therefore, a trade-off between performance during normal operation and recovery time after a failure.

Finally, it is also important to consider whether to co-locate an OTM and a storage layer daemon, if possible. Since the OTMs do not result in disk I/O for any locally attached disks, co-locating the storage layer with the transaction management layer allows effective resource usage. As CPUs become even more powerful, they will have abundant capacity for both the OTMs as well as storage layer daemons.

During the experiments, we also learnt the importance of monitoring the performance of the nodes in a system, in addition to detecting node failures, especially in a shared (or virtualized) cluster such as EC2. There were multiple instances in an EC2 cluster when we observed a node's performance deteriorating considerably, often making the node unusable. Detecting such unexpected behavior in the cluster and replacing the node with another new node or migrating partitions to other OTMs is important to ensure good performance. We augmented our system models with node-specific performance measures to detect such events and react to them.

4.6 Summary

In this chapter, we presented ElasTraS, an elastically scalable transaction processing system to meet the unique requirements of a cloud platform. ElasTraS leverages the design principles of scalable Key-value stores and decades of research in transaction processing, thus resulting in a scale-out DBMS with transactional semantics. ElasTraS can effectively serve large numbers of small tenants while scaling-out large tenants using schema level partitioning. ElasTraS allows effective resource sharing between multiple partitions (or tenant databases) co-located at the same OTM while scaling-out to large numbers of partitions. ElasTraS operates at the granules of database partitions which are the granules of assignment, load balancing, and transactional access. For applications with a small data footprint, a partition in ElasTraS can be a self contained database. For larger application databases, ElasTraS uses schema level partitioning to effectively scale-out while supporting rich transactional semantics.

We demonstrated that a prototype implementation deployed on a cluster of commodity servers can efficiently serve thousands of tenants while sustaining aggregate throughput of billions of transactions per day. We further demonstrated that while having performance similar to RDBMSs, ElasTraS can achieve lightweight elasticity and effective multitenancy, two critical facets of a multitenant database for cloud platforms.

Chapter 5

Dynamically Defined Partitions

"Coming together is a beginning; keeping together is progress; working together is success."

– Henry Ford.

In the previous chapter, we presented the design of a system that allowed scale-out for databases that can be statically partitioned. Many applications, however, have fast evolving access patterns, thus negating the benefits of static partitioning based on access patterns. Consider the example of an online multi-player game, such as an online casino. An instance of the game has multiple players and the gaming application requires transactional access to the player profiles while the game is in progress. For example, every player profile might have an associated balance (in a real or virtual currency) and the balance of all players must be transactionally updated as the game proceeds. To allow efficient execution of these transactions, it is imperative that the data items corresponding to the profiles are co-located within a database partition. However, a game instance lasts for small periods of time as players move from one game to another. Furthermore, over a period of time, a player might participate in game instances involving different groups of players. Therefore, the group of data items on which the application requires transactional access changes rapidly over time. Moreover, as the game becomes popular, there can be hundreds of thousands of similar independent game instances that concurrently exist [53, 86]. A DBMS serving these applications must therefore scale to large numbers of concurrent transactions.

In a statically partitioned system, the profiles of the players participating in a game instance might belong to different partitions. Providing transaction guarantees across these groups of player profiles will result in distributed transactions. The challenge is to design a scalable DBMS to support such applications that provides the benefits of partitioning (i.e., efficient non-distributed transaction execution) even when the access patterns do not statically partition. In this chapter, we present our solution to this problem. Ours is the first solution, and to the best of our knowledge the only published solution, to this problem.

We present the Key Group abstraction that allows the application to dynamically specify the data items on which it wants transactional access [33]. A Key Group, therefore, is a dynamically defined granule of transactional access. To allow efficient transaction execution on a Key Group, we propose the Key Grouping protocol, that co-locates the read/write accesses rights (or **ownership**) of the data items part of a Key Group at a single node. In essence, the Key Grouping protocol is a lightweight mechanism to re-organize data ownership without actually moving the data. The Key Grouping protocol ensures that this transfer of ownership is completed safely even in the presence of message or node failures. The Key Group abstraction and the Key Grouping protocol are designed to operate as layer on top of a statically partitioned distributed DBMS. [1]

5.1 The Key Group Abstraction

5.1.1 Key Groups

Key Group is a powerful yet flexible abstraction for applications to define the granules of transactional access. Any data item (or key) in the data store can be selected as part of a Key Group. The Key Groups are transient; the application can dynamically create (and dissolve) the groups. For instance, in the multi-player gaming application, a Key Group is created at the start of a game instance and deleted at its completion. At any instant of time, a given key is part of a single group. However, a key can participate in multiple groups whose lifetimes are temporally separated. For instance, a player can participate in a single game at any instant of time, but can be part of multiple game instances that do not temporally overlap.

Transactional guarantees are provided only for keys that are part of a group, and only during the lifetime of the group. All keys in the data store need not be part of groups. At any instant, multiple keys might not be part of any group; they conceptually form one-member groups.

Every group has a **leader** selected from one of the member keys in the group; the remaining members are called the **followers**. The leader is part of the group's identity. However, from an application's perspective, the operations on the leader are no different

[1]An earlier version of the work reported in this chapter was published as the paper entitled "G-Store: a scalable data store for transactional multi key access in the cloud" in the proceedings of the 2010 ACM International Conference on Symposium on Cloud Computing (SoCC). DOI: `http://doi.acm.org/10.1145/1807128.1807157`.

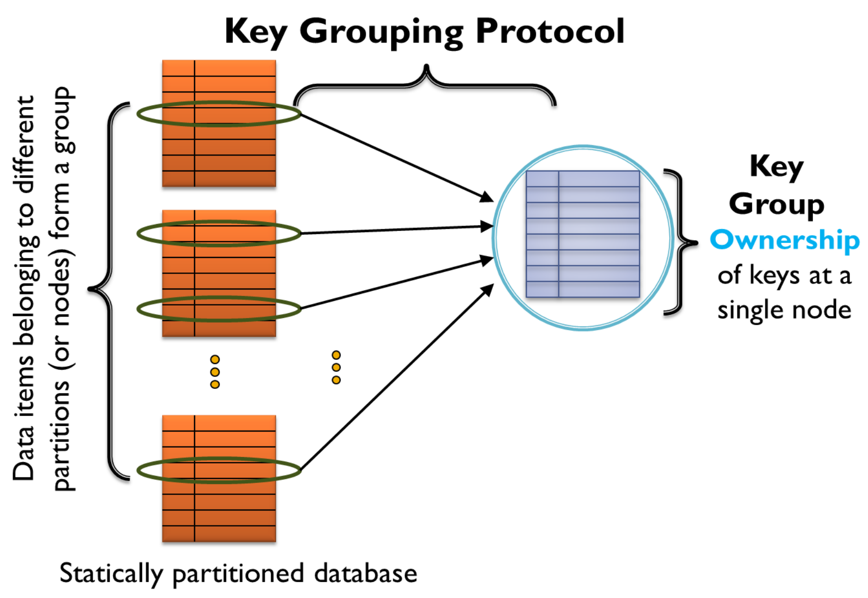

Key Grouping Protocol
Data items belonging to different partitions (or nodes) form a group
Key Group Ownership of keys at a single node
Statically partitioned database

Figure 5.1: An illustration of the Key Group abstraction and the subsequent co-location of the Key Group's members to efficiently support transactions on the Key Group.

from those supported for the followers. For ease of exposition, in the rest of the chapter, we will use the terms leader and follower to refer to the data items as well as the nodes where the data items are stored. Even though keys are often co-located at a node in a Key-value store, their co-location is by *chance* and not by *choice*. Therefore, these keys are treated as independent.

5.1.2 Design Rationales

The Key Group abstraction captures an application-defined relationship amongst the members of a group. Once this relationship is established, ownership of the keys in a Key Group is co-located at a single node by *choice*, thus limiting transactions on a Key Group to a single node. As postulated earlier, this co-location of the data items is the critical to efficient transaction execution and high scalability.

Figure 5.1 provides an illustration of how the Key Group abstraction allows ownership co-location. The leader key's owner (called the **leader node**) is assigned ownership of the group and the nodes corresponding to the followers **yield** ownership to the leader, i.e., the leader node gains exclusive ownership to all the followers. Once this ownership transfer is complete, all read/write accesses for the group members is served by the leader. Note that group creation, conceptually, transfers the ownership of the followers to the leader while the followers' data remain with nodes that originally owned the followers (i.e., the **follower nodes**). A Key Group, therefore, decouples the ownership of the data items from their storage, thus allowing a lightweight re-organization of ownership.

Ownership transfer from followers to the leader during group creation, and the opposite during group deletion, must be resilient to failures or other dynamic changes of the underlying system. Therefore, we propose the Key Grouping protocol to guarantee correct ownership transfer. The Key Grouping protocol involves distributed synchronization between the leader and the followers. However, this distributed synchronization is limited to only the group creation and deletion phases; transactions executing on the group execute locally at the leader. The rationale is to limit distributed synchronization only to update the ownership information which is part of the system state. The cost of this distributed synchronization is amortized by the efficient transaction execution during the group's lifetime.

5.2 The Key Grouping Protocol

Group creation is initiated by an application client (or a client) sending a *group create* request with the **group id** and the members. The group id is a combination of a unique system generated id and the leader key. Group creation can either be **atomic** or **best effort**. Atomic group creation implies that either all members join the group or else the group is automatically deleted if one of the followers did not join. Best effort creation forms the group with whatever keys that joined the group. A data item might not join a group either if it is part of another group (since we require groups to be disjoint), or if the data item's follower node is not reachable.

In the Key Grouping protocol, the leader is the *coordinator* and the followers are the *participants* or *cohorts*. The leader key can either be specified by the client or is selected by the system. The group create request is routed to the node which owns the leader key. The leader logs the member list, and sends a *Join Request* ($\langle \mathtt{J} \rangle$) to all the followers (i.e., each node that owns a follower key). Once the group creation phase terminates successfully, the client can issue operations on the group. When the client wants to disband the group, it initiates the group deletion phase with a *group delete* request.

Conceptually, ownership transfer from followers to the leader is equivalent to the leader acquiring "locks" on the followers. Similarly, the reverse process is equivalent to releasing the "locks". The Key Grouping protocol is thus reminiscent of the locking protocols for transaction concurrency control [40, 87]. The difference is that in our approach, the locks are held by the Key Groups (i.e., the system) whereas in classical lock based schedulers, the locks are held by the transactions.

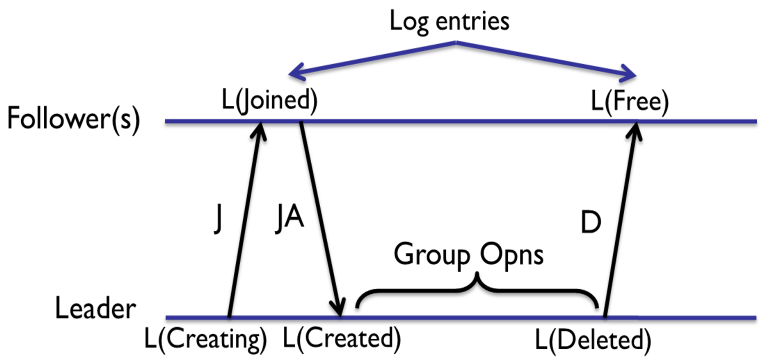


Figure 5.2: Key Grouping protocol with reliable messaging.

In the rest of the section, we discuss two variants of the Key Grouping protocol assuming different message delivery guarantees and show how group creation and deletion can be safely done in the presence of various failure and error scenarios. In our initial description of the protocol, we assume no failures. We then describe protocol operation in the presence of message failures and node failures.

5.2.1 Protocol with Reliable Messaging

We first describe the Key Grouping protocol assuming reliable messaging, i.e., the messages are not lost, duplicated, or reordered. The leader requests key ownership from the followers (*Join Request* $\langle \texttt{J} \rangle$), and depending on availability, ownership is either transferred to the leader or the request is rejected (*Join Ack* $\langle \texttt{JA} \rangle$). The $\langle \texttt{JA} \rangle$ message indicates whether the follower joined the group. Depending on the application's requirements, the leader can inform the application as members join the group or when the group creation phase has terminated. The group deletion phase involves notifying the followers with a *Delete Request* $\langle \texttt{D} \rangle$. Figure 5.2 illustrates the protocol in a failure free scenario.

5.2.2 Protocol with Unreliable Messaging

Reliable message delivery guarantees are often expensive. For example, protocols such as TCP provide guaranteed delivery and ordering only on an active connection. However, group creation requires delivery guarantees across connections. Hence, using TCP alone will not be enough to provide message delivery guarantees in the presence of node failures or network partitions. We now present a variant of the Key Grouping protocol that does not require any message delivery guarantees. The basics of the protocol

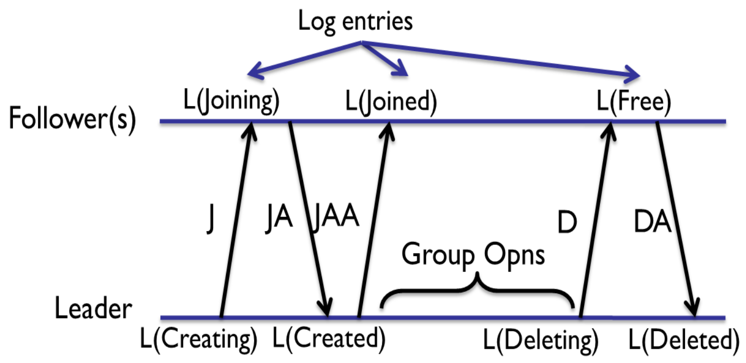


Figure 5.3: Key Grouping protocol with unreliable messaging.

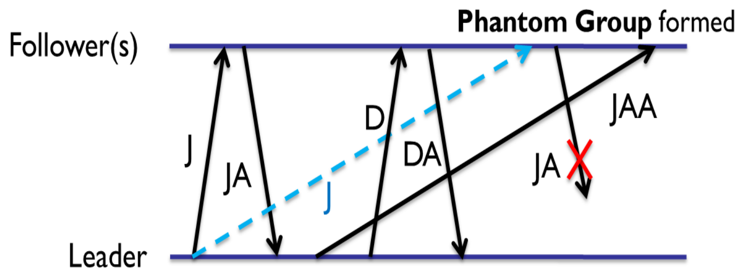


Figure 5.4: Message failures resulting in creation of a *phantom group*.

remain similar to that described in Section 5.2.1. Additional messages are incorporated to deal with message failures.

Figure 5.3 illustrates the protocol with unreliable messaging which, in the steady state, results in two additional messages, one during creation and one during deletion. During group creation, the ⟨`JA`⟩ message, in addition to notifying whether a key is free or part of a group, acts as an acknowledgement for the ⟨`J`⟩ request. On receipt of a ⟨`JA`⟩, the leader sends a *Join Ack Ack* ⟨`JAA`⟩ to the follower, the receipt of which finishes the group creation phase for that follower. This group creation phase is two phase, and is similar to the 2PC protocol [44] for transaction commitment. The difference stems from the fact that our protocol also allows best effort group creation while 2PC would be equivalent to atomic group creation. During group dissolution, the leader sends a *Delete Request* ⟨`D`⟩ to the followers. On receipt of a ⟨`D`⟩ the follower regains ownership of the key, and then responds to the leader with a *Delete Ack* ⟨`DA`⟩. The receipt of ⟨`DA`⟩ from all the followers completes group deletion.

Though simple in scenarios without failures, group creation can be complicated in the presence of message failures, such as lost, reordered, or duplicated messages. For example, consider the pathological scenario in Figure 5.4 which has message loss, duplication, and reordering in a protocol instance where the *group delete phase* overlaps with the *group create phase*. The leader sends a $\langle$`J`$\rangle$ message to a follower and this $\langle$`J`$\rangle$ message is duplicated in the network due to some error (the duplicate message is shown as a dashed line). The follower receives a copy of the $\langle$`J`$\rangle$ message, and replies with the $\langle$`JA`$\rangle$ message. On receipt of a $\langle$`JA`$\rangle$, the leader sends a $\langle$`JAA`$\rangle$ which is delayed in the network and is not immediately delivered to the follower. In the meantime, the client requests deletion of the group, and the leader sends out the $\langle$`D`$\rangle$ message. The follower, before receiving the $\langle$`JAA`$\rangle$, responds to the $\langle$`D`$\rangle$ message with a $\langle$`DA`$\rangle$ message and the group has been deleted from the follower's perspective. Once the leader receives the $\langle$`DA`$\rangle$, it also purges the state of the group. After a while, the follower receives the duplicate $\langle$`J`$\rangle$ message for the deleted group, and sends out a $\langle$`JA`$\rangle$ for the group which is lost in the network. The follower then receives the delayed $\langle$`JAA`$\rangle$ and its receipt completes a group creation phase for the follower. In this pathological scenario, the follower has yielded control to a non-existent group, or a **phantom group**, which has no leader. Since the follower's $\langle$`JA`$\rangle$ message was lost, the leader is not aware of the existence of this phantom group. Creation of a phantom group makes data items unavailable due to a non-existent leaders and hence non-existent owners of data items. The Key Grouping protocol must therefore provide mechanisms to detect and tolerate *duplicate* and *stale* messages and ensure that phantom groups, if any, are detected and deleted.

We now describe the Key Grouping protocol in detail, and then explain how various message failure scenarios are handled. We first describe the actions taken by the nodes on receipt of the different protocol messages. We then describe the handling of lost, reordered, and duplicate messages.

Group creation phase

Group create request. On receipt of a group create request from the client, the leader verifies the request for a unique group id. The leader appends an entry to its *log* that stores the group id and the members in the group. After the log entry is **forced** (i.e., flushed to persistent storage), the leader sends a $\langle$`J`$\rangle$ request to each of the follower nodes. The $\langle$`J`$\rangle$ messages are retried until the leader receives a $\langle$`JA`$\rangle$ from the followers.

Join request $\langle$`J`$\rangle$. On receipt of a $\langle$`J`$\rangle$ request at a follower, the follower ascertains the freshness and uniqueness of the message. If the message is detected as a duplicate, then the follower sends a $\langle$`JA`$\rangle$ without appending any log entry. Otherwise, if the follower key is not part of any active group, the follower appends a log entry denoting the ownership transfer and the identity of the leader key. This ownership transfer is an update to the system's metadata and the follower's log is the persistent storage for this information. This log entry must therefore be forced before a reply is sent. The follower's state is set to *joining*. The follower then replies with a ⟨JA⟩ message notifying its intent to yield. To deal with spurious ⟨JAA⟩ messages and eliminate the problem of phantom groups, the follower should be able to link the ⟨JAA⟩ to the corresponding ⟨JA⟩. This is achieved by using a sequence number generated by the follower called the **yield id**. A yield id is associated to a follower node and is monotonically increasing. The yield id is incremented every time a follower sends new ⟨JA⟩ and is logged along with the entry logging the ⟨J⟩ message. The yield id is copied into the ⟨JA⟩ message along with the group id. The ⟨JA⟩ message is retried until the follower receives the ⟨JAA⟩ message. This retry ensures that the phantom groups are not left undetected.

Join Ack ⟨JA⟩. On receipt of a ⟨JA⟩ message, the leader checks the group id. If it does not match the identifiers of any of the currently active groups, then the leader sends a ⟨D⟩ message and does not log this action or retry this message. Occurrence of this event is possible when the message was a delayed message, or the follower yielded to a delayed ⟨J⟩. In either case, a ⟨D⟩ message would be sufficient and also deletes any phantom groups that might have been formed. If the group id matches a current group, then the leader sends a ⟨JAA⟩ message copying the yield id from the ⟨JA⟩ to the ⟨JAA⟩ irrespective of whether the ⟨JA⟩ is a duplicate. If this is the first ⟨JA⟩ received from that follower for this group, a log entry is appended to indicate that the follower has joined the group; however, the leader does not need to force the entry. The ⟨JAA⟩ message is never retried, and the loss of ⟨JAA⟩ messages is handled by the retries of the ⟨JA⟩ message. The receipt of ⟨JA⟩ messages from all the followers terminates the group creation phase at the leader.

Join Ack Ack ⟨JAA⟩. On receipt of a ⟨JAA⟩ message, the follower checks the group id and yield id to determine freshness and uniqueness of the message. If the yield id in the message does not match the expected yield id, then this ⟨JAA⟩ is treated as a spurious message and is ignored. This prevents the creation of phantom groups. In the scenario shown in Figure 5.4, the delayed ⟨JAA⟩ will have a different yield id since it corresponds to an earlier group. Hence, the follower will reject it as a spurious message, thus preventing the creation of a phantom group. If the message is detected to be unique and fresh, then the follower key's state is set to *joined*. The follower node logs this event, which completes the group creation process for the follower; the log entry does not need to be forced.

Group deletion phase

Group delete request. When the leader receives the *group delete* request from the application client, it forces a log entry for the request and initiates the process of *yielding* ownership back to the followers. The leader then sends a ⟨D⟩ message to each follower in the group. The ⟨D⟩ messages are retried until all ⟨DA⟩ messages are received. At this point, the group has been marked for deletion and the leader will reject any future transactions accessing this group.

Delete request ⟨D⟩. When the follower receives a ⟨D⟩ request, it validates this message, and appends a log entry on successful validation of the message. This log entry signifies that it has regained ownership of the key. Since regaining ownership is a change in the system state, the log is forced after appending this entry. Irrespective of whether this ⟨D⟩ message was duplicate, stale, spurious, or valid, the follower responds with a ⟨DA⟩ message; this ⟨DA⟩ message is not retried.

Delete ack ⟨DA⟩. On receipt of a ⟨DA⟩ message, the leader checks for the validity of the message. If this is the first message from that follower for this group, and the group id corresponds to an active group, then a log entry is appended indicating that the ownership of the data item has been successfully transferred back to the follower. Once the leader has received a ⟨DA⟩ from all the followers, the group deletion phase terminates. The log is not forced on this protocol action.

Analysis

In the steady state without any failures, the group creation phase results in three messages and two log forces (one each at the leader and the follower) for every follower node. Since the ⟨JA⟩ and ⟨JAA⟩ are acknowledgements, they need not be forced; these entries are flushed as a side-effect of other force requests or when the log buffer fills up and is flushed. Similarly, the group deletion phase results in two messages and two log forces (one at the leader and one at the follower) for every follower; the log is not forced on receipt of a ⟨DA⟩ message.

Message loss, reordering, or duplication

Message related errors occur due to a variety of reasons, such as network equipment errors, partitions, and node failures. Lost messages are handled by associating timers with transmitted messages and retrying a message if an acknowledgement is not received before the timeout. For messages that do not have an associated acknowledgement (i.e., the ⟨JAA⟩ and ⟨DA⟩), the sender relies on the recipient's retry. For example, in the case of the ⟨JAA⟩ message, the follower is expecting the ⟨JAA⟩ as an acknowledgement of the ⟨JA⟩ message. Therefore, if a ⟨JAA⟩ message is lost, the follower will retry the ⟨JA⟩ message, on receipt of which, the leader will resend the ⟨JAA⟩. The loss of ⟨DA⟩ is handled in a similar way.

To detect duplicate and reordered messages, the protocol uses unique identifiers for all messages. All the messages have the group id, and the ordering of the messages within the group creation phase is achieved using a yield id which is unique within a group. Referring back to the scenario of Figure 5.4 where phantom groups are formed due to stale and duplicate messages, the follower uses the yield id to detect that the ⟨JAA⟩ is stale, i.e., from a previous instance of a ⟨JA⟩ message, and rejects this message. As a result, it will retry the ⟨JA⟩ message after a timeout. On receipt of the ⟨JA⟩ at the leader, it determines that this message is from a deleted or stale group, and replies with a ⟨D⟩ message. It is intuitive that the loss of messages does not cause issues since the messages will be retried by the follower. Therefore, group id and yield id together allow the protocol to deal with duplicate and reordered messages.

Concurrent group creates and deletes

A leader might receive a group delete request from the application before group creation has completed; the Key Grouping protocol handles such scenarios. For the followers that had received the ⟨J⟩ and/or ⟨JAA⟩, this is equivalent to a normal delete and is handled the same way a ⟨D⟩ is handled during normal operation. Special handling is needed for the followers that have not received the ⟨J⟩ message yet as a result of message loss or reordering. For those followers, the ⟨D⟩ message is for a group about which the follower has no knowledge. If the follower rejects this message, the leader might be blocked. To prevent this, the follower sends a ⟨DA⟩ to the leader. However, since the follower is unaware of the group, it cannot maintain any state associated with this message; maintaining this information in the log prevents garbage collection of this entry. When the follower receives the ⟨J⟩ request, it has no means to determine that this is a stale message and thus accepts it as a normal message and replies with a ⟨JA⟩. However, when the leader receives the ⟨JA⟩ for this non-existent group, it sends a ⟨D⟩ message which frees the follower and deletes the group at the follower. Similar to a normal ⟨JA⟩ message, this ⟨JA⟩ is retried. This retry handles the loss of ⟨JA⟩ or ⟨D⟩.

5.3 Transactions on a Key Group

The Key Grouping protocol ensures that the ownership of the group members are co-located at a node. Furthermore, since the leader node is the unique owner of the group's data, it can cache the data to serve the read and writes locally. This co-location

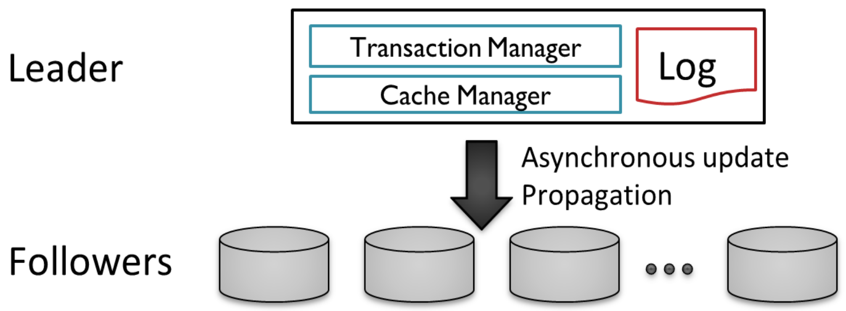


Figure 5.5: Efficient transaction execution on a Key Group.

and caching allows efficient single-site transaction execution. Figure 5.5 illustrates the transaction execution logic at the leader, similar to that of classical RDBMS storage engines [15, 87]. The transaction manager is responsible for concurrency control and recovery using the log. The cache manager is responsible fetching the data for the group members from their respective follower nodes, cache updates made by transactions executing during the lifetime of the group, and asynchronous propagation of the updates to the follower nodes that now act as storage nodes. The cache manager guarantees that the updates are propagated to the followers before a group is deleted. In this section, we provide the details of transaction management and propagation of the cached updates.

5.3.1 Transaction Manager

Once group creation completes, application clients can issue transactions accessing the data items within a Key Group. Transactions are routed to the leader. Standard concurrency control techniques, such as two phase locking (**2PL**), optimistic concurrency control (**OCC**), or snapshot isolation (**SI**), can be used to guarantee transaction isolation. To guarantee atomicity and durability of the updates made by a transaction, all update operations are appended to a *transaction log* that is logically separate from the log used for the Key Grouping protocol. Log entries for a transaction are buffered during transaction execution and before a transaction commits, a commit entry is appended and the log is forced. This log force ensures that the log entries are persistent. In the event of a failure, recovery on the transaction log guarantees atomicity and durability. Updates made by a transaction are cached at the leader. The cache manager guarantees update propagation to the followers and the log's garbage collection.

5.3.2 Cache Manager

The cache manager's design is similar to that in a classical RDBMS where data is stored and fetched from multiple disks; the follower nodes are equivalent to the multiple disks. The group's data can either be piggy-backed with the group creation requests or can be fetched on-demand when accessed by the group. For cache management, the cache manager uses standard data eviction policies [87].

As updates on the group's data accumulate, the cache manager batches the updates and asynchronously ships them to the follower nodes. For every follower, the cache manager maintains a map of the sequence number for the last update that was acknowledged by the follower. Once a follower node acknowledges receipt of the updates, this map is updated and the corresponding log entries at the leader are marked for garbage collection. The frequency of update propagation is configurable and is a trade-off between network I/O and the time taken to recover the leader's state. Since the cache manager guarantees update propagation before group deletion, a group deletion request has to wait until update propagation is complete. In the event of a follower node failure, updates for the data items served by the follower node will be queued at the leader. A group delete request will block in such a scenario. Conceptually, the update propagation logic is a guaranteed ordered delivery publish/subscribe system where the leader is the publisher and the followers are the subscribers.

5.4 Recovering from Node Failures

For node failures, we assume fail-stop behavior and do not consider Byzantine or malicious failures. We use write-ahead logging for node recovery. We further assume that this log is persistent beyond a node's failure, i.e., a node failure does not result in permanent data loss or the log is replicated to tolerate a node's permanent data loss. The Key Grouping protocol is oblivious of the dynamics, such as node failures or partition reassignments, in the underlying statically partitioned data store. The Key Grouping protocol uses identities of the data items instead of identities of the nodes owning the data items. We assume that the underlying partitioned system store maintains a persistent mapping of the data items to the nodes owning them. The leader and follower nodes use this information to communicate.

The Key Grouping protocol appends log entries for every protocol action and forces the entries corresponding to the critical protocol actions. This write-ahead log is the persistent storage for the grouping information and is used to recover the state after a node failure. After a failure, a node must recover from the leader and follower logs and resume disrupted group operations. This includes restoring the state of the groups that were successfully formed, as well as restoring the state of the protocol for the groups whose creation or deletion were in progress when the node crashed. The recovery algorithm is a one-pass algorithm that starts at the oldest log entry and replays the messages in the order in which they were received. Recall that duplicate, delayed, or spurious messages are not logged during normal operation and hence are not replayed during recovery. Once the log replay completes, the leader's (or the follower's) state has been re-created and it can resume normal operation. The leader and the follower logs are logically separate; however, they might be physically stored in a single log file in case a node is hosting both the leader and follower processes.

In addition to the protocol and group states, the unique identifiers, such as the group id and the yield id, are also critical to protocol operation and are recovered after a failure. This information is also logged during normal protocol operation and is recovered. During normal operation, the yield id is monotonically incremented and the highest known yield id is forced to the log before the last $\langle \texttt{JA} \rangle$ message was sent by the follower. Therefore, a failed node can recover the highest yield id and restart with a yield id larger than that obtained from the log. Therefore, after recovery is complete, the entire state of the protocol is restored to the point before the crash.

The leader node must also recover the group's transaction state from the transaction log. Standard log replay algorithms for transaction recovery [69, 87] are used to recover the committed transactions' state and undo the aborted transactions's effects. Log replay recreates that state of the cache before the failure. Data item versions are used at the follower nodes to guarantee idempotence and handle repeated failures resulting in duplicate cache flushes.

The Key Grouping protocol recovers from node and message failures. However, there are various availability tradeoffs in the presence of arbitrary failures. On one hand, failures with groups might result in data unavailability. For example, consider the case when a follower, due to the receipt of a delayed $\langle \texttt{J} \rangle$ request, has yielded control of a key k to a phantom group. Since the group to which k joined is already deleted, k should be available to be accessed independently or to join another group. However, k cannot be accessed since k's status indicates that it is part of a group. k will be available only after the $\langle \texttt{D} \rangle$ message is received from the phantom group's leader. Deletion of phantom groups is, therefore, dependent on the availability of the leader. If the leader has failed or cannot communicate with k's storage node, then k remains unavailable for an extended period of time even though the node on which k resides is available. This scenario is impossible when storage and ownership is coupled.

On the other hand, the Key Group abstraction also allows for extended availability in the presence of certain types of failures. For example, consider the case where key k is part of a group. The leader of the group owns k during the group's lifetime. If the leader has cached k's data, the leader can continue serving k even if the follower node storing k has crashed. If storage and ownership was coupled, failure of the node on which k resides would make k unavailable. In our design, k remains available as long as the leader node is available.

5.5 Correctness Guarantees

The Key Grouping protocol guarantees safety and liveness in the presence of message or node failures. Our failure model assumes that messages can fail, i.e., messages can be lost, reordered, delayed, or duplicated. Node failures are fail-stop and we do not tolerate Byzantine failures or malicious behavior. The Key Grouping protocol can tolerate network partitions but does not guarantee progress in the event of a network partition. We first define the safety and liveness guarantees and then prove that the Key Grouping protocol provides these guarantees.

Definition 5.5.1. ***Safety:*** *At no instant of time should more than one node claim ownership of a data item.*

Definition 5.5.2. ***Liveness:*** *In the absence of repeated failures, i.e., if the leader and the follower nodes can communicate for sufficiently long durations, no data item is indefinitely left without an owner.*

Our definitions of safety and liveness are in terms of ownership of data items. Safety ensures that a data item has at most one owner and liveness ensures that in the absence of failures, a data item has at least one owner. The rationale is that in the underlying statically partitioned system, every data item has exactly one owner. The Key Grouping protocol transfers this data ownership dynamically between nodes, and is therefore correct if it guarantees the same behavior.

The protocol's safety is straightforward in the absence of any failures. The handshake between the leader and the follower guarantees that there is at most one owner of a data item. During group creation, once a follower yields ownership of a data item, the leader becomes the new owner. Similarly, during group deletion, once the leader relinquishes ownership, the follower regains ownership of the data item. Thus, a data item is guaranteed to have at least one owner in the absence of failures.

Theorem 5.5.3. *Safety and liveness is guaranteed in the absence of any failures.*

We now prove safety and liveness in the presence of both message and node failures.

5.5.1 Ensuring Safety

We first prove that message failures cannot jeopardize data safety.

Lemma 5.5.4. *Message failures do not result in multiple concurrent owners of a data item.*

A follower yields ownership of a data item only if it received a $\langle$`J`$\rangle$ message and the data item is not part of a group. Lemma 5.5.4 follows directly from the fact that the above check is independent of message failures.

Lemma 5.5.5. *Node failures do not result in multiple concurrent owners of a data item.*

Proof. Assume for contradiction that a node failure results in multiple concurrent owners of a data item. Concurrent ownership is impossible during normal operation without failures (see Theorem 5.5.3). Therefore, yielding ownership to different leaders must be interleaved by a node failure. Let i be the data item and G_1 and G_2 be the two groups. Further assume that i yielded ownership to G_1 before failure and G_2 after the failure. Therefore, the event corresponding to i joining G_1 was not recovered after F_i failed, thus allowing i to join G_2. However, this contradicts that F_i forced a log entry corresponding to the receipt of G_1's $\langle$`J`$\rangle$ and recovery after F_i's failure will restore i's state as part of G_1. $\square$

Corollary 5.5.6. *It is impossible to have multiple concurrent owners of a data item.*

Corollary 5.5.6 follows directly from Theorem 5.5.3 and Lemmas 5.5.4 and 5.5.5.

5.5.2 Ensuring Liveness

Liveness of data items will be jeopardized if either a data item is part of a phantom group or a group is left without a leader node. In either case, the data items are unavailable to application accesses. We show that both scenarios are impossible if the leader and the followers can communicate for sufficiently long duration.

Lemma 5.5.7. *Creation of phantom groups is impossible.*

Proof. Assume for contradiction that a phantom group might have been formed. This is possible only if the follower receives a spurious delayed $\langle$`J`$\rangle$ message followed by a delayed $\langle$`JAA`$\rangle$ message (similar to that shown in Figure 5.4). The follower will yield to a phantom group when it accepts the $\langle$`JAA`$\rangle$ and *terminates* the group creation phase. However, such a scenario is impossible since the follower, using the yield id, detects that the $\langle$`JAA`$\rangle$ message does not correspond to the current group creation phase and rejects it. $\square$

Lemma 5.5.7 proves that phantom groups cannot be formed. In addition, the Key Grouping protocol must also ensure that a data item that yielded to a spurious $\langle$`J`$\rangle$ message, and hence is without an owner, is freed eventually if the follower node can communicate with the perceived leader node.

Lemma 5.5.8. *A spurious $\langle \texttt{J} \rangle$ message does not indefinitely leave a data item without an owner.*

Proof. **Proof by contradiction.** Assume for contradiction that a data item that yielded control to the spurious $\langle \texttt{J} \rangle$ message is left without an owner, i.e., the follower never received a $\langle \texttt{D} \rangle$ for the phantom group. The follower will retry the $\langle \texttt{JA} \rangle$ message until it receives a $\langle \texttt{JAA} \rangle$ that matches the yield id of the $\langle \texttt{JA} \rangle$ or it receives a $\langle \texttt{D} \rangle$ message. If the follower received a $\langle \texttt{D} \rangle$ message, the data item will be freed and the follower will regain ownership. Therefore, the follower must have received a $\langle \texttt{JAA} \rangle$ matching the yield id of the $\langle \texttt{JA} \rangle$, i.e., the perceived leader replied with a $\langle \texttt{JAA} \rangle$ corresponding to the $\langle \texttt{JA} \rangle$ message. This is, however, impossible since the group is a phantom and hence is non-existent at the leader. Hence the proof. □

Corollary 5.5.9. *Message failures do not leave a data item without an owner for indefinite periods.*

Corollary 5.5.9 follows directly from Lemmas 5.5.7 and 5.5.8. We now prove that liveness is not jeopardized in the presence of a node failure.

Lemma 5.5.10. *In the event of a failure of a group's leader node, the group is not orphaned, i.e., left without a leader node.*

Proof. Assume for contradiction that a group G_i is indefinitely left without a leader after its original leader node L_i failed. If L_i's failure was transient, it will recover from its log after a restart. If L_i's failure was permanent, another node will recover L_i's state from the log. In either case, G_i left without an owner implies that the replay of L_i's log did not contain any entry corresponding to G_i. This is however impossible since the log at the leader was forced on the receipt of the group create request and before the first $\langle \texttt{J} \rangle$ message was sent for G_i. □

Forcing of the leader's log at critical protocol events ensures that these events are persistent beyond the leader node's failures. Log replay during recovery restores the state of active groups and resumes any group creation or deletion that were in progress before the failure.

Corollary 5.5.11. *A data item is not left without an owner for indefinite periods after a failure, i.e., ownership is restored after the failed leader or follower nodes have recovered and can communicate with each other.*

Corollary 5.5.11 follows directly from Corollary 5.5.9 and Lemma 5.5.10.

5.6 Prototype Implementation

In this section, we present the details of a prototype implementation, called G-Store, which uses the Key Group abstraction and the Key Grouping protocol to efficiently support transactional multi-key accesses. We provide two implementation designs of G-Store. In the first design, the grouping protocol is executed as a layer outside the data store, and all accesses to the data store are handled by this layer. We refer to this design as the **client based implementation** since the grouping layer is a client to the Key-value store. This design is suitable for building a pluggable grouping library compatible with multiple underlying Key-value stores or in the case where the Key-value store is part of an infrastructure and cannot be modified. The second design involves extending a Key-value store to implement the Key Grouping protocol as a *middleware layer* within the Key-value store. We refer to this design as the **middleware based design**. The middleware layer handles the group specific operations and the Key-value store handles data storage. This design is suitable where the infrastructure provider is implementing a data store which supports *multi-key* accesses and has access to the code base for the Key-value store.

The operations supported by G-Store are a superset of the operations supported by the underlying Key-value store's operations. In addition to reads/scans, writes, or read/modify/update of a single row of data, G-Store supports multi-step transactions accessing one or more data items in a group. Any combination of read, write, or read/-modify/update operations on keys of a group can be enclosed in a transaction.

In our implementation, we consider Key-value stores, such as Bigtable [24] and HBase [49], which support reads with strong replica consistency, thus allowing the grouping layer to operate on the most up-to-date data. Therefore, the transactions on a group can also guarantee strong consistency on data accesses. Key-value stores such as PNUTS [26] and Dynamo [35] can be configured to support weaker consistency of reads. The interplay of strong consistency guarantees of transactions with weaker consistency guarantees of the data store can result in weaker consistency and isolation guarantees of transactions on a group; a detailed analysis of this interplay is beyond the scope of this work. In the rest of this section, we first describe these two implementation designs, and then describe some implementation issues common to both these designs.

5.6.1 Client based Implementation

This implementation consists of a client layer outside the Key-value store which provides the Key Group semantics using the *single key* accesses supported by the underlying Key-value store. Figure 5.6 provides an illustration of this design. The application clients interact with the grouping layer, which provides a view of a data store support-

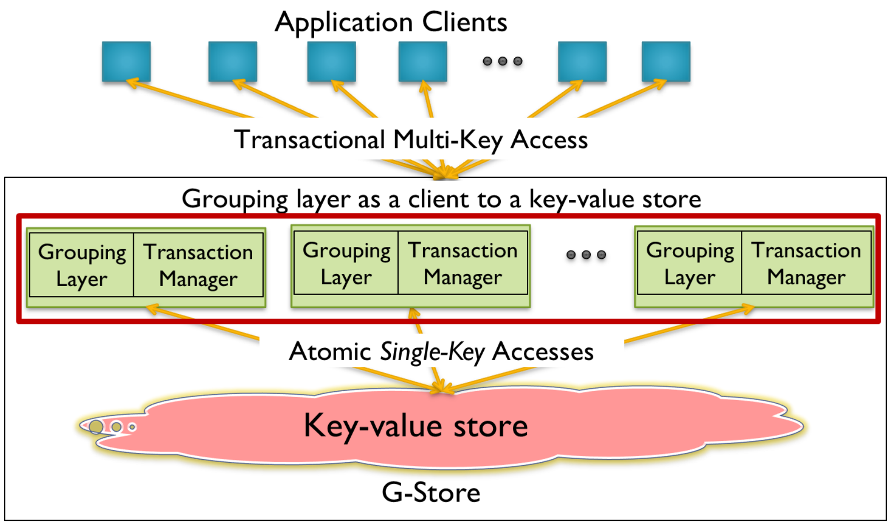


Figure 5.6: A client based implementation of G-Store.

ing transactional access to a Key Group. The grouping layer treats the Key-value store as a black box and interacts with it using the store's client APIs.

The grouping layer executes the Key Grouping protocol to co-locate ownership of the keys at a single node. For scalability, the grouping layer can potentially be distributed across a cluster of nodes as depicted in Figure 5.6. The key space is horizontally partitioned amongst the nodes in the grouping layer; our implementation uses range partitioning, but other partitioning schemes can also be supported. A *group create* request from the application client is forwarded to the grouping layer node responsible for the leader key in the group. That node is designated as the leader node and executes the Key Grouping protocol.

In this design, the followers are the actual keys resident in the Key-value store. The $\langle \texttt{J} \rangle$ request acquires exclusive *locks* on the follower keys. Locking information for each key is stored in a new column introduced to each table. If a key is available, then the lock column is set to a special value `FREE`. If the key is part of a group, the lock column stores the group id of the key's group. The $\langle \texttt{J} \rangle$ request performs a *test and set* operation on the lock column to test availability. If the key is available, then the *test and set* operation atomically obtains the lock and stores the group id. Since the *test and set* operation is on a *single key*, all Key-value stores support this operation. Moreover, if multiple concurrent group create requests content for a key, only one is guaranteed to succeed. The result of the *test and set* operation acts as the $\langle \texttt{JA} \rangle$. Once the $\langle \texttt{J} \rangle$ request has been sent to all the keys in the group, the group creation phase terminates. Very

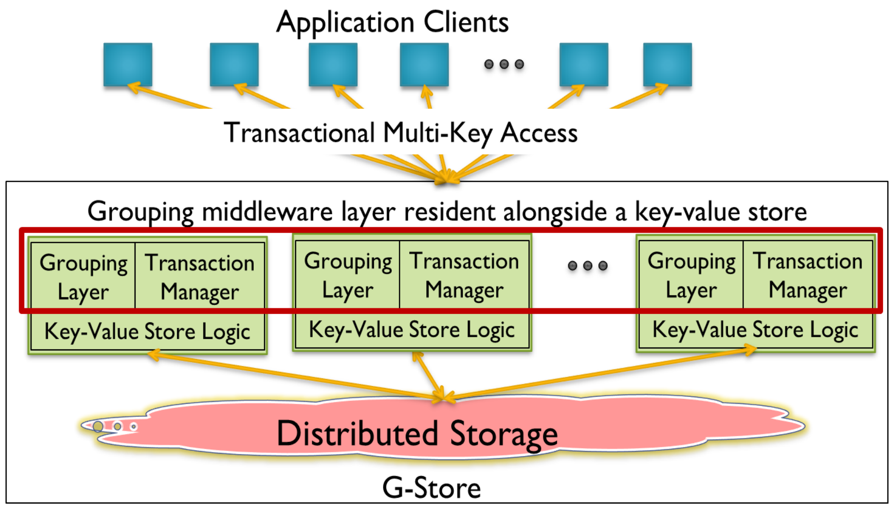


Figure 5.7: A middleware based implementation of G-Store.

little state about the groups is maintained in the grouping layer, so that failures in the grouping layer need minimal recovery.

5.6.2 Middleware based Implementation

We now provide a design of G-Store where the Key Grouping protocol and the group related operations are implemented as a middleware layer conceptually resident within a Key-value store. In this implementation, we use an open-source Key-value store, HBase [49], where the entire key space is horizontally range-partitioned into *regions*. Regions are distributed on a cluster of nodes, called *region servers*, and each region server is responsible for multiple regions. The grouping layer logic is co-located with the region server logic, as shown in Figure 5.7. The grouping layer logic owns all the keys served by that region server. This implementation allows the protocol messages to be batched for all the keys co-located at a follower node. The grouping layer provides the abstraction of Key Groups and executes the Key Grouping protocol, while the *transaction manager* guarantees transactional access to keys in a Key Group.

In this design, the grouping layer maintains the group's state and the Key-value store is not aware of the groups' existence. The grouping layer is responsible for logging the protocol actions and also recovering the state of the groups. On receipt of a group creation request, the leader node sends the $\langle \texttt{J} \rangle$ message to all the follower nodes. The grouping layer logic at each region server keeps track of its share of keys that are part of an existing group. Depending on the state of the key, the follower replies with a $\langle \texttt{JA} \rangle$

message. Since the Key Group abstraction does not mandate all keys to be part of a group, accesses to keys not part of any group are delegated to the Key-value store logic. Our implementation showed that this middleware based design is minimally intrusive to the design of the Key-value store and requires minimal modifications to the code of the Key-value stores.

5.6.3 Transaction Management

Our prototype implements OCC [61] and 2PL [40] as two different concurrency control mechanisms to ensure serializability. In OCC, transactions do not obtain locks when reading or writing data. They rely on the optimistic assumption that there are no conflicts with other concurrently executing transactions. Before a transaction commits, it is validated to guarantee that the optimistic assumption was indeed correct and the transaction did not conflict with any other concurrent transaction. In case a conflict is detected, the transaction is aborted. Writes made by a transaction are kept local and are applied to the database only if the transaction commits. We implemented parallel validation in OCC which results in a very short critical section for validation, thus allowing more concurrency.

The lock based scheduler is pessimistic and acquires appropriate locks before accessing a data item. The two phase locking rule, where a transaction can acquire a lock only if it has not yet released a lock, guarantees serializability. We implemented strict 2PL where the locks are released at transaction completion.

All groups co-located at the same leader node share a common log which is stored in the DFS to allow quick recovery from node failures. The leader, follower, and transaction logs share the same physical log. This sharing minimizes the number of DFS accesses and allows effective batching. Each log entry has the group id to allow the entry to be associated to the corresponding group. Log entries are buffered until a force request is made or until the log buffer fills up. A force on the log guarantees that the log is persistent. The log is forced corresponding to some critical events in the grouping protocol and when a transaction on a group commits. The following optimizations minimize the number of DFS accesses: (i) no log entry is written to record the start of a transaction, (ii) a `COMMIT` entry is appended only for update transactions, (iii) no log entry is made for aborted transactions, the absence of a `COMMIT` record implies an abort, and (iv) group commits are implemented to commit transactions in groups and batch their log writes.

5.6.4 Discussion

Leader key selection. The leader key for a group determines at which node the ownership of all the group members will be co-located. Selecting the leader key, therefore, presents various trade-offs. In order the minimize the protocol cost and the cost of data movement, a greedy strategy might be to determine the node which owns the biggest portion of the group members and then select one of the node's owned keys as the leader. This strategy minimizes the data movement costs. However, this strategy might create hotspots and results and add the cost of leader key selection during group creation. Moreover, due to node failures or re-assignments in the underlying partitioned system during the lifetime of the group, some other node might own more keys that the node selected at the time of group creation. On the other hand, the system can randomly select one key from the group to become the leader. The randomness in selection provides better load balancing but might lead to higher messaging costs compared to the greedy strategy. Our prototype implements the random strategy.

Group id and yield id generation. The unique identifiers, group id and yield id, play an important role in protocol correctness. The group id is a collection of the leader key and a another identifier; the combination of the two identifiers should be unique. Since the leader key is owned by a single node, a unique identifier local to the leader node is sufficient. Similarly, the yield id is monotonically increasing at a given follower node and need not have a global order. Therefore, both identifiers can be generated locally by the leader and the follower processes. An atomic counter, incremented for every new id generated, is sufficient to generate the identifiers.

Non-overlapping groups. The Key Grouping protocol does not allow the Key Groups to overlap. As a result, if there are concurrent requests for a key to join different Key Groups, only one of them will succeed. This strategy simplifies the protocol implementation and also ensures that transactions on a Key Group executes at a single node. Overlapping Key Groups can, however, be handled if the overlaps are few. The overlapping Key Groups are co-located at the same leader node so that the same transaction manager can execute the transactions that span the Key Groups due to the overlap. This co-location of overlapping Key Groups allows non-distributed transaction execution. However, the co-location strategy cannot handle scenarios of large numbers of overlapping Key Groups.

Non blocking group creation and deletion. When a follower node receives a $\langle \texttt{J} \rangle$ request for a key that is currently part of another group, the request can be blocked until the key becomes available. Such blocking is, however, not desirable for multiple practical considerations. First, concurrent group create requests with overlapping keys might result in a deadlock due to the blocking behavior. Second, as observed in the case of transaction locking, as data contention increases, i.e., more groups overlap, the blocking behavior will result in higher group creation latency. On the other hand, during group deletion, the leader must first propagate all the updates to the followers. The delete request might therefore be blocked in the event of failures. Our implementation takes a non-blocking strategy for both the scenarios. During group create, if a follower key is unavailable, the follower node sends a negative acknowledgement as part of the $\langle \texttt{JA} \rangle$ message. Depending on the configuration, the leader might either retry the request or create the group without the follower. When the application requests a group delete, the request is logged at the leader node and the application is acknowledged without blocking for the group to be deleted. The leader then asynchronously deletes the group.

5.7 Experimental Evaluation

We now experimentally evaluate the performance of the Key Grouping protocol and G-Store on a cluster of machines in the Amazon Web Services Elastic Compute Cloud (EC2) infrastructure. Both the Key Grouping protocol as well as G-Store have been implemented in Java. We use HBase version 0.20.2 [49] as the Key-value store in both the client based and middleware based designs. We evaluate the performance of the system using an application benchmark which resembles the type of workloads for which the system has been designed. We first describe the benchmark used for the experiments and the experimental set up used for evaluating the system, and then provide results from the experiments.

5.7.1 An Application Benchmark

The application benchmark is based on an online multi-player game. The game's participants have a profile represented as a key-value pair with a user's unique id used as the key. All information about the players is contained in a single *Players* table. Note that even though the benchmark uses only a single table, G-Store is designed to support operations over multiple tables; the keys are composed from the table name and the row key. In this benchmark, the number of concurrent groups and the average number of keys in a group are used are used as scaling factors for the *Players* table's size. If there are $\mathcal{G}$ concurrent groups with an average of $\mathcal{K}$ keys per group, then the *Players* table must contain at least $n \times \mathcal{G} \times \mathcal{K}$ keys, where n is the scaling factor used to determine the number of keys that are not part of any group.

The application clients submit group create requests where the number of keys in the group are chosen from a normal distribution with $\mathcal{K}$ as the mean and $\sigma_{\mathcal{K}}$ as the stan-

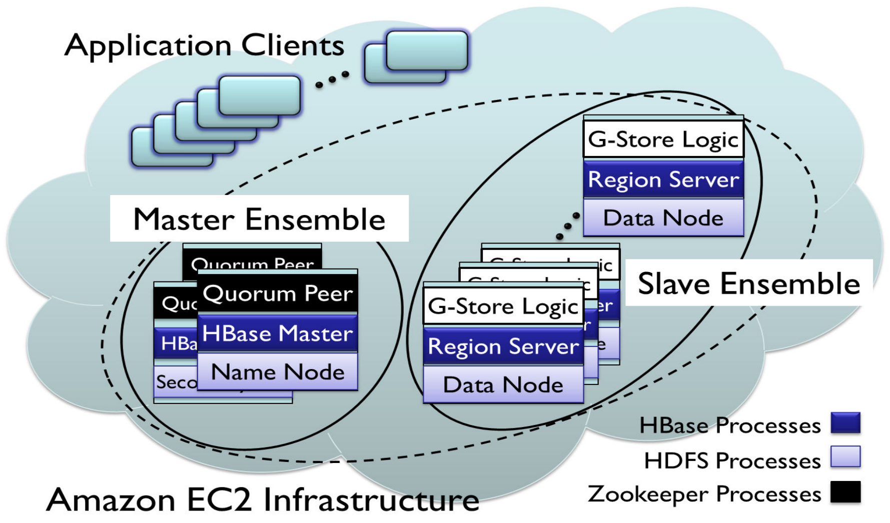


Figure 5.8: Cluster setup for experiments to evaluate G-Store's performance.

dard deviation. Keys in a group are either chosen from a uniform random distribution (called **random**) or as a contiguous sequence of keys where the first key in the sequence is chosen from a uniform random distribution (called **contiguous**). Once group creation succeeds, the client issues operations on the group. The number of operations on a group is chosen from a normal distribution with mean $\mathcal{N}$ and standard deviation $\sigma_{\mathcal{N}}$. Each client submits an operation on a group, and once the operation is completed, the client waits for a random time between δ and 2δ before submitting the next operation. On completion of all operations, the group is deleted, and the client starts with a new group create request. A client can have concurrent *independent* requests to multiple groups, and a group can also have multiple clients. Since G-Store provides isolation against concurrent operations on a group, clients need not be aware of the existence of concurrent clients.

The parameters in the benchmark—the number of concurrent groups $\mathcal{G}$, the average number of keys $\mathcal{K}$, the scaling factor n, the average number of operations on a group $\mathcal{N}$, and the time interval between operations δ—are used to evaluate various aspects of the system's performance. The first three parameters control the scale of the system, while $\mathcal{N}$ controls the amortization effect that the group operations have on the group creation and deletion overhead. The parameter δ allows the clients to control the load on the grouping layer.

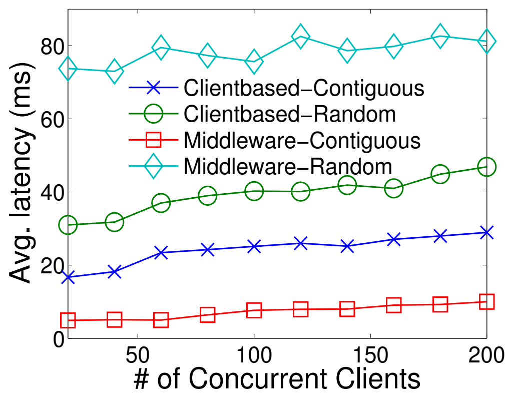
5.7.2 Experimental Setup

Experiments were performed on a cluster of commodity machines in Amazon EC2. All machines in the cluster were "High CPU Extra Large Instances" (`c1.xlarge`) with 7 GB of memory, 20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each), 1690 GB of local instance storage, and a 64-bit platform. Figure 5.8 provides an illustration of the experimental cluster setup. HBase uses HDFS [50] as a fault tolerant distributed file system, and we use HDFS version $0.20.1$ for our experiments. HBase also uses Zookeeper [54] for distributed lease management; we used Zookeeper version $3.2.2$. An ensemble of servers is used to replicate Zookeeper's state; to tolerate n failures, the ensemble must have $(2n + 1)$ nodes. We used a 3 node Zookeeper ensemble referred to as the *master ensemble*. The nodes in the master ensemble also host the HBase Master, the HDFS Namenode, and the HDFS Secondary Namenode. The HBase Master and HDFS Namenode perform metadata operations and hence are lightly loaded. A separate ensemble of nodes, referred to as the *slave ensemble*, is used to host the slave processes of HBase and HDFS, i.e., the HBase region servers and the HDFS Data nodes. The slave ensemble also hosts the G-Store logic for the Key Grouping protocol transaction manager, and cache manager implementations. Our experiments used a slave ensemble of 10 nodes which amounts to about 70 GB of main memory, 16.15 TB of disk space, and 80 virtual CPU cores. Together the master and slave ensembles constitute the data management infrastructure. Application clients were executed on a different set of nodes within EC2.

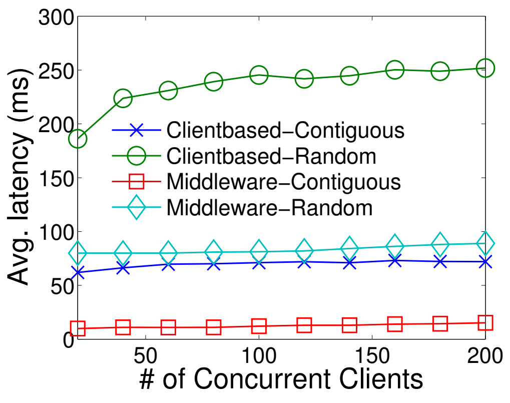
The *Players* table has 25 column families, loosely similar to columns in RDBMSs, which store various profile-related information such as Name, Age, Sex, Rating, and Account Balance. Each column family contains a single column. The table contains about 1 billion randomly generated rows and each column family in a row contains about 20–40 bytes of data. The data size on disk is about a terabyte without compression and with $3\times$ replication. After the data insertion, the HBase cluster is allowed to repartition and load balance the data before any group-related operations are performed, thus allowing more even distribution of the workload across all nodes. During the rest of the experiment, data in the table are updated, but no bulk insertions or deletions are performed. We use *best effort* group creation which succeeds when a response is received from all the followers, irrespective of whether they joined.

5.7.3 Group Creation

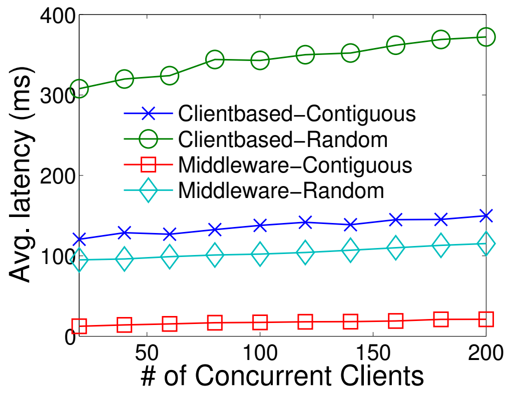
In this section, we evaluate the group creation latency, i.e., the time taken to create a group, and the group creation throughput, i.e., the number of groups concurrently created. The goal of these experiments is to evaluate the overhead of group creates and the system's scalability in dealing with thousands of concurrent group create requests.



(a) 10 keys per group.



(b) 50 keys per group.

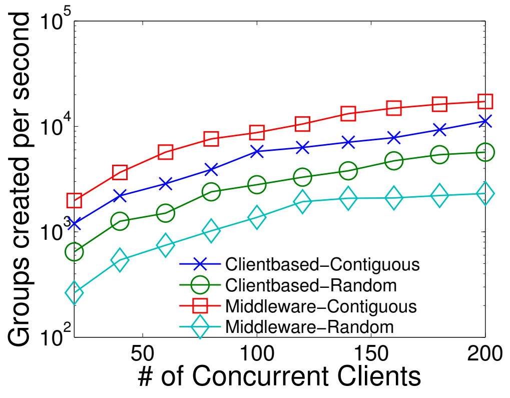


(c) 100 keys per group.

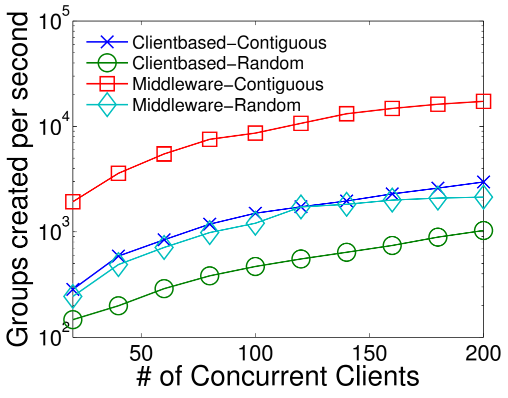
Figure 5.9: The average time taken to create a group.

The group creation-only workload is simulated by setting the number of operations on the group ($\mathcal{N}$) to 0 so that the application client creates the group, and then deletes the group once group creation succeeds. We evaluate both the variants of G-Store, the client based design and the middleware based design, and the two different key selection distributions, random and contiguous. The contiguous selection emulates Megastore's *entity group* abstraction [11] which mandates that keys in an entity group are from a contiguous key space. The random selection demonstrates the flexibility of the Key Group abstraction in allowing arbitrarily selected keys.

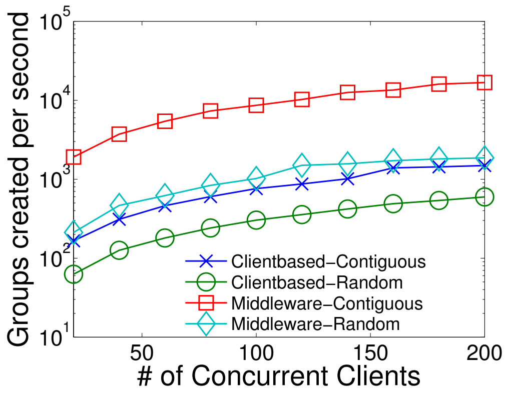
Figure 5.9 plots the latency of group-create requests as the number of concurrent clients submitting requests increases. The group creation latency is measured as the time elapsed between the client issuing the group create request and obtaining the group creation notification from the grouping layer. The latency (in ms) is plotted along the y-axis, and the number of concurrent clients is plotted along the x-axis. The different



(a) 10 keys per group.



(b) 50 keys per group.



(c) 100 keys per group.

Figure 5.10: Number of concurrent group create requests served.

lines in the graphs correspond to different G-Store implementations and the different key distribution. The different plots correspond to different sizes of the groups (varying parameter $\mathcal{K}$).

As expected, the middleware based design outperforms the client based design as the group size increases. Additionally, when the keys in a Key Group are from a contiguous space, the performance of the Key Grouping protocol is considerably improved. The performance benefits of the middleware based design can be attributed to the benefits of batching the protocol messages. When the keys are contiguous, group creation typically results in a single $\langle \texttt{J} \rangle$ message. By contrast, in the client based design, irrespective of whether the keys are contiguous or not, multiple messages are needed, thus resulting in higher latency. This is because the Key-value store only supports single key *test and set* operations which are used to implement the “lock” or followers *yielding* control to the leader.

Another interesting observation is that the middleware based design is not very sensitive to the number of keys in a group. This is again due to batching of the requests resulting in fewer messages. If there are n nodes in the cluster, there can be at most n messages irrespective of the group's size. On the other hand, the latency of the client based design is dependent on the number of keys in the group. Therefore, for most practical applications, the middleware based implementation will result in better performance.

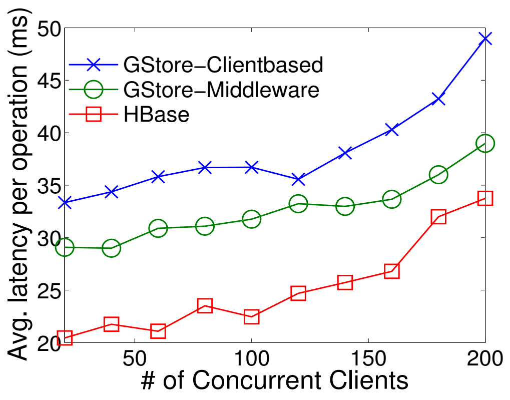
Figure 5.10 further asserts the superior performance of the middleware based design. We plot the throughput of group-create requests served per second, aggregated over the entire cluster. Along y-axis is the number of group-create requests processed per second (drawn in a logarithmic scale), and along the x-axis is the number of concurrent clients. Figure 5.10 also demonstrates the almost linear scalability in terms of the group create requests being served; as the number of clients increase from 20 to 200, an almost 10 times increase in group creation throughput is observed. Furthermore, a 10 node G-Store cluster is able to processes tens of thousands of concurrent groups.

5.7.4 Operations on a Group

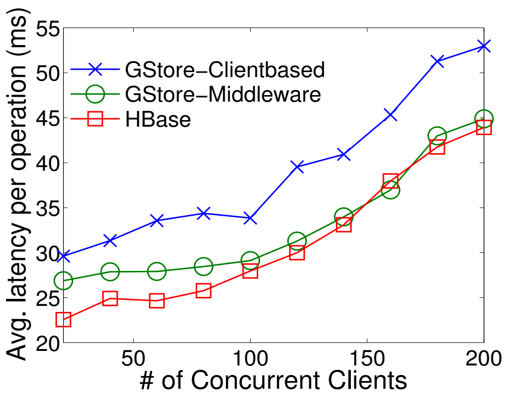
In this experiment, we evaluate the performance of G-Store for a workload with operations after group creation. Once a group has been formed, transaction execution is similar in the client based and the middleware based designs. In these experiments, we compare the performance of G-Store to that of HBase in terms of the average operation latency. For G-Store, the reported average latency includes the group creation and deletion times, in addition to the time taken by the operations. Since HBase does not have any notion of groups, no transactional guarantees are provided on accesses made to the group members. HBase performance numbers provide a baseline to measure the overhead of providing transactional guarantees using the Key Group abstraction compared to accesses without any guarantees. For HBase, the updates to a group of keys are applied as a batch of updates to the rows which are part of the group without any transactional guarantees. We evaluate the system's performance for different values of $\mathcal{N}$, i.e., the number of operations on a group. We set the number of keys in a group ($\mathcal{K}$) to 50, and the sleep interval δ between consecutive operations to 10 ms.

Figure 5.11 plots the average latency of each operation, as the number of operations on a group are varied. The x-axis plots the number of concurrent clients which is varied from 20 to 200. The y-axis plots the average latency per operation (in ms). The average operation latency in G-Store includes the group creation latency and is computed as:

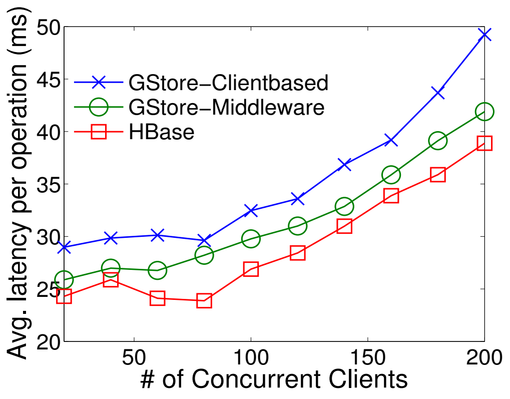
$$\frac{\text{Group Create Latency} + \text{Latency of operations}}{\text{Number of operations}}$$



(a) 10 operations per group created.



(b) 50 operations per group created.



(c) 100 operations per group created.

Figure 5.11: Average latency per operation on a group. The HBase workload simulates grouping operations by issuing operations only limited to a group of keys. No groups are formed in HBase and no guarantees are provided for the operations on the group.

For HBase, the latency is the average over all operations. As is evident from Figure 5.11, the grouping layer introduces very little overhead (10–30%) compared to that of HBase while providing ACID guarantees on Key Group accesses which HBase does not provide. The overhead is considerably reduced as the number of operations per group increases. This reduction in overhead is primarily due to the benefits of batching of the updates and local transaction execution. The asynchronous propagation of updates from the grouping layer to the data store also results in low transaction latencies. Therefore, for a very low overhead, G-Store provides transactional accesses to a Key Group using a Key-value store as a substrate.

5.8 Extensions to the Key Group Abstraction

Our discussion till now assumes that all the data items join a Key Group when it is created and all of them leave when the group is deleted. The Key Grouping protocol, however, can handle data items joining and leaving a Key Group at any point during the group's lifetime. The Key Group abstractions can therefore be generalized. A Key Group is a set of data items on which an application seeks transactional access. This set can be dynamic over the lifetime of a group, thus allowing data items to join or leave the group while the group is active. Transactional guarantees are provided only to the data items that are part of the group when the transaction is executing. As earlier, new groups can be formed and groups can be deleted at any time. Key Groups continue to remain disjoint, i.e., no two concurrent Key Groups will have the same data item.

Conceptually, the Key Grouping protocol handles the joining and deletion of a group's data items individually; these requests are batched to improve performance. Therefore, the Key Grouping protocol remains unchanged to support this generalized Key Group abstraction. When the application requests a data item k to join an already existing group, the leader executes the creation phase of the Key Grouping protocol only for k joining the group. When k leaves a group, the leader ensures that k is not being accessed an active transaction and all of k's updates have propagated to the follower node. The leader then executes the deletion phase only for k leaving the group.

Since the Key Grouping protocol remains unchanged, correctness of the Key Grouping protocol is straightforward. Atomicity and durability of transactions is similarly guaranteed using logging and recovery as earlier. Transaction serializability within a group is guaranteed by the concurrency control technique used. However, as data items move from one group to another, conflict dependencies are carried over. We now show that the generalized Key Group abstraction can guarantee transaction serializability. Intuitively, the leader has to release the logical "lock" on k when it leaves the group and the deletion executes as a *transaction* serialized after all transactions that accessed k.

Our discussion uses 2PL which guarantees serializability by locking data items accessed by a transaction. However, the proofs can be extended to other concurrency control techniques that guarantee serializability. Let $T_1, T_2, \ldots$ denote the transactions and $G_1, G_2, \ldots$ denote the groups. Let $l_{T_1}(k)$ and $ul_{T_1}(k)$ denote the lock and unlock actions on a data item k by transaction T_1. Further, due to the two phase rule, any lock acquired by a transaction must precede a lock released by the transaction, i.e., $l_{T_1}(i) < ul_{T_1}(j)$ for all data items i and j accessed by T_1.

Lemma 5.8.1. *It is impossible to have a cyclic conflict dependency of the form $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_1$ for transactions executing on two groups G_1 and G_2.*

Proof. Assume for contradiction that a cyclic conflict dependency is possible. Without loss of generality, assume that T_1 executed on G_1. Further, let us assume that T_1 and T_2

conflict on data item x. Therefore, $ul_{T_1}(x) < l_{T_2}(x)$. Similarly, if T_2 and T_3 conflict on y, then $ul_{T_2}(y) < l_{T_3}(y)$. Due to the two phase rule, we know, $l_{T_2}(x) < ul_{T_2}(y)$, and hence we have, $ul_{T_1}(x) < l_{T_3}(y)$. Using similar arguments, the conflict dependency $T_1 \rightarrow \cdots \rightarrow T_i$ implies that $ul_{T_1}(x) < l_{T_i}(z)$.

Let T_{i+1} be the first transaction executing on G_2 in the sequence $T_1 \rightarrow \cdots \rightarrow T_i \rightarrow T_{i+1}$. Let T_i and T_{i+1} conflict on data item k that moved from G_1 to G_2. Since k can leave G_1 only if it is not locked by an active transaction executing on G_1, therefore, $ul_{T_i}(k) < l_{T_{i+1}}(k)$, i.e., $ul_{T_1}(x) < l_{T_{i+1}}(k)$. The dependency $T_{i+1} \rightarrow \cdots \rightarrow T_1$ and arguments similar to above implies that $ul_{T_{i+1}}(k') < l_{T_1}(k'')$, where k'' is the data item that moved from G_2 to G_1. Since, $l_{T_{i+1}}(k) < ul_{T_{i+1}}(k')$, we have $ul_{T_1}(x) < l_{T_1}(k'')$, which means that T_1 violated the two phase rule, a contradiction. $\square$

Corollary 5.8.2. *It is impossible to have a cyclic conflict dependency of the form $T_1 \rightarrow T_2 \rightarrow \cdots T_1$ for transactions executing on groups $G_1, G_2, \ldots, G_n$.*

Corollary 5.8.2 follows Lemma 5.8.1 by induction on the number of groups. Therefore, transaction serializability is guaranteed.

5.9 Summary

In this chapter, we presented the Key Group abstraction that allows the applications to dynamically specify the groups of data items on which it wants transactional access; a Key Group is a dynamically defined granule of transactional access. We also presented the Key Grouping protocol that allows the system to co-locate the ownership of a group's data items at a single node, thus allowing efficient transaction execution. We further showed how the Key Grouping protocol can handle message and node failures while guaranteeing safe ownership transfer during group creation and group deletion. We demonstrated the benefits of the Key Group abstraction and the Key Grouping protocol by building a prototype implementation, G-Store, that provided transactional multi-key access on top of a Key-value store that only supports single-key based accesses. Our evaluation showed the low overhead of Key Grouping protocol in co-locating ownership of the data items in a Key Group. We also present a more dynamic variant of the Key Group abstraction allowing data items to join and leave a group during its lifetime.

Our current implementation requires the applications to specify the data items that form a Key Group. In the future, we would like to explore the possibility of mining the application's access patterns to automatically identify the data items that will form a Key Group. Another future direction would be to explore the feasibility of supporting a wider class of applications the involve overlapping Key Groups or require queries span multiple Key Groups.

Part II

Lightweight Elasticity

Chapter 6

Live Migration for Database Elasticity

"To him whose elastic and vigorous thought keeps pace with the sun, the day is a perpetual morning."

– Henry David Thoreau.

Elasticity and pay-per-use pricing are the key features of cloud computing that obviate the need for static resource provisioning for peak loads and allows on-demand resource provisioning based on the workload. However, as noted in Chapter 1, the database tier is not as elastic as the other tiers of the web-application stack. In order to effectively leverage the underlying elastic infrastructure, the database tier must also be elastic, i.e., when the load increases, add more servers to the database tier and migrate some database partitions to the newly allocated servers to distribute the load, and vice versa. Elastic scaling and load balancing therefore calls for lightweight techniques to migrate database partitions in a live system.

In this dissertation, we formulate the problem of live database migration for elastic scaling and load balancing in DBMSs and present the first two approaches to live database migration. In this Chapter, we define some cost metrics to evaluate live migration techniques, analyze some existing approaches to live database migration and formally define the problem.

6.1 Migration Cost Metrics

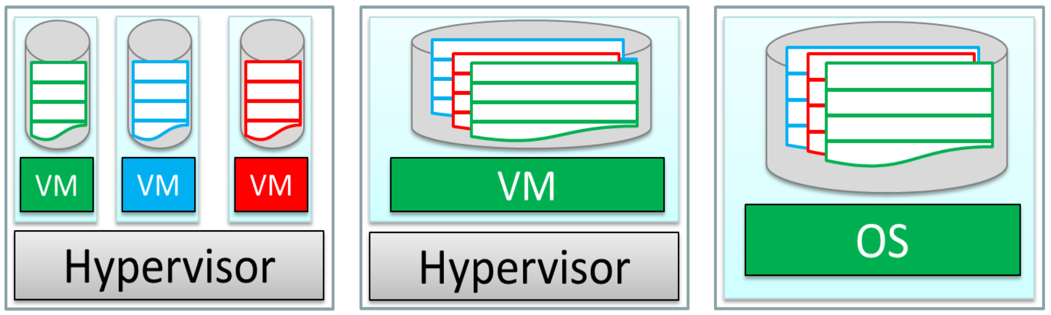
Migrating database partitions in a live system is a hard problem and comes at a cost. We discuss four cost metrics to evaluate the effectiveness of a live database migration technique, both from the user and the system perspective.

- **Service unavailability:** The duration of time for which a database partition is unavailable during migration. Unavailability is defined as the period of time when all requests to the database fail.
- **Number of failed requests:** The number of well-formed requests that fail due to migration. Failed requests include both aborted transactions and failed operations; a transaction consists of one or more operations. Aborted transactions signify failed interactions with the system while failed operations signify the amount of work wasted as a result of an aborted transaction. The failed operations account for transaction complexity; when a transaction with more operations aborts, more work is wasted for the tenant. We therefore quantify both types of failures in this cost metric.
- **Impact on response time:** The change in transaction latency (or response time) observed as a result of migration. This metric factors any overhead introduced in order to facilitate migration as well as the impact observed during and after migration.
- **Data transfer overhead:** Any additional data transferred during migration. Database migration involves the transfer of data corresponding to the partition being migrated. This metric captures the messaging overhead during migration as well any data transferred in addition to the minimum required to migrate the partition.

The first three cost metrics measure the external impact on the application users of the system and their SLAs while the last metric measures the internal performance impact. In a cloud data platform, a provider's service quality is measured through SLAs and satisfying them is foremost for customer satisfaction. A long unavailability window or a large number of failed requests resulting from migration might violate the availability SLA, thus resulting in a penalty. For instance, in Google AppEngine, if the availability drops below 99.95%, then tenants receive a service credit [8]. Similarly, low transaction latency is critical for good tenant performance and to guarantee the latency SLAs. For example, a response time higher than a threshold can incur a penalty in some service models. A live migration technique must have minimal impact on tenant SLAs to be effective in elastic scaling.

6.2 Problem Formulation

Various approaches to database migration are possible. One straightforward approach is to leverage VM migration [17, 25, 67] which is a standard feature in current VMs. The shared hardware multitenancy model, as shown in Figure 6.1(a), is one possible approach. In this approach, every database partition has its independent database process and VM and a Hypervisor (or a virtual machine monitor) orchestrates resource



(a) One partition per VM. (b) Multiple partitions per VM. (c) Shared process multitenancy.

Figure 6.1: Live database migration for lightweight elasticity.

sharing between VMs (and hence partitions) co-located at the same node. VM migration techniques can be used to migrate individual VMs. Therefore, any partition can be migrated between two nodes, thus enabling fine-grained load balancing. However, this approach results in high performance impact during normal operation. This overhead arises from heavy contention for the I/O resources due to uncoordinated and competing accesses by the independent database servers. This is in addition to the overhead introduced due to duplication of some components, such as the OS and database process, as well as due to the use of virtualized resources. A recent experimental study shows that compared to the shared process model, this design requires $2\times$ to $3\times$ more machines to serve the same number of database partitions, and for a given assignment results in $6\times$ to $12\times$ less performance [28].

An alternative design to mitigate this performance problem is shown in Figure 6.1(b). Multiple partitions are consolidated within a single database process that is executing within a VM. This design only suffers from the overhead introduced due to virtualization and can still leverage VM migration. However, since multiple partitions are co-located within a single VM, all of them must now be migrated, thus losing the ability of fine grained load balancing as supported in the shared hardware model.

To minimize the performance impact due to resource sharing between partitions it is imperative that multiple partitions share the same database process. This design is called shared process multitenancy (shown in Figure 6.1(c)). Furthermore, to enable lightweight fine-grained elastic load balancing, the system must possess the ability to migrate *individual* partitions on-demand in a live system. We characterize this as virtualization embedded into the database tier allowing the individual database partitions to be virtualized from the node serving the partition. This decoupling of the partitions from the nodes serving the partitions allows the system to effectively react to changes in load patterns by dynamically orchestrating resources, i.e., allocating and de-allocating

Notation	Description
$\mathbb{P}_M$	The tenant database or partition being migrated
$\mathbb{N}_S$	Source node for $\mathbb{P}_M$
$\mathbb{N}_D$	Destination node for $\mathbb{P}_M$
T_S, T_D	Transaction executing at nodes $\mathbb{N}_S$ and $\mathbb{N}_D$ respectively
P_k	Database page k

Table 6.1: Notational conventions for live database migration.

resources on-demand. These technologies are critical to effective capacity planning, to ensure high resource utilization, and to optimize the system's operating cost.

Most traditional DBMSs do not support live migration primarily because enterprise infrastructures are typically statically provisioned for peak expected load, and elasticity was not considered an important feature in database systems. A partition can still be migrated in a traditional DBMS without leveraging VM migration. The source DBMS node ($\mathbb{N}_S$) stops serving the partition ($\mathbb{P}_M$), aborts all transactions active on $\mathbb{P}_M$, flushing all the changes made at $\mathbb{N}_S$, copies the partition's data to the destination DBMS node ($\mathbb{N}_D$), and restarts serving $\mathbb{P}_M$ at $\mathbb{N}_D$. We call this approach **stop and copy**. This approach has high migration cost: $\mathbb{P}_M$ becomes unavailable during migration and all transactions active at the start of migration are aborted. Furthermore, the entire database cache is lost when the $\mathbb{P}_M$ is restarted at $\mathbb{N}_D$, thereby incurring a high post migration overhead for warming up the database cache. Therefore, this approach has a high impact on the tenant's SLA, thus preventing it from being effectively used for elastic scaling.

In this dissertation, we propose two different techniques for live migration within the database tier using the shared process multitenancy model. Our techniques focus on OLTP systems running short read-write transactions. The first technique, **Albatross**, allows for live database migration in decoupled storage architectures where the partition's persistent data is stored in a network-addressable storage abstraction. In such an architecture, the persistent data is not migrated. Albatross focusses on migrating the database cache and the state of active transactions to minimize the performance impact when $\mathbb{P}_M$ restarts at $\mathbb{N}_D$. The second technique, **Zephyr**, allows for live database migration in a shared nothing architecture where the partition's persistent data is stored on disks locally attached to the DBMS nodes and hence must also be migrated. Therefore, Zephyr focusses on migrating this persistent data with no downtime. Chapter 7 presents the details of Albatross while Chapter 8 presents the details of Zephyr. Table 6.1 summarizes the notational conventions used.

Chapter 7

Decoupled Storage Architectures

"Man could not stay there forever. He was bound to spread to new regions, partly because of his innate migratory tendency and partly because of Nature's stern urgency."

– Huntington Ellsworth.

In this chapter, we present the detailed design and implementation of Albatross [34], a live migration technique for decoupled storage architectures.[1] In a decoupled storage architecture, the persistent data is stored in network-addressable storage (**NAS**) servers—an architecture used by ElasTraS and G-Store presented earlier in the dissertation. Albatross assumes the shared process multitenancy model and migrates a database partition ($\mathbb{P}_M$) from the source DBMS node ($\mathbb{N}_S$) to the destination DBMS node ($\mathbb{N}_D$). For simplicity of exposition, we assume that a partition is a self contained database for a small application tenant; we use the terms tenant and partition interchangeably.[2]

Albatross leverages the semantics of database systems to migrate the database cache and the state of transactions active during migration. In decoupled storage architectures, the persistent data of a database partition is stored in the NAS and therefore does not need migration. Migrating the database cache allows the partition being migrated to start "warm" at the destination, thus minimizing the impact on transaction latency. To minimize the unavailability window, this copying of the state is performed iteratively while the source continues to serve transactions on the partition being migrated. Copying the state of transactions active during migration allows them to resume execution

[1]The name Albatross is symbolic of the lightweight nature and efficiency of the technique that is typically attributed to Albatrosses.

[2]The work reported in this chapter was published as the paper entitled "Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration" in the proceedings of the Very Large Data Bases endowment (PVLDB), Vol 4, No. 8, May 2011.

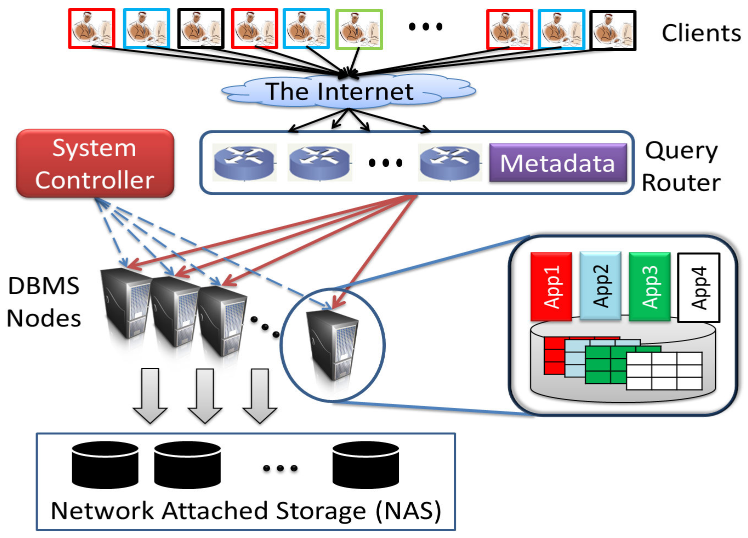


Figure 7.1: A reference de-coupled storage multitenant DBMS architecture.

at the destination. Therefore, Albatross results in negligible impact from the tenants' perspective, allowing the system to effectively use migration while guaranteeing that (i) the tenants' SLAs are not violated, (ii) transaction execution is serializable, and (iii) migration is safe in spite of failures. Moreover, in Albatross, the destination node performs most of the work of copying the state, thus effectively relieving load on the overloaded source node.

7.1 Reference System Architecture

We consider a decoupled storage database architecture for OLTP systems executing short running transactions (see Figure 7.1). Our reference system model uses the shared process multitenancy model where a partition is entirely contained in a single database process which co-locates multiple partitions. Application clients connect through a **query router** which abstracts physical database connections as logical connections between a tenant and its partition. Even though Figure 7.1 depicts the query router as a single logical unit, a deployment will have a distributed query router to scale to a large number of connections. The mapping of a partition to its server is stored as *system metadata* which is cached by the router.

A cluster of DBMS nodes serves the partitions; each node has its own local transaction manager (TM) and data manager (DM). A TM consists of a *concurrency control* component for transaction execution and a *recovery component* to deal with failures.

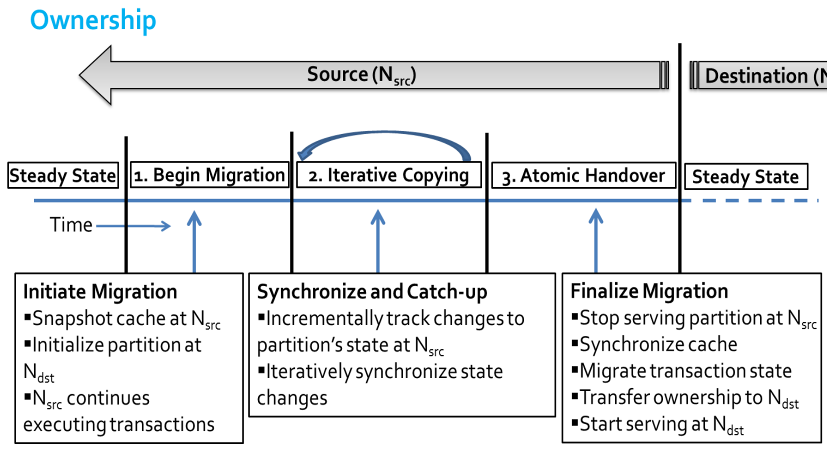

Ownership
Source (N_{src})
Destination (N_{dst})
Steady State
1. Begin Migration
2. Iterative Copying
3. Atomic Handover
Steady State
Time
Initiate Migration
▪Snapshot cache at N_{src}
▪Initialize partition at N_{dst}
▪N_{src} continues executing transactions
Synchronize and Catch-up
▪Incrementally track changes to partition's state at N_{src}
▪Iteratively synchronize state changes
Finalize Migration
▪Stop serving partition at N_{src}
▪Synchronize cache
▪Migrate transaction state
▪Transfer ownership to N_{dst}
▪Start serving at N_{dst}

Figure 7.2: Migration timeline for Albatross (times not drawn to scale).

A partition is served by a single DBMS node, called its owner. This unique ownership allows transactions to execute efficiently without distributed synchronization amongst multiple DBMS nodes.

Network-addressable storage provides a scalable, highly available, and fault-tolerant storage of the persistent image of the tenant databases. This *decoupling* of storage from ownership obviates the need to copy a tenant's data during migration. This architecture is different from shared disk systems which use the disk for arbitration amongst concurrent transaction [15]. A system controller performs control operations including determining the partition to migrate, the destination, and the time to initiate migration.

7.2 The Albatross Technique

7.2.1 Design Overview

Albatross aims to have minimal impact on tenant SLAs while leveraging the semantics of the database structures for efficient database migration. This is achieved by iteratively transferring the database cache and the state of active transactions. For a two phase locking (2PL) based scheduler [40], the transaction state consists of the lock table; for an Optimistic Concurrency Control (OCC) [61] scheduler, this state consists of the read-sets and write-sets of active transactions and a subset of committed transactions. Figure 7.2 depicts the timeline of Albatross when migrating $\mathbb{P}_M$ from $\mathbb{N}_S$ to $\mathbb{N}_D$.

Phase 1: *Begin Migration:* Migration is initiated by creating a snapshot of the database cache at $\mathbb{N}_S$. This snapshot is then copied to $\mathbb{N}_D$. $\mathbb{N}_S$ continues processing transactions while this copying is in progress.

Phase 2: *Iterative Copying:* Since $\mathbb{N}_S$ continues serving transactions for $\mathbb{P}_M$ while $\mathbb{N}_D$ is initialized with the snapshot, the cached state of $\mathbb{P}_M$ at $\mathbb{N}_D$ will lag that of $\mathbb{N}_S$. In this iterative phase, at every iteration, $\mathbb{N}_D$ tries to "catch-up" and synchronize the state of $\mathbb{P}_M$ at $\mathbb{N}_S$ and $\mathbb{N}_D$. $\mathbb{N}_S$ tracks changes made to the database cache between two consecutive iterations. In iteration i, Albatross copies to $\mathbb{N}_D$ the changes made to $\mathbb{P}_M$'s cache since the snapshot of iteration $i - 1$. This phase terminates when approximately the same amount of state is transferred in consecutive iterations or a configurable maximum number of iterations have completed.

Phase 3: *Atomic Handover:* This phase transfers the ownership of $\mathbb{P}_M$ is transferred from $\mathbb{N}_S$ to $\mathbb{N}_D$. $\mathbb{N}_S$ stops serving $\mathbb{P}_M$, copies the final un-synchronized database state and the state of active transactions to $\mathbb{N}_D$, flushes changes from *committed* transactions to the persistent storage, transfers control of $\mathbb{P}_M$ to $\mathbb{N}_D$, and notifies the *query router* of the new location of $\mathbb{P}_M$. To ensure safety in the presence of failures, this operation is guaranteed to be *atomic*. The successful completion of this phase makes $\mathbb{N}_D$ the owner of $\mathbb{P}_M$ and completes the migration.

The iterative phase minimizes the amount of $\mathbb{P}_M$'s state to be copied and flushed in the handover phase, thus minimizing the unavailability window. In the case where the transaction logic executes at the client, transactions are seamlessly transferred from $\mathbb{N}_S$ to $\mathbb{N}_D$ without any loss of work. The handover phase copies the state of active transaction along with the database cache. For a 2PL scheduler, it copies the lock table state and reassigns the appropriate locks and latches at $\mathbb{N}_D$; for an OCC scheduler, it copies the read/write sets of the active transactions and that of a subset of committed transactions whose state is needed to validate new transactions. For a 2PL scheduler, updates of active transactions are done in place in the database cache and hence are copied over during the final copy phase; in OCC, the local writes of the active transactions are copied to $\mathbb{N}_D$ along with the transaction state. For transactions executed as stored procedures, $\mathbb{N}_S$ tracks the invocation parameters of transactions active during migration. Any such transactions active at the start of the handover phase are aborted at $\mathbb{N}_S$ and are automatically restarted at $\mathbb{N}_D$. This allows migrating these transactions without moving the process state at $\mathbb{N}_S$. Durability of transactions that committed at $\mathbb{N}_S$ is ensured by synchronizing the commit logs of the two nodes.

Iterative copying in Albatross is reminiscent of migration techniques used in other domains, such as in VM migration [25] and process migration [82]. The major difference is that Albatross leverages the semantics of the database internals to copy only the information needed to re-create the state of the partition being migrated.

7.2.2 Failure Handling

In the event of a failure, data safety is primary while progress towards successful completion of migration is secondary. Our failure model assumes node failures and a connection oriented network protocol, such as TCP, that guarantees message delivery in order within a connection. We do not consider Byzantine failures or malicious node behavior. We further assume that node failures do not lead to complete loss of the persistent data: either the node recovers or the data is recovered from the NAS where data persists beyond DBMS node failures. If either $\mathbb{N}_S$ or $\mathbb{N}_D$ fails prior to Phase 3, migration of $\mathbb{P}_M$ is aborted. Progress made in migration is not logged until Phase 3. If $\mathbb{N}_S$ fails during Phases 1 or 2, its state is recovered, but since there is no persistent information of migration in the commit log of $\mathbb{N}_S$, the progress made in $\mathbb{P}_M$'s migration is lost during this recovery. $\mathbb{N}_D$ eventually detects this failure and in turn aborts this migration. If $\mathbb{N}_D$ fails, migration is again aborted since $\mathbb{N}_D$ does not have any log entries for a migration in progress. Thus, in case of failure of either node, migration is aborted and the recovery of a node does not require coordination with any other node in the system.

The atomic handover phase (Phase 3) consists of the following major steps: (*i*) flushing changes from all committed transactions at $\mathbb{N}_S$; (*ii*) synchronizing the remaining state of $\mathbb{P}_M$ between $\mathbb{N}_S$ and $\mathbb{N}_D$; (*iii*) transferring ownership of $\mathbb{P}_M$ from $\mathbb{N}_S$ to $\mathbb{N}_D$; and (*iv*) notifying the query router that all future transactions must be routed to $\mathbb{N}_D$. Steps (iii) and (iv) can only be performed after the Steps (i) and (ii) complete. Ownership transfer involves three participants—$\mathbb{N}_S$, $\mathbb{N}_D$, and the query router—and must be atomic. We perform this handover as a **transfer transaction** and a Two Phase Commit (2PC) protocol [44] with $\mathbb{N}_S$ as the coordinator guarantees atomicity in the presence of node failures. In the first phase, $\mathbb{N}_S$ executes steps (i) and (ii) in parallel, and solicits a vote from the participants. Once all the nodes acknowledge the operations and vote *yes*, the transfer transaction enters the second phase where $\mathbb{N}_S$ relinquishes control of $\mathbb{P}_M$ and transfers it to $\mathbb{N}_D$. If one of the participants votes *no*, this *transfer transaction* is aborted and $\mathbb{N}_S$ remains the owner of $\mathbb{P}_M$. This second step completes the transfer transaction at $\mathbb{N}_S$ which, after logging the outcome, notifies the participants about the decision. If the handover was successful, $\mathbb{N}_D$ assumes ownership of $\mathbb{P}_M$ once it receives the notification from $\mathbb{N}_S$. Every protocol action is logged in the commit log of the respective nodes.

7.3 Correctness guarantees

Migration correctness or *safety* implies that during normal operation or in case of a failure during migration, the system's state or application data is not corrupted or lost.

Definition 7.3.1. ***Safe Migration.*** *A migration technique is safe if the following conditions are met:* (*i*) Data Safety and Unique ownership: *The persistent data of a partition is transactionally consistent and at most one DBMS node* owns *a partition at any instant of time; and* (*ii*) Durability: *Updates from committed transactions are durable.*

We argue migration safety using a series of guarantees provided by Albatross and reason about how these guarantees are met.

Guarantee 7.3.2. ***Atomicity of handover.*** *$\mathbb{P}_M$ is owned by exactly one of $\mathbb{N}_S$ or $\mathbb{N}_D$.*

This is trivially satisfied when no failures occur. We now describe how logging and recovery ensures atomicity during failures.
At most one owner: A failure in the first phase of the atomic handover protocol is handled similar to a failure during Phases 1 and 2—both $\mathbb{N}_S$ and $\mathbb{N}_D$ recover normally and abort $\mathbb{P}_M$'s migration and $\mathbb{N}_S$ remains the owner. Failure in this phase does not need coordinated recovery. After receiving responses (both yes or no votes), $\mathbb{N}_S$ is ready to complete the *transfer transaction* and enters the second phase of atomic handover. Once the decision about the outcome is forced into $\mathbb{N}_S$'s log, the transfer transaction enters the second phase. Forcing the log record constitutes the atomic event. A failure in this phase requires coordinated recovery. If $\mathbb{N}_S$ forced the log record to commit, $\mathbb{N}_D$ is the new owner of $\mathbb{P}_M$, otherwise $\mathbb{N}_S$ continues as the owner. If $\mathbb{N}_S$ failed before notifying $\mathbb{N}_D$, $\mathbb{N}_D$ must wait until the state of $\mathbb{N}_S$ is recovered from its log before $\mathbb{N}_D$ starts serving $\mathbb{P}_M$. Therefore, the atomic handover protocol guarantees that there is at most one owner of $\mathbb{P}_M$.
At least one owner: A pathological condition arises when after committing the transfer transaction at $\mathbb{N}_S$, both $\mathbb{N}_S$ and $\mathbb{N}_D$ fail. Atomic handover guarantees that in such a scenario, both $\mathbb{N}_S$ and $\mathbb{N}_D$ do not relinquish ownership of $\mathbb{P}_M$. If the handover was complete before $\mathbb{N}_S$ failed, when $\mathbb{N}_S$ recovers, it transfers ownership to $\mathbb{N}_D$. Otherwise $\mathbb{N}_S$ continues as the owner of $\mathbb{P}_M$. The synchronized recovery of $\mathbb{N}_S$ and $\mathbb{N}_D$ guarantees at least one owner.

Guarantee 7.3.3. *Changes made by aborted transactions are neither persistently stored nor copied over during migration.*

This follows from the invariant that in the steady state, the combination of the database cache and the persistent disk image does not have changes from aborted transactions. In OCC, changes from uncommitted transactions are never publicly visible. In locking based schedulers, the cache or the persistent data might have changes from uncommitted transactions. Such changes are undone if a transaction aborts. Any such changes copied over during the iterative phases are guaranteed to be undone during the first round of the atomic handover phase.

Guarantee 7.3.4. *Changes made by committed transactions are persistent and never lost during migration.*

Cache flush during the handover phase ensures that writes from transactions that have committed at $\mathbb{N}_S$, are persistent. The log entries of such committed transactions on $\mathbb{P}_M$ are discarded at $\mathbb{N}_S$ after successful migration.

Guarantee 7.3.5. *Migrating active transactions does not violate the durability condition even if the commit log at $\mathbb{N}_S$ is discarded after successful migration.*

This is ensured since Albatross copies the commit log entries for transactions active during migration to $\mathbb{N}_D$, which are then forced to $\mathbb{N}_D$'s commit log when these transactions commit.

Guarantee 7.3.2 ensures *data safety* and Guarantees 7.3.3, 7.3.4, and 7.3.5 together ensure *durability*, thus guaranteeing the safety of Albatross. Therefore, in the presence of a failure of either $\mathbb{N}_S$ or $\mathbb{N}_D$, the migration process is aborted without jeopardizing the safety.

Guarantee 7.3.6. ***Serializability.*** *Copying the transaction state in the final handover phase of Albatross ensures serializable transaction execution after migration.*

OCC guarantees serializability by validating transactions against conflicts with other concurrent and committed transactions. The handover phase copies the state of active transactions and that of a subset of transactions that committed after the earliest of the active transactions started. Therefore, all such active transactions can be validated at $\mathbb{N}_D$ and checked for conflicts.

For a 2PL Scheduler, the two phase locking rule ensures serializability of a locking based scheduler. The final handover phase copies the state of the lock table such that active transactions have the locks that were granted to them at $\mathbb{N}_S$ when they resume execution at $\mathbb{N}_D$. Therefore, a transaction continues to acquire locks using the two phase rule at $\mathbb{N}_D$, thus ensuring serializability.

We now articulate two important properties of Albatross that allow the system to gracefully tolerate failures and characterize its behavior in the presence of failures during migration.

Property 7.3.7. ***Independent Recovery.*** *Except during the execution of the atomic handover protocol, recovery from a failure of $\mathbb{N}_S$ or $\mathbb{N}_D$ can be performed independently.*

At any point of time before atomic handover, $\mathbb{N}_S$ is the owner of $\mathbb{P}_M$. If $\mathbb{N}_S$ fails, it recovers without interacting with $\mathbb{N}_D$ and continues to be the owner of $\mathbb{P}_M$. Similarly, if $\mathbb{N}_D$ fails, it recovers its state. Unless the handover phase was initiated (Phase 3 of the protocol), $\mathbb{N}_D$ has no log record about the migration in progress, so it "forgets" the migration and continues normal operation. Similarly, once handover has been successfully completed, $\mathbb{N}_D$ becomes the new owner of $\mathbb{P}_M$. A failure of $\mathbb{N}_S$ at this instant can be recovered independently as $\mathbb{N}_S$ does not need to recover the state of $\mathbb{P}_M$. Similarly, a failure of $\mathbb{N}_D$ requires recovery of only its state; $\mathbb{N}_D$ can independently recover the state of $\mathbb{P}_M$ since it had successfully acquired the ownership of $\mathbb{P}_M$.

Property 7.3.8. ***A single failure does not incur additional unavailability.*** *Any unavailability of $\mathbb{P}_M$ resulting from the failure of one of $\mathbb{N}_S$ or $\mathbb{N}_D$ during migration is equivalent to unavailability due to a failure during normal operation.*

From an external observer's perspective, $\mathbb{N}_S$ is the owner of $\mathbb{P}_M$ until the atomic handover phase (Phase 3) has successfully completed. Any failure of $\mathbb{N}_D$ before Phase 3 does not affect the availability of $\mathbb{P}_M$. A failure of $\mathbb{N}_S$ during this phase makes $\mathbb{N}_S$ unavailable, which is equivalent to a failure of $\mathbb{N}_S$ under normal operation where $\mathbb{P}_M$ would also become unavailable. Similarly, after migration is complete, $\mathbb{N}_D$ becomes the owner of $\mathbb{P}_M$. Any failure of $\mathbb{N}_S$ does not affect $\mathbb{P}_M$, and a failure of $\mathbb{N}_D$ which makes $\mathbb{P}_M$ unavailable is equivalent to the failure of $\mathbb{N}_D$ during normal operation. The only complexity arises in the case of a failure in Phase 3 when a coordinated recovery is needed. If $\mathbb{N}_S$ fails before successful completion of Phase 3, even if $\mathbb{N}_S$ had locally relinquished ownership of $\mathbb{P}_M$, if the transfer transaction did not complete, $\mathbb{N}_D$ cannot start serving $\mathbb{P}_M$ in which case it becomes unavailable. This is similar to the blocking behavior in 2PC [44]. However, since the handover transaction did not complete, from an observer's perspective, $\mathbb{N}_S$ was still the owner of $\mathbb{P}_M$, and hence this unavailability is equivalent to the failure of $\mathbb{N}_S$ during normal operation. Thus, it is evident, single site failures during migration does not impact availability of $\mathbb{P}_M$.

The ability to safely abort migration at an incomplete state and the single owner philosophy allow independent recovery of the failed node's state even after a failure during migration. This is crucial for effective use of migration for elasticity without unnecessarily making tenants unavailable when a node fails. Furthermore, one of the implications of Property 7.3.8 is that in spite of using a 2PC protocol, the handover phase does not block any system resources as a result of a failure, limiting the impact of failure to only the partitions being served by the failed node. This is contrary to the case where a coordinator failure in 2PC causes other transactions conflicting with blocked transactions to also block.

During normal operation, the progress of Albatross is guaranteed by the maximum bound on the number of iterations that forces a handover. Since Albatross does not log the progress of migration, the state synchronized at $\mathbb{N}_D$ is not persistent. The rationale is that since Albatross copies the main memory state of $\mathbb{P}_M$, which is lost after a failure, little gain can be achieved by logging the progress at either node. Progress towards migration is therefore not guaranteed in case of repeated failures.

7.4 Implementation Details

We implemented Albatross in ElasTraS, a scale-out decoupled storage OLTP DBMS presented earlier in Chapter 4. ElasTraS's architecture is similar to the abstract system model depicted in Figure 7.1. The *DBMS Nodes* are equivalent to *Owning Transaction Managers* (OTM) which *own* a number of partitions and provide transactional guarantees on them using optimistic concurrency control (OCC). The *NAS* is the *distributed fault tolerant storage* (DFS) which stores the persistent data and the transaction logs. The *controller* is equivalent to the TM Master which is responsible for system management and initiating migration. Its role in migration is only limited to notifying the source OTM ($\mathbb{N}_S$) and the destination OTM ($\mathbb{N}_D$) to initiate migration. Transaction routing is handled by the client library that is linked to every application client; the router transparently migrates the client connections after migration without any changes to the application code. The client library uses a collection of metadata tables that store the mapping of a partition to the OTM which is currently serving the partition. The combination of the client library and the metadata tables constitute the query router.

ElasTraS uses an append-only storage layer (the Hadoop distributed file system). Updates to a tenant's data is periodically flushed to create new files on the DFS. Data is physically stored as a collection of immutable segments which store the data sorted by the keys. A segment is a collection of blocks with an index to map blocks to key ranges. An OTM caches the contents of blocks that were read by the application. Updates are maintained as a separate main memory buffer which is periodically flushed to the DFS as new segments.

Creating a database snapshot. In the first step of migration, $\mathbb{N}_S$ creates a snapshot of the tenant's database cache. Albatross does not require a transactionally consistent snapshot of the cache. $\mathbb{N}_S$'s cache snapshot is a list of identifiers for the immutable segment blocks that are cached. This list of block identifiers is obtained by scanning the read cache using a read lock. It is passed to $\mathbb{N}_D$ which then reads the blocks directly from the DFS and populates its cache. This results in minimal work at $\mathbb{N}_S$ and delegates all the work of warming up the cache to $\mathbb{N}_D$. The rationale is that during migration, $\mathbb{N}_D$ is expected to have less load than $\mathbb{N}_S$. Therefore, transactions at $\mathbb{N}_S$ observe minimal impact during snapshot creation and copying. After $\mathbb{N}_D$ has loaded all the blocks into its cache, it notifies $\mathbb{N}_S$ of the amount of data transferred (Δ_0); both nodes now enter the next iteration. No transaction state is copied in this phase.

Iterative copying phase. In every iteration, changes made to the read cache at $\mathbb{N}_S$ are copied to $\mathbb{N}_D$. After a first snapshot is created, the data manager of $\mathbb{P}_M$ at $\mathbb{N}_S$ tracks changes to the read cache (both insertions and evictions) and incrementally maintains the list of identifiers for the blocks that were evicted from or loaded in to the read cache since the previous iteration, which is copied to $\mathbb{N}_D$ in subsequent iterations. Again, only the block identifiers are passed; $\mathbb{N}_D$ populates its cache using the identifiers and notifies $\mathbb{N}_S$ of the amount of data transferred (Δ_i). This iterative phase continues until the amount of data transferred in successive iterations is approximately the same, i.e., $\Delta_i \approx \Delta_{i-1}$. The rationale behind this termination condition is that when $\Delta_i \approx \Delta_{i-1}$, irrespective of the magnitude of Δ_i, little gain is expected from subsequent iterations. Δ_i is small for most cases, except when the working set of the database does not fit into the cache, thus resulting in frequent changes to the cache due to blocks being evicted and new blocks added to the cache. A maximum bound on the number of iterations ensures termination when Δ_i fluctuates between iterations.

The write cache is periodically flushed during the iterative copying phase when its size exceeds a specified threshold. A write-cache flush creates a new block whose identifier is passed to $\mathbb{N}_D$ which loads the new block into its read cache. After the handover, $\mathbb{N}_D$ starts serving $\mathbb{P}_M$ with an empty write cache, but the combination of the read and write cache contains the same state of data as in $\mathbb{N}_S$. Since the data manager hides this cache separation, transactions on $\mathbb{P}_M$ continue execution at $\mathbb{N}_D$ unaware of the migration.

Copying the transaction state. ElasTraS uses OCC [61] for concurrency control. In OCC, the transaction state consists of the read and write sets of the active transactions and a subset of committed transactions needed to validate new transactions. The read-/write sets of active transactions and committed transactions are maintained in separate main-memory structures. Two counters are used to assign transaction numbers and commit sequence numbers. In Albatross, the transaction state is copied only in the final handover phase. Writes of an active transaction, stored locally with the transaction's state, are also copied to $\mathbb{N}_D$ during handover along with the counters maintained by the transaction manager. The state of a subset of committed transactions (ones that committed after any one of the current set of active transactions started) is copied to $\mathbb{N}_D$ to validate the active transactions at $\mathbb{N}_D$. The small size of transaction states allows efficient serialization. After handover, $\mathbb{N}_D$ has the exact same transaction state of $\mathbb{P}_M$ as $\mathbb{N}_S$, thus allowing it to continue executing the transactions that were active at the start of the handover phase.

Handover phase. The handover phase flushes changes from committed transactions. After the transaction state and the final changes to the read cache have been copied, the atomic handover protocol makes $\mathbb{N}_D$ the unique owner of $\mathbb{P}_M$ and updates the mapping in the metadata used by the query router. The query router (ElasTraS clients) caches the metadata. After handover, $\mathbb{N}_S$ rejects any request to $\mathbb{P}_M$ which invalidates the system metadata cached at the clients; the clients subsequently read the updated metadata.

The metadata tables in ElasTraS are served by one of the live OTMs. The OTM serving the metadata tables participates in the *transfer transaction* of the atomic handover phase. The TM Master can be a participant of the transfer transaction so that it is aware of the outcome of migration; however, it is not needed for correctness. In our implementation, the TM Master is notified by $\mathbb{N}_S$ after handover completes. Clients that have open connections with $\mathbb{P}_M$ at $\mathbb{N}_S$ are notified directly about the new address of $\mathbb{N}_D$. This prevents an additional network round-trip to read the updated metadata mappings.

For a transaction accessing $\mathbb{P}_M$ during the atomic handover phase, the ElasTraS client library transparently retries the operation; once the handover completes, this retried operation is routed to $\mathbb{N}_D$. Since an OTM's commit log is stored in the DFS, it is not migrated. $\mathbb{P}_M$'s transaction log at $\mathbb{N}_S$ is garbage collected once the transactions active at the start of the handover phase have completed at $\mathbb{N}_D$, though the entries for transactions that committed at $\mathbb{N}_S$ can be purged after the handover completes.

7.5 Experimental Evaluation

We now evaluate our prototype implementation of Albatross using a variety of workloads. We measure migration cost using four cost measures: tenant unavailability window, number of failed requests (aborted transactions or failed operations), impact on transaction latency (or response time), and additional data transfer during migration. We compare performance with the stop and copy technique presented in Chapter 6; we call the technique **stop and migrate (S&M)** since when using a decoupled storage, data is not copied to the destination; S&M only migrates the ownership of $\mathbb{P}_M$ after all the updates at $\mathbb{N}_S$ have been flushed. In S&M, flushing the cached updates from committed transactions results in a long unavailability window. An optimization, called **flush and migrate (F&M)**, performs a flush while continuing to serve transactions, followed by the final stop and migrate step. Both S&M and F&M were implemented in ElasTraS along with Albatross.

7.5.1 Experimental Setup

Experiments were performed on a six node cluster, each with 4 GB memory, a quad core processor, and a 200 GB disk. The distributed fault-tolerant storage and the OTMs are co-located in the cluster of five worker nodes. The TM master (controller) and the clients generating the workloads were executed on a separate node. Each OTM was serving 10 partitions on average. When an operation fails due to a partition being unavailable (due to migration or otherwise), the ElasTraS client library transparently retries these operations until the tenant becomes available again and completes the request. We set the maximum number of iterations in Albatross to 10; Albatross converged within $3 - 7$ iterations in our experiments.

Parameter	Default value
Transaction size	10 operations
Read/Write distribution	80% reads, 20% writes
Partition size	1 GB
Transaction load	50 transactions per second (TPS)
Cache size	250 MB
Access distribution	Hotspot distribution
Default hotspot distribution	80% operations accessing 20% of the database

Table 7.1: Default values for YCSB parameters used in Albatross's evaluation.

In all experiments, except the one presented in Section 7.5.5, we evaluate migration cost when both $\mathbb{N}_S$ and $\mathbb{N}_D$ were lightly loaded so that the actual overhead of migration can be measured. The load on a node is measured using the amount of resources (for instance CPU cycles, disk I/O bandwidth, or network bandwidth) being utilized at the node. When resource utilization is less than 25%, it is referred to as lightly loaded, utilization between $25-70\%$ is referred to as moderately loaded, and utilization above 70% is called overloaded. For simplicity, we only consider CPU utilization.

7.5.2 Methodology

We evaluate migration cost using the modified Yahoo! cloud serving benchmark (YCSB) and the TPC-C benchmark presented in Chapter 4 (see Section 4.5.2). In our experiments, we consider tenant applications with small databases and every tenant is assigned a partition in ElasTraS. We therefore use the terms tenant and partition interchangeably.

For the experiments using YCSB, we vary different YCSB parameters to cover a wide spectrum of workloads. These parameters include the percentage of read operations in a transaction, number of operations in a transaction, size of a partition, load offered on a partition, cache size, and the distribution from which the keys accessed are selected. For experiments using the Zipfian distribution, the co-efficient is set to 1.0. In a specific experiment, we vary one of these parameters while using the default values for the rest of the parameters; the default values of these parameters are provided in Table 7.1. In every experiment, we execute about $12,000$ transactions (about 240 seconds at 50 TPS) to warm up the cache, after which migration is initiated. Clients continue to issue transactions while migration is in progress. We only report the latency for committed transactions; latency measurements from aborted transactions are ignored.

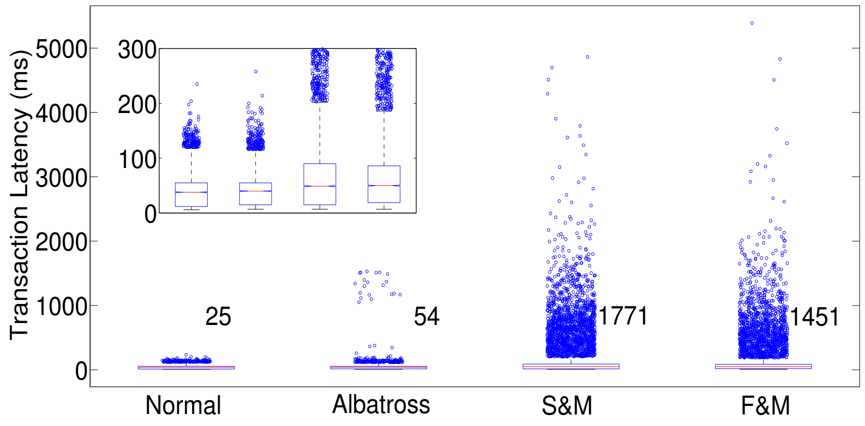


Figure 7.3: Transaction latency distribution (box and whisker plot). Inset shows the same graph but with a limited value of the y-axis to show the box corresponding to the 25^{th} and 75^{th} percentiles.

Each instance of YCSB corresponds to a single tenant served by one of the live OTMs. Multiple YCSB instances emulate a multitenant workload.

For the experiments using TPC-C, each tenant database size is about 1 GB and contains 4 TPC-C warehouses. The cache per tenant is set to 500 MB. We vary the load on each tenant from 500 tpmC (transactions per minute TPC-C) to 2500 tpmC. Again, multiple TPC-C benchmark instances simulate a multitenant workload.

7.5.3 Evaluation using Yahoo! Cloud Serving Benchmark

Impact on response times and throughput

In the first experiment, we analyze the impact of migration on transaction latency using the default workload parameters described above. We ran a workload of $10,000$ transactions, after warming up the cache with another workload of $10,000$ transactions; Figure 7.3 plots the distribution of latency (or response time) of each individual transaction as a box and whisker plot. The four series correspond to the observed transaction latency of an experiment when migration was not initiated (Normal) and that observed when migration was initiated using each of the three different techniques. The inset shows the same plot, but with a restricted range of the y-axis. The box in each series encloses the 25^{th} and 75^{th} percentile of the distribution with the median shown as a horizontal line within each box. The whiskers (the dashed line extending beyond the box) extend to the most extreme data points not considered outliers, and outliers are

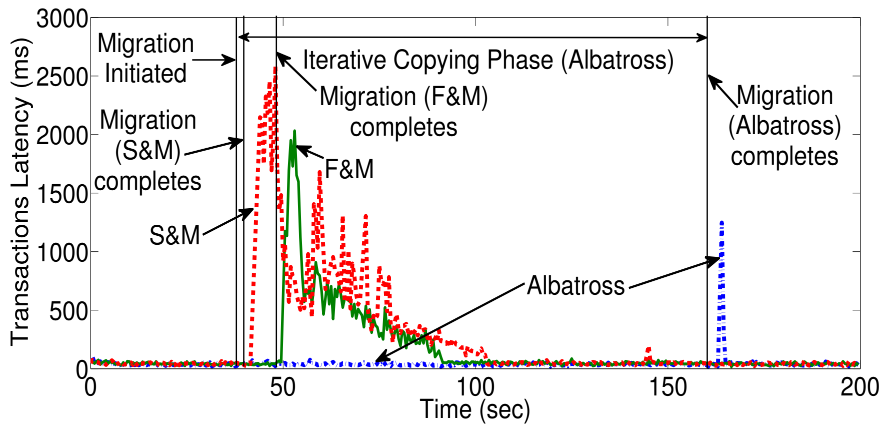


Figure 7.4: Transaction latency observed for different migration techniques. There are 3 different series and each correspond to an execution using one of the discussed migration techniques. All series are aligned at the time at which migration was initiated (about 38 seconds).

plotted individually as circles (in blue).[3] The number beside each series denotes the number of outlier data points that lie beyond the whiskers.

As is evident from Figure 7.3, when migrating a tenant using Albatross, the transaction latencies are almost similar to that in the experiment without migration. A cluster of data points with latency about $1000 - 1500$ ms correspond to transactions that were active during the handover phase which were stalled during the handover and resumed at $\mathbb{N}_D$. On the other hand, both S&M and F&M result in a high impact on transaction latency with about 1500 or more transactions having a latency higher than that observed during normal operation. The high impact on latency for S&M and F&M is due to cache misses at $\mathbb{N}_D$ and contention for the NAS. Since all transactions active at the start of migration are aborted in F&M and S&M, they do not contribute to the increase in latency.

The low impact of Albatross on transaction latency is further strengthened by the experiment reported in Figure 7.4 which plots the average latency observed by the tenants as time progresses; latencies were averaged in disjoint 500 ms windows. The different series correspond to the different migration techniques and are aligned based on the migration start time (about 38 seconds). Different techniques complete migration at different time instances as is shown by the vertical lines; S&M completes at about 40 seconds, F&M completes at around 45 seconds, while Albatross completes

[3]The whiskers denote the sampled minimum and sampled maximum (`http://en.wikipedia.org/wiki/Sample_minimum`).

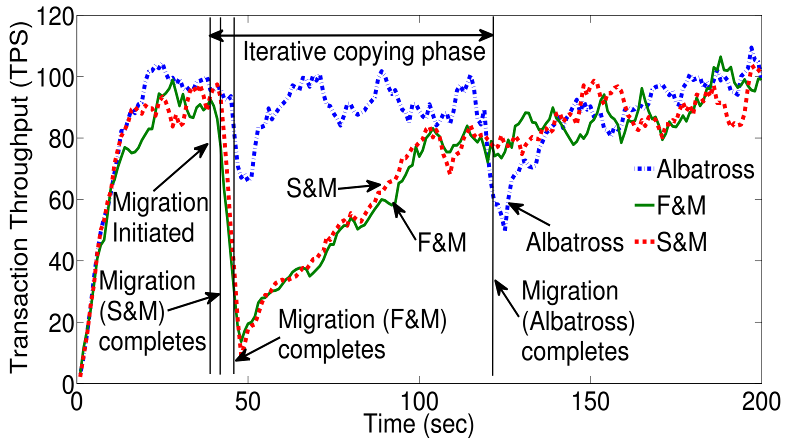
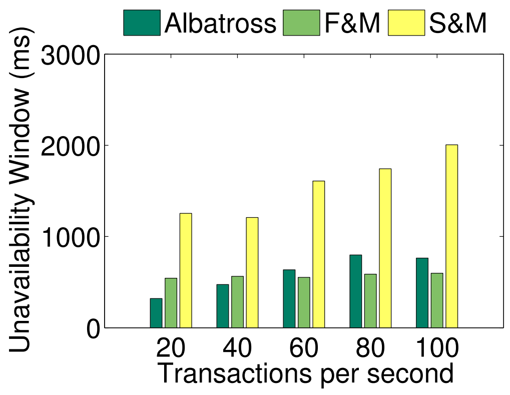


Figure 7.5: Impact of migration on transaction throughput.

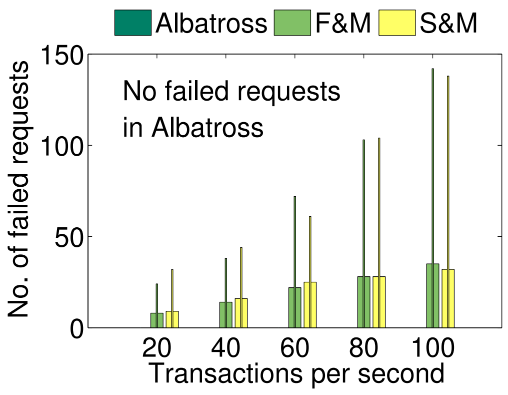
at around 160 seconds. The iterative phase for Albatross is also marked in the figure. As is evident, both F&M and S&M result in an increase in latency immediately after migration completes, with the latency gradually decreasing as the cache at $\mathbb{N}_D$ warms up. On the other hand, even though Albatross takes longer to finish, it has negligible impact on latency while migration is in progress. This is because in Albatross, most of the heavy lifting for copying the state is done by $\mathbb{N}_D$, thus having minimal impact on the transactions executing at $\mathbb{N}_S$. A small spike in latency is observed for Albatross immediately after migration completes which corresponds to active transactions being stalled temporarily during the final handover phase.

The low impact on latency ensures that there is also a low impact on transaction throughput. Figure 7.5 plots the impact of migration on throughput as time progresses (plotted along the x-axis). The y-axis plots the throughput measured for a second long window. The load is generated by four clients threads which issue transactions immediately after the previous transaction completes. The different series correspond to different migration techniques. As is evident from the figure, both S&M and F&M result in a high impact on the client throughput due to increased transaction latency after migration, coupled with throughput reduction during the unavailability window. On the other hand, Albatross results in minor throughput fluctuations, once during the first snapshot creation phase and once during the unavailability window in the handover phase; Albatross results in negligible impact during migration since the list of block identifiers in the cache snapshot is maintained incrementally and $\mathbb{N}_D$ performs most of the work done during the synchronization phase.

For all the techniques, an impact on transaction latency (and hence throughput) is observed only in a time window immediately after migration completes. Hence, for



(a) Partition unavailability.



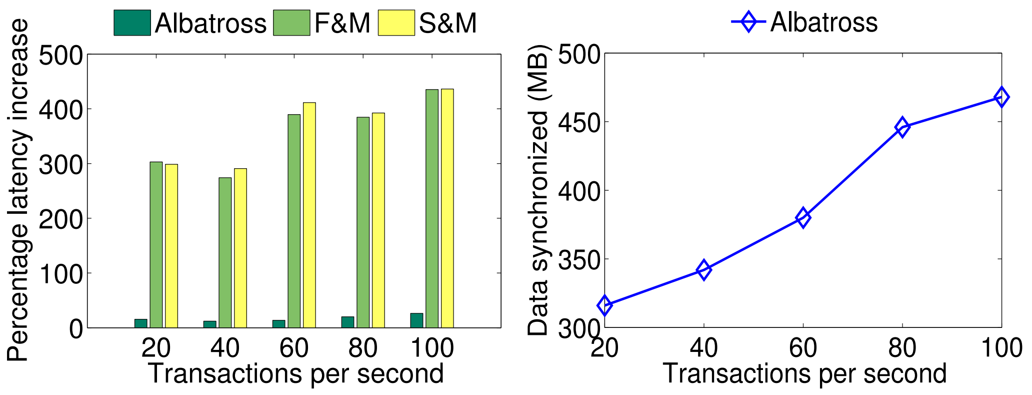
(b) Failed requests.

Figure 7.6: Evaluating the impact of transaction load on partition unavailability and number of failed requests. For failed requests (7.6(b)), the wider bars represent aborted transactions and narrower bars represent failed operations.

brevity in reporting the impact on latency, we report the percentage increase in transaction latency for $\mathbb{P}_M$ in the time window immediately after migration, with the base value being the average transaction latency observed before migration. We select 30 seconds as a representative time window based on the behavior of latency in Figure 7.4 where $\mathbb{N}_D$ is warmed up within about $30 - 40$ seconds after the completion of migration. We also measured the percentage increase in latency in the period from start of migration to 30 seconds beyond completion of the respective migration techniques. Since Albatross takes much longer to complete compared to the other techniques and has minimal impact on latency during migration, this measure favors Albatross and unfairly reports a lower increase for Albatross. Therefore, we consider the 30 second window after migration such that all techniques can be evenly evaluated.

Effect of load

Figures 7.6 and 7.7 plot migration cost as a function of the load, expressed as transactions per second (TPS), on $\mathbb{P}_M$. As the load on a partition increases (from 20 TPS to 100 TPS), the amount of un-flushed changes in the write cache also increases. Hence the unavailability window of S&M increases with load (see Figure 7.6(a)). But since both Albatross and F&M flush the cache (at least once) before the final phase, they are not heavily impacted by load. The unavailability window of Albatross increases slightly since at higher load more transaction state must be copied during the final handover phase. Similarly, a higher load implies more transactions are active at the start of migration; all such active transactions which are aborted in F&M and S&M, thus



(a) Transaction latency increase. (b) Data transfer overhead.

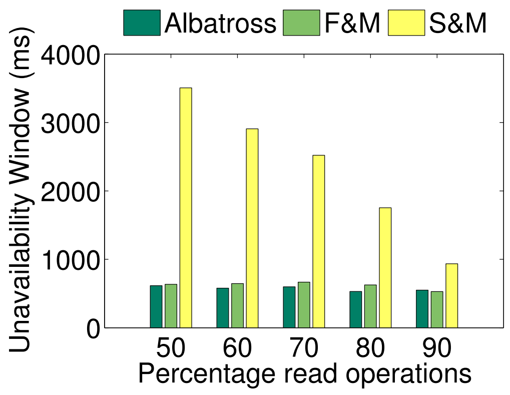
Figure 7.7: Evaluating the impact of transaction load on transaction latency and the amount of data transferred during migration.

resulting in a large number of failed requests. Figure 7.6(b) plots the number of failed requests, where the wider bars represent transactions aborted and the narrower bars represent failed operations. Albatross does not result in any failed requests since it copies transaction state and allows transactions to resume at $\mathbb{N}_D$.

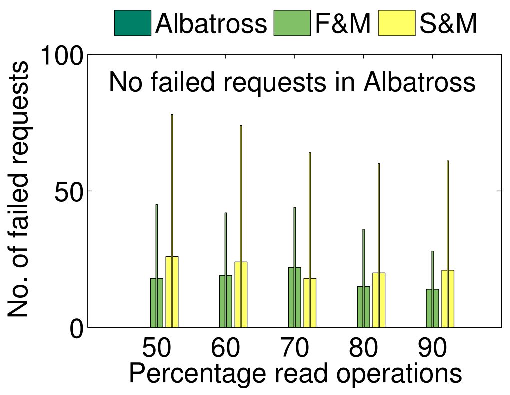
Both F&M and S&M incur a high penalty on transaction latency. The impact on latency increases with load since more read operations incur a cache miss, resulting in higher contention for accessing the NAS (see Figure 7.7(a)). Albatross results in only $5-15\%$ transaction latency increase (over $80-100$ ms average latency) in the 30 second window after migration, while both F&M and S&M result in $300-400\%$ latency increase. Finally, Figure 7.7(b) plots the amount of data synchronized as a function of load. In spite of the increase in data transmission, this does not adversely affect performance when using Albatross. Both S&M and F&M incurs the cost of warming the cache at $\mathbb{N}_D$ which starts with an empty cache. Thus, S&M and F&M incur data transfer overhead of approximately the cache size, i.e., 250 MB in this experiment.

Effect of Read/Write Ratio

We now present results from experiments varying other parameters of YCSB. Figures 7.8 and 7.9 plot the impact of varying the percentage read operations in a transaction; we vary the read percentage from 50 to 90. For an update heavy workload, the write cache has a large amount of un-flushed updates that must be flushed during migration. As a result, S&M incurs a long unavailability window of about $2-4$ seconds; the length of which decreases with a decrease in the percentage of writes (see Figure 7.8(a)). On the other hand, both F&M and Albatross flush the majority of updates

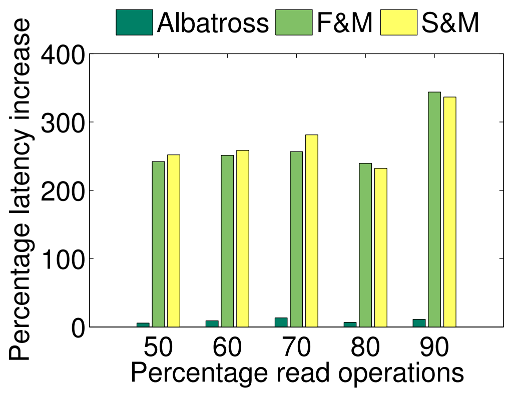


(a) Partition unavailability.

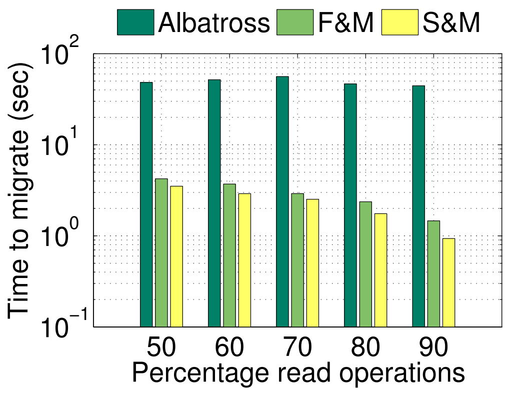


(b) Failed requests.

Figure 7.8: Evaluating the impact of varying the percentage of read operations in a transaction on partition unavailability and number of failed requests.



(a) Transaction latency increase.

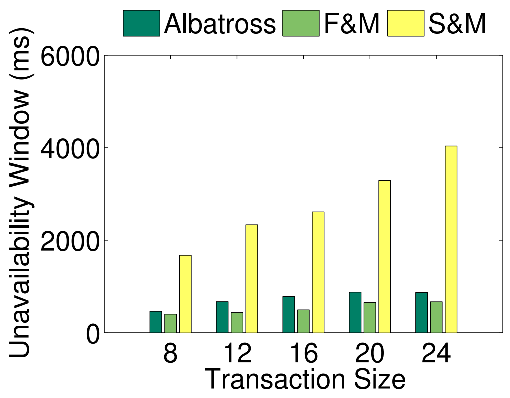


(b) Total migration time (log scale).

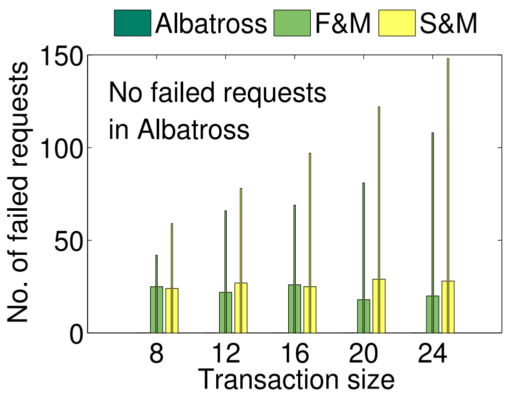
Figure 7.9: Evaluating the impact of varying the percentage of read operations in a transaction on transaction latency and the time to migrate a partition.

before the final stop phase. Therefore, their unavailability window is unaffected by the distribution of reads and writes. However, since both S&M and F&M do not migrate transaction state, all transactions active at the start of migration are aborted, resulting in a large number of failed requests (see Figure 7.8(b)). Albatross, on the other hand, does not have any failed requests.

As observed in Figure 7.9(a), Albatross results in only $5 - 15\%$ transaction latency increase, while both F&M and S&M incur a $300 - 400\%$ increase in transaction latency due to the cost of warming up the cache at the destination. Since Albatross warms up the cache at the destination during the iterative phase, the total time taken by Albatross from



(a) Partition unavailability.



(b) Failed requests.

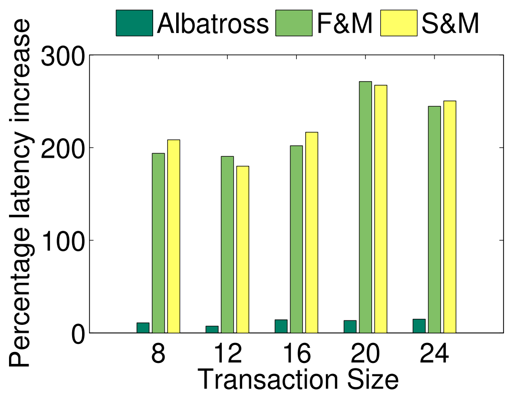
Figure 7.10: Evaluating the impact of varying the number of operations in a transaction on partition unavailability and number of failed requests.

the start to finish is much longer compared to that of F&M and S&M; S&M is the fastest followed by F&M (see Figure 7.9(b)). However, since $\mathbb{P}_M$ is still active and serving requests with no impact on transaction latency, this background loading process does not contribute to migration cost from the tenant's perspective. The iterative copying phase transfers about 340 MB data between $\mathbb{N}_S$ and $\mathbb{N}_D$, which is about 35% greater that the cache size (250 MB). F&M and S&M also incur network overhead of 250 MB resulting from cache misses at $\mathbb{N}_D$ and a fetch from NAS.

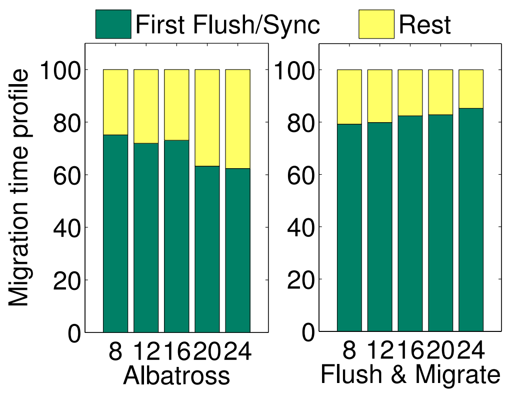
Effect of Transaction Size

Figures 7.10 and 7.11 show the effect of transaction size on migration cost; we vary the number of operations in a transaction from 8 to 24. As the transaction size increases, so does the number of updates, and hence the amount of un-flushed data in the write cache. Therefore, the unavailability window for S&M increases with increased transaction size (see Figure 7.10(a)). In this experiment, F&M has a smaller unavailability window compared to Albatross. This is because Albatross must copy the transaction state in the final handover phase, whose size increases with increased transaction size. F&M, on the other hand, aborts all active transactions and hence does not incur that cost. The number of failed requests is also higher for F&M and S&M, since an aborted transaction with more operations result in more work wasted (see Figure 7.10(b)).

The impact on transaction latency also increases with size since larger transactions have more reads (see Figure 7.11(a)). This is because the transaction load is kept constant in this experiment and more operations per transactions implies more operations issued on $\mathbb{P}_M$ per unit time. Transaction size also impacts the amount of time spent



(a) Transaction latency increase.



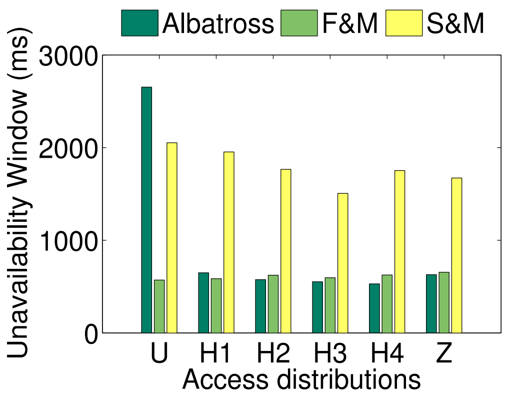
(b) Percentage time distribution.

Figure 7.11: Evaluating the impact of varying the number of operations in a transaction on latency and the distribution of time spent in the different migration phases.

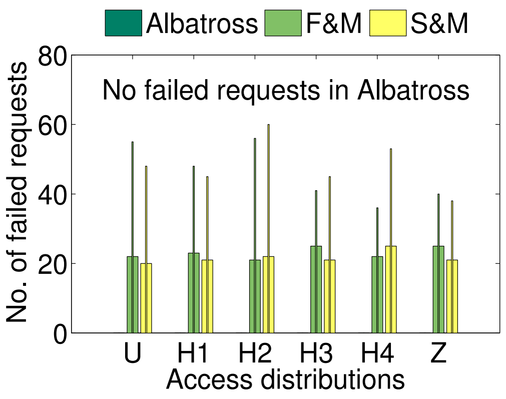
in the different migration phases; Figure 7.11(b) shows a profile of the total migration time. As expected, the majority of the time is spent in the first sync or flush, since it results in the greatest amount of data being transferred or flushed. As the number of operations in a transaction increases, the amount of state copied in the later iterations of Albatross also increases. Therefore, the percentage of time spent on the first iteration of Albatross decreases. On the other hand, since the amount of data to be flushed in F&M increases with transaction size, the time taken for the first flush increases.

Effect of Access Distributions

Figures 7.12 and 7.13 plot the migration cost as a function of the distributions that determine the data items accessed by a transaction; we experimented with uniform, Zipfian, and four different variants of the hotspot distribution where we vary the size of the hot set and the number of operations accessing the hot set. Since the cache size is set to 25% of the database size, uniform distribution incurs a high percentage of cache misses. As a result, during the iterative copy phase, the database cache changes a lot because of a lot of blocks being evicted and loaded. Every iteration, therefore, results in a significant amount of data being transferred. Albatross tracks the amounts of data transferred in each iteration and this value converges quickly; in this experiment, Albatross converged after 3 iterations. However, the final handover phase has to synchronize a significant amount of data, resulting in a longer unavailability window. Therefore, a high percentage of cache misses results in a longer unavailability window for Albatross. F&M and S&M are, however, not affected since these techniques do not copy the database cache. This effect disappears for skewed workload where as expected,



(a) Partition unavailability.



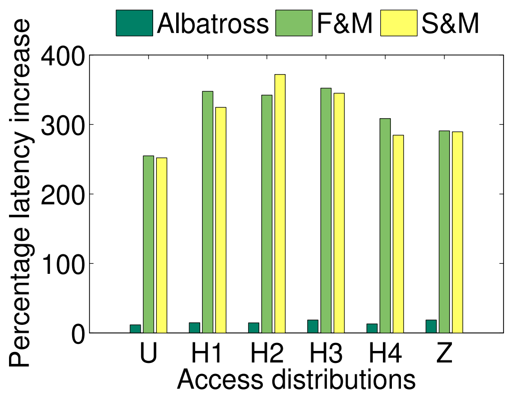
(b) Failed requests.

Figure 7.12: Evaluating the impact of different data access distributions on the partition's unavailability and the number of failed requests. U denotes uniform and Z denotes Zipfian. H1–H4 denote hotspot distributions: 90-10, 90-20, 80-10, and 80-20, where x-y denotes $x\%$ operations accessing $y\%$ data items.

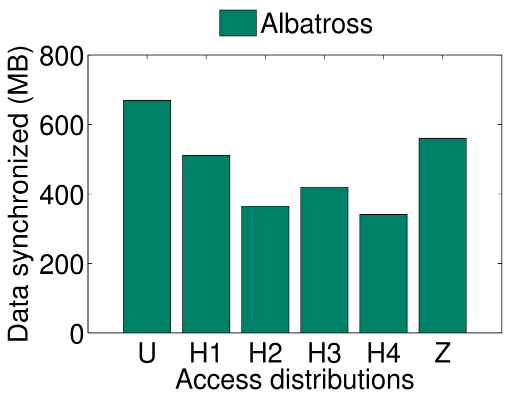
Albatross and F&M have similar unavailability window and S&M has a comparatively longer unavailability window.

Albatross does not result in any failed requests, while the number of failed requests in F&M and S&M is not heavily affected by the distribution (see Figure 7.12(b)). The uniform distribution results in a higher number of cache misses even at $\mathbb{N}_S$ which offsets the impact of cache misses at $\mathbb{N}_D$. Therefore, the percentage increase in transaction latency for S&M and F&M is lower for the uniform distribution when compared to other access patterns (see Figure 7.13(a)). Irrespective of the access distribution, Albatross has little impact on latency.

Figure 7.13(b) plots the amount of data synchronized by Albatross. Following directly from our discussion above, a uniform distribution results in larger amounts of data being synchronized when compared to other distributions. It is however interesting to note the impact of the different hotspot distributions on data synchronized. For H1 and H3, the size of the hot set is set to 10% of the database, while for H2 and H4, the size of the hot set is set to 20%. Since in H1 and H3, a fraction of the cold set is stored in the cache, this state changes more frequently compared to H2 and H4 where the cache is dominated by the hot set. As a result, H1 and H3 result in larger amounts of data synchronized. For the Zipfian distribution, the percentage of data items accessed frequently is even smaller than that in the experiments with 10% hot set, which also explains the higher data synchronization overhead.



(a) Transaction latency increase.



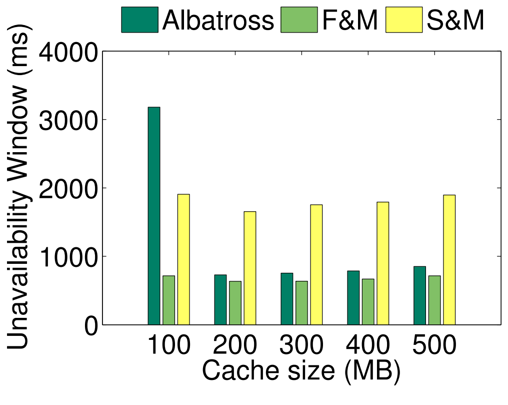
(b) Data transferred during migration.

Figure 7.13: Evaluating the impact of different data access distributions on transaction response times and the amount of data transferred during migration. U denotes uniform and Z denotes Zipfian. H1–H4 denote hotspot distributions: 90-10, 90-20, 80-10, and 80-20, where x-y denotes $x\%$ operations accessing $y\%$ data items.

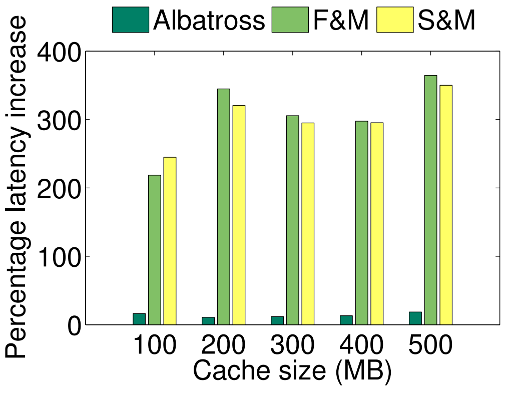
Effect of Cache Size

Figure 7.14 plots migration cost as a function of the cache size while keeping the database size fixed; the cache size is varied from 100 MB to 500 MB and the database size is 1 GB. Since Albatross copies the database cache during migration, a smaller database cache implies lesser data to synchronize. When the cache size is set to 100 MB, the unavailability window of Albatross is longer than that of F&M and S&M (see Figure 7.14(a)). This behavior is caused by the fact that at 100 MB, the cache does not entirely accommodate the hot set of the workload (which is set to 20% of the data items or 200 MB), thus resulting in a high percentage of cache misses. This impact of a high percentage of cache misses on unavailability window is similar to that observed for the uniform distribution. However, since the iterations converge quickly, the amount of data synchronized is similar to that observed in other experiments. For cache sizes of 200 MB or larger, the hot set fits into the cache, and hence expected behavior is observed. Even though Albatross has a longer unavailability window for a 100 MB cache, the number of failed operations and the impact on transaction latency continues to be low. For F&M and S&M, the impact on transaction latency is lower for the 100 MB cache because a large fraction of operations incurred a cache miss even at $\mathbb{N}_S$ which somewhat offsets the cost due to cache misses at $\mathbb{N}_D$ (see Figure 7.14(b)). Number of failed operations and data synchronized show expected behavior.

Figure 7.15 plots the impact of migration on latency as time progresses. In this experiment, we consider a scenario where the working set of the database does not fit



(a) Partition unavailability.



(b) Transaction latency increase.

Figure 7.14: Evaluating the impact of varying the cache size allocated to each partition on unavailability and the increase in transaction latency.

in the cache. The cache size is set to 100 MB when using a hotspot distribution where the hot set is 20% of the database. This experiment confirms our earlier observation that when the working set does not fit in the cache, even though Albatross results in a longer unavailability window, there is minimal impact on transaction latency.

Effect of Database Size

Figure 7.16 plots the migration cost as a function of the database size. Since $\mathbb{P}_M$'s persistent data is not migrated, the actual database size does not have a big impact on migration cost. We therefore vary the cache size along with the database size such that the cache is set to 25% of the database size. Since the cache is large enough to accommodate the hot set (we use the default hotspot distribution with the hot set as 20% of the database), the migration cost will be lower for a smaller database (with a smaller cache); the cost increases with an increase in the database size (see Figure 7.16(a)). Similarly, as the size of the database cache increases, the amount of state synchronized and the time taken for the synchronization also increases (see Figure 7.16(b)).

7.5.4 Evaluation using the TPC-C Benchmark

We now evaluate the migration cost using the TPC-C benchmark [81] adapted for a multitenant setting. The goal is to evaluate Albatross using complex transaction workloads representing real-life business logic and complex schema. Figure 7.17 plots the migration cost as the load on each tenant partition is varied; in both sub-figures, the y-axis plots the migration cost measures, while the x-axis plots the load on the system.

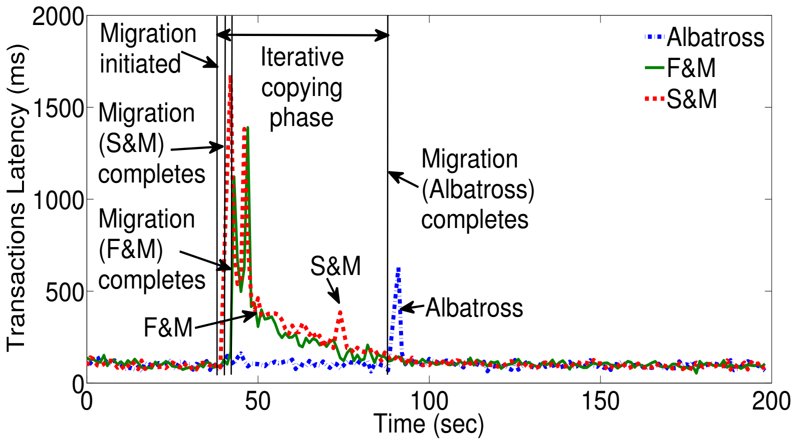


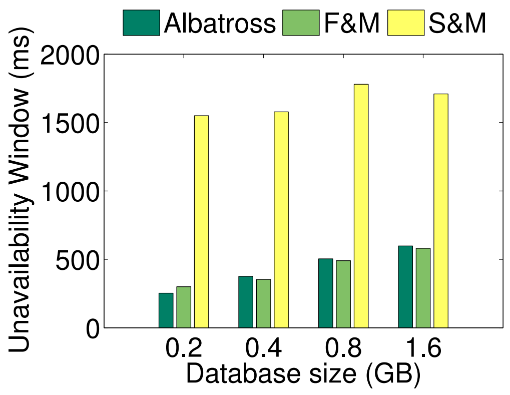
Figure 7.15: Impact of migration on transaction latency when working set does not fit in the cache. Even though Albatross results in longer unavailability window, it continues to have low impact on transaction latency.

As the load on each partition increases, the amount of data transferred to synchronize state also increases. As a result, the length of the unavailability window increases with an increase in the load on the tenant (see Figure 7.17(a)). Furthermore, since the arrival rate of operations is higher at a higher load, more transactions are active at any instant of time, including the instant when migration is initiated. As a result, S&M and F&M result in more failed requests as the load increases. This increase in number of failed requests is evident in Figure 7.17(b).

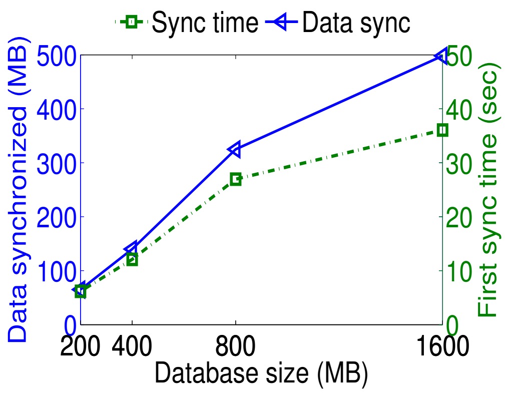
From this experiment, it is evident that even with complex transactional workloads, the performance of Albatross is considerably better than S&M and F&M. The behavior of transaction latency increase and amount of data synchronized is similar to previous set of experiments. Albatross incurred less than 15% increase in transaction latency compared to a 300% increase for F&M and S&M, while Albatross synchronized about 700 MB data during migration; the cache size was set to 500 MB.

7.5.5 Migration cost during overload

In all the previous experiments, neither $\mathbb{N}_S$ nor $\mathbb{N}_D$ were overloaded. We now evaluate the migration cost in a system with high load; we use YCSB for this experiment. Figure 7.18 shows the impact of migrating a tenant from an overloaded node ($\mathbb{N}_S$) to a lightly loaded node ($\mathbb{N}_D$). In this experiment, the load on each tenant (or partition) is set to 50 TPS and the number of tenants served by $\mathbb{N}_S$ is gradually increased to 20 when $\mathbb{N}_S$ becomes overloaded. As the load on $\mathbb{N}_S$ increases, all tenants whose database

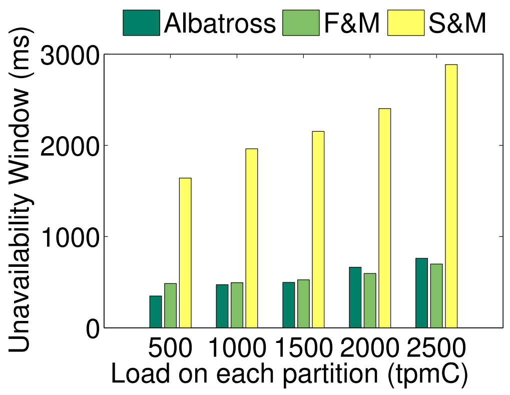


(a) Partition unavailability.

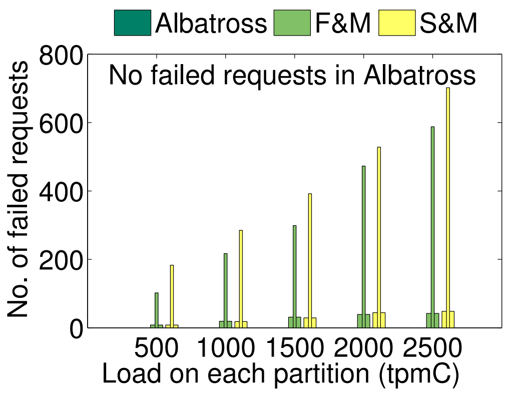


(b) Data synchronized and time taken for first synchronization.

Figure 7.16: Evaluating the impact of varying the tenant database size (or size of a partition) on the length of unavailability window and the amount of data synchronized and the time taken to complete synchronizing the initial snapshot.



(a) Unavailability window.



(b) No. of failed requests.

Figure 7.17: Evaluating migration cost using the TPC-C benchmark.

is located at $\mathbb{N}_S$ experience an increase in transaction latency. At this point, one of the tenants at $\mathbb{N}_S$ is migrated to $\mathbb{N}_D$.

In Figure 7.18, the y-axis plots the percentage change in transaction latency in the 30 second window after migration; a negative value implies reduction in latency. T_M is the tenant that was migrated, T_S is a tenant at $\mathbb{N}_S$ and T_D is a tenant at $\mathbb{N}_D$. The latency of T_M's transactions is higher when it is served by an overloaded node. Therefore, when T_M is migrated from an overloaded $\mathbb{N}_S$ to a lightly loaded $\mathbb{N}_D$, the transaction latency of T_M should decrease. This expected behavior is observed for Albatross, since it has a

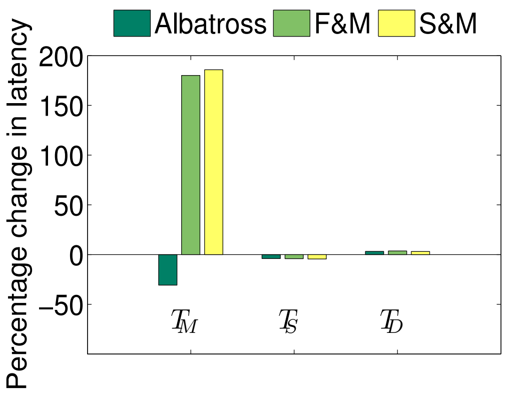


Figure 7.18: Impact of migrating tenant T_M from a heavily loaded node to a lightly loaded node. T_S and T_D represent a representative tenant at $\mathbb{N}_S$ and $\mathbb{N}_D$ respectively.

low migration cost. However, the high cost of F&M and S&M result in an increase in transaction latency even after migrating T_M to a lightly loaded $\mathbb{N}_D$. This further asserts the effectiveness of Albatross for elastic scaling/load balancing when compared to other heavyweight techniques like S&M and F&M. All migration techniques, however, have low overhead on other tenants co-located at $\mathbb{N}_S$ and $\mathbb{N}_D$. This low overhead is evident from Figure 7.18, where a small decrease in latency of T_S results from lower aggregate load on $\mathbb{N}_S$ and a small increase in transaction latency of T_D results from the increased load at $\mathbb{N}_D$.

Discussion

A low cost migration technique is important to allow aggressive consolidation in a multitenant system. It is evident from our experiments that Albatross results in a much lower performance impact than S&M and F&M. In Albatross, the impact of cache size on transaction latency is not significant. However, a larger cache results in a more significant impact on S&M and F&M due to the higher cost of warming up the larger cache at $\mathbb{N}_D$. Therefore, as the available memory and hence the available cache at the servers grow, S&M and F&M are expected to have a more significant impact on transaction latency than that of Albatross. As servers grow in memory, so will the network bandwidth. However, the rate of growth in memory is expected to be faster than that of the network bandwidth. The benefits of Albatross will be even more significant in such scenarios.

Albatross enables the system to guarantee that if the need arises, tenants can be migrated to ensure that their SLAs are met. For instance, as is evident from the experiment reported in Figure 7.18, even though the load on every tenant at $\mathbb{N}_S$ is only 50

TPS, as the number of tenants at $\mathbb{N}_S$ increases, it causes an overload. A low cost migration technique can help alleviate such scenarios commonly encountered in multitenant systems.

Albatross is also useful in the scenario of a load spike. The system can either migrate other lightly loaded tenants from the overloaded node to another node or migrate the overloaded tenant to a node with more resources. The first option minimizes the total load on the source while isolating other tenants from being impacted by the heavily loaded tenant. Moreover, as observed in our experiments, migrating lightly loaded tenants is less expensive than migrating a highly loaded tenant. On the other hand, the second option requires migrating only one tenant, though at a higher migration cost. The migration strategy chosen will depend on the workload and tenant characteristics. An intelligent system controller can make prudent use of live migration for elastic load balancing.

7.6 Summary

In this chapter, we presented Albatross, a technique for live database migration in a decoupled storage architecture that results in minimal performance impacts and minimal disruption in service for the tenant whose database is being migrated. Albatross decouples a partition from the DBMS node *owning* it, and allows the system to routinely use migration as a primitive for elastic load balancing. Since the persistent data is not migrated in a decoupled storage architecture, Albatross focusses on migrating the database cache and the state of the active transactions. This ensures that the destination node of migration starts with a warm cache, thus minimizing the impact of migration. We presented the detailed design of Albatross and discussed its correctness guarantees and behavior in the presence of failures. We also demonstrated the effectiveness of Albatross and analyzed its trade-offs using two OLTP benchmarks, YCSB and TPC-C. Our evaluation showed that Albatross can migrate a live database partition with no aborted transactions, negligible impact on transaction latency and throughput both during and after migration, and an unavailability window as low as 300 ms.

Albatross is the first end-to-end solution for live database migration in a decoupled storage architecture. Our focus was therefore on proving feasibility and ensuring the safety in the presence of failures. DBMSs provide stringent guarantees in the event of a failure and all those guarantees must be provided even during migration. Various simple optimizations are possible in the design of Albatross. For instance, since Albatross iteratively copies the state of the database cache, segments that are frequently being updated are copied more than once. Incorporating the access patterns into deciding which segments to copy during an iteration might reduce the amount of data transferred during migration. It will also be useful to *predict* the migration cost so that the system controller can effectively use migration without violating the SLAs. In the future, we would like to explore the feasibility of using machine learning techniques, such as kernel canonical correlation analysis [10], for predicting the migration cost.

Chapter 8

Shared Nothing Architectures

"The gentle wind that blows And whispers as it goes."

– Ina L. Jenkins.

In this chapter, we present the detailed design and implementation of Zephyr [39], a live migration technique for shared nothing database architectures.[1] In a shared nothing database architecture, the persistent data is stored on disks locally attached to the DBMS servers. This architecture is used in many systems, such as Cloud SQL Server [14], Relational Cloud [28], and MySQL Cluster. In a shared nothing architecture, the persistent data must also be moved when migrating a database partition. This is contrary to the decoupled storage architecture where persistent data is not migrated. Zephyr assumes a shared process multitenancy model and migrates a database partition ($\mathbb{P}_M$) from the source DBMS node ($\mathbb{N}_S$) to the destination DBMS node ($\mathbb{N}_D$).[2]

Zephyr views migration as a sequence of phases, called migration modes, and migrates $\mathbb{P}_M$ with no unavailability. Zephyr guarantees no downtime by introducing a synchronized *dual mode* that allows both $\mathbb{N}_S$ and $\mathbb{N}_D$ to simultaneously execute transactions on $\mathbb{P}_M$. Migration starts with the transfer of $\mathbb{P}_M$'s metadata to $\mathbb{N}_D$ which can then start serving new transactions, while $\mathbb{N}_S$ completes the transactions that were active when migration started. Zephyr views a partition as a collection of database pages. Read/write access (or ownership) on the database pages of $\mathbb{P}_M$ is partitioned between the two nodes with $\mathbb{N}_S$ owning all pages at the start and $\mathbb{N}_D$ acquiring page ownership on-demand as transactions at $\mathbb{N}_D$ access those pages. At any instant of time, at most one

[1]Zephyr, meaning a gentle breeze, is symbolic of the lightweight nature of the proposed technique.

[2]The work reported in this chapter was published as the paper entitled "Zephyr: live migration in shared nothing databases for elastic cloud platforms" in the proceedings of the 2011 ACM International Conference on Management of Data (SIGMOD). DOI: `http://doi.acm.org/10.1145/1989323.1989356`.

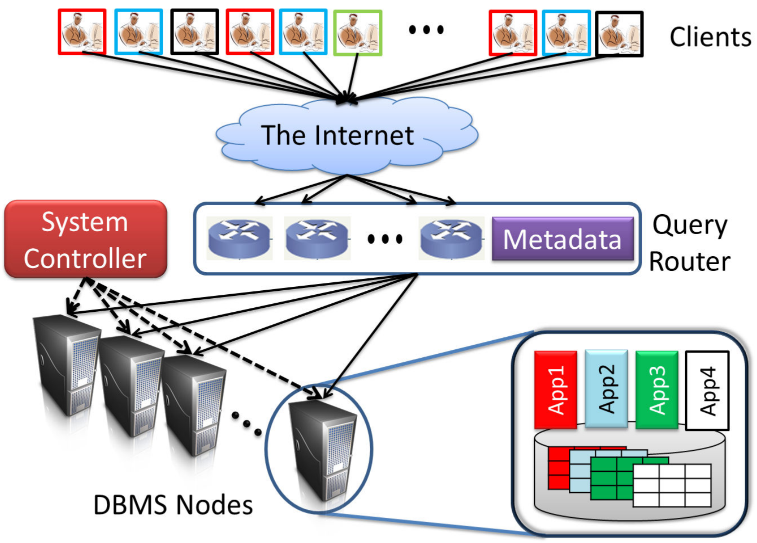


Figure 8.1: A reference shared nothing multitenant DBMS architecture.

of $\mathbb{N}_S$ or $\mathbb{N}_D$ owns a database page. Lightweight synchronization between the source and the destination, only during the short dual mode, guarantees serializability, while obviating the need for 2PC. Once $\mathbb{N}_S$ completes execution of all active transactions, migration completes by transferring ownership of all database pages owned by $\mathbb{N}_S$ to $\mathbb{N}_D$. Zephyr minimizes the amount of data transferred between the nodes during migration, guarantees *correctness* in the presence of failures, and ensures the strongest level of transaction isolation. Zephyr uses standard tree-based indices and lock-based concurrency control, thus allowing it to be used in a variety of RDBMS implementations.

8.1 Reference System Architecture

We consider a standard shared nothing database architecture for OLTP systems executing short-running transactions, with a two phase locking [40] based scheduler, and a page-based model with a B+ tree index [15]. Figure 8.1 provides an overview of the architecture. Following are the salient features of the system.

First, clients connect to the database through query routers that handle client connections and hide the physical location of the tenant's database. Routers store this mapping as metadata which is updated whenever there is a migration.

Second, a cluster of DBMS nodes serves the partitions; each node has its own local transaction manager (TM) and data manager (DM). A partition is served by a single DBMS node, called its owner. We consider the shared process multitenancy model

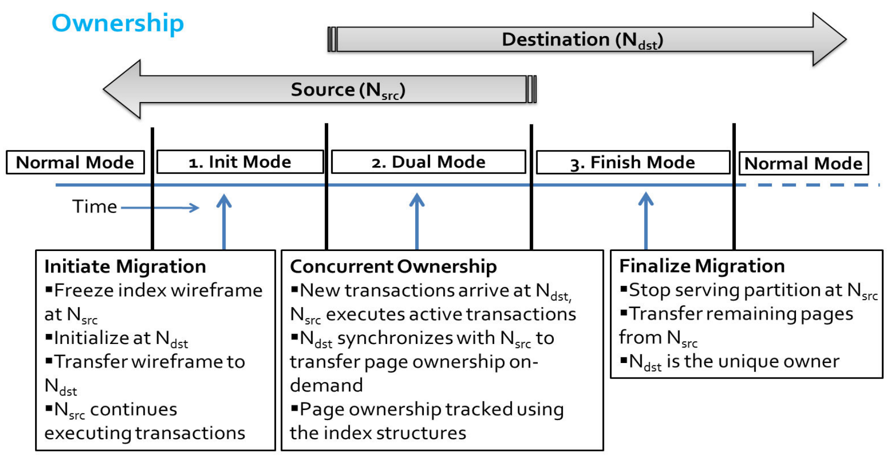

Ownership
Destination (N_{dst})
Source (N_{src})
Normal Mode
1. Init Mode
2. Dual Mode
3. Finish Mode
Normal Mode
Time
Initiate Migration
▪Freeze index wireframe at N_{src}
▪Initialize at N_{dst}
▪Transfer wireframe to N_{dst}
▪N_{src} continues executing transactions
Concurrent Ownership
▪New transactions arrive at N_{dst}, N_{src} executes active transactions
▪N_{dst} synchronizes with N_{src} to transfer page ownership on-demand
▪Page ownership tracked using the index structures
Finalize Migration
▪Stop serving partition at N_{src}
▪Transfer remaining pages from N_{src}
▪N_{dst} is the unique owner

Figure 8.2: Migration timeline for Zephyr (times not drawn to scale).

which strikes a balance between isolation and scale. Conceptually, each partition has its own transaction manager and buffer pool. However, since most current systems do not support this, we use a design where co-located partitions share all resources within a database instance.

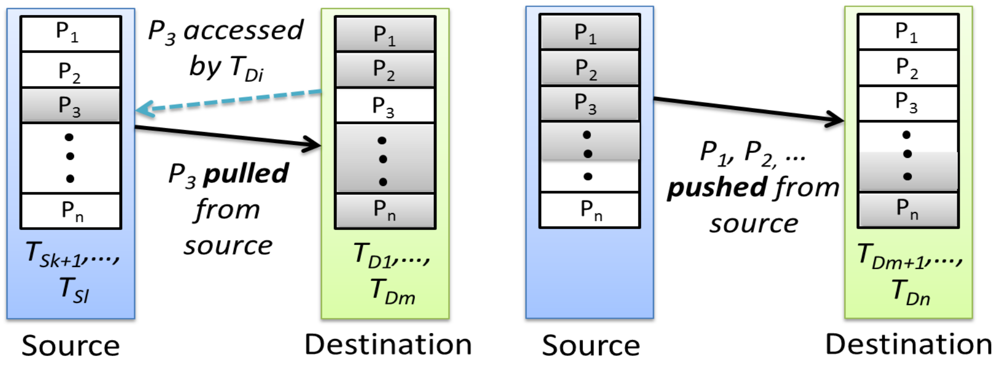
Third, there exists a system controller that determines the partition to be migrated, the initiation time, and the destination of migration. The system controller gathers usage statistics and builds a model to optimize the system's operating cost while guaranteeing the SLAs.

8.2 The Zephyr Technique

To ease presentation, in this section, we provide an overview of Zephyr using some simplifying assumptions. We assume *no failures*, *small tenants* limited to a single database partition, and *no replication*. For small tenants limited to a single partition, we use the terms tenant and partition interchangeably. Furthermore, we **freeze** the *index structures* during migration, i.e., we disallow any structural changes to them. Failure handling and correctness is discussed in Section 8.3, while an extended design relaxing these assumptions is described in Section 8.4.

8.2.1 Design Overview

Zephyr's main design goal is to minimize the service interruption resulting from migrating a tenant's database ($\mathbb{P}_M$), i.e., migrate $\mathbb{P}_M$ without making it unavailable for



(a) Dual Mode. (b) Finish Mode.

Figure 8.3: Ownership transfer of the database pages during migration. P_i represents a database page. A white box around P_i represents that the node currently *owns* the page while a grayed box around P_i implies that the node *knows* about P_i but does not own it.

updates. Zephyr uses a sequence of three modes to allow the migration of $\mathbb{P}_M$ while transactions are executing on it. During normal operation (the **Normal Mode**), $\mathbb{N}_S$ is the node serving $\mathbb{P}_M$ and executing all transactions $(T_{S1}, \ldots, T_{Sk})$ on $\mathbb{P}_M$, i.e., $\mathbb{N}_S$ is the *owner* of $\mathbb{P}_M$. Once the system controller determines the destination for migration ($\mathbb{N}_D$), it notifies $\mathbb{N}_S$ which initiates migration to $\mathbb{N}_D$.

Figure 8.2 shows the different migration modes as time progresses from the left to the right: the **Init Mode** where migration starts, the **Dual Mode** where both $\mathbb{N}_S$ and $\mathbb{N}_D$ share the ownership of $\mathbb{P}_M$ and simultaneously execute transactions on $\mathbb{P}_M$, and the **Finish Mode** which is the last step of migration before $\mathbb{N}_D$ assumes sole ownership of $\mathbb{P}_M$. Figure 8.3 shows the transition of $\mathbb{P}_M$'s data through the three migration modes, depicted using ownership of database pages and executing transactions. We now explain the three migration modes in detail.

Init Mode In the Init Mode, $\mathbb{N}_S$ bootstraps $\mathbb{N}_D$ by sending the minimal information (the **wireframe** of $\mathbb{P}_M$) that allows $\mathbb{N}_D$ to execute transactions on $\mathbb{P}_M$. The wireframe consists of the schema and data definitions of $\mathbb{P}_M$, the index wireframe, and user authentication information. In this mode, $\mathbb{N}_S$ is still the unique owner of $\mathbb{P}_M$ and executes transactions $(T_{S1}, \ldots, T_{Sk})$ without synchronizing with any other node. However, the wireframe is made immutable and remains in this frozen state for the remaining duration of migration. Therefore, there is no service interruption for $\mathbb{P}_M$ while $\mathbb{N}_D$ initializes the necessary resources for $\mathbb{P}_M$.

Zephyr views a database partition as a collection of database pages with some index to keep track of the database pages. Indices in a database include the clustered index

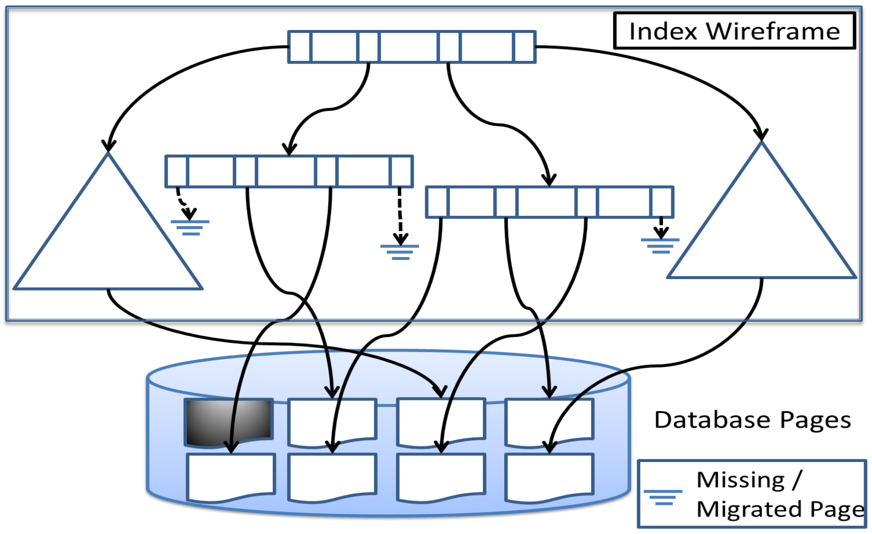


Figure 8.4: A B+ tree index structure with page ownership information. The rectangle enclosing the internal nodes of the tree index comprises the index wireframe. Zephyr augments the index to also store the page ownership information. A page in white represents an owned page. A sentinel marks missing pages. An allocated database page without ownership is represented as a grayed page.

storing the database and all the secondary indices; non-indexed attributes are accessed through the clustered index. For concreteness, we assume a B+ tree index, where the internal nodes of the index contain only the keys while the actual data pages are in the leaves. The index wireframe therefore only includes the internal nodes of the indices for the database tables. Figure 8.4 provides an illustration of a B+-tree index where the part of the tree enclosed in a rectangular box is the index wireframe.

$\mathbb{N}_S$ constructs the wireframe with minimal impact on concurrent operations using shared multi-granularity intention locks on the indices. When $\mathbb{N}_D$ receives the wireframe, it has $\mathbb{P}_M$'s metadata, but the data pages are still owned by $\mathbb{N}_S$. Since migration involves a gradual transfer of page level ownership, both $\mathbb{N}_S$ and $\mathbb{N}_D$ must maintain a list of owned pages. We use the B+ tree index for tracking page ownership. A valid pointer to a database page implies unique page ownership, while a sentinel value (`NULL`) indicates a missing page. In the init mode, $\mathbb{N}_D$ therefore initializes all the pointers to the leaf nodes of the index to the sentinel value. Once $\mathbb{N}_D$ completes initialization of $\mathbb{P}_M$, it notifies $\mathbb{N}_S$, which then initiates the transition to dual mode. $\mathbb{N}_S$ then executes an **atomic handover** protocol, a variant of the 2PC protocol [44], which notifies the *query router* to route all new transactions to $\mathbb{N}_D$ which now shares the ownership of $\mathbb{P}_M$ with $\mathbb{N}_S$. After the atomic handover, migration enters dual mode.

Dual Mode In dual mode, both $\mathbb{N}_S$ and $\mathbb{N}_D$ execute transactions on $\mathbb{P}_M$, and database pages are migrated to $\mathbb{N}_D$ *on-demand*. All new transactions $(T_{D1}, \ldots, T_{Dm})$ arrive at $\mathbb{N}_D$, while $\mathbb{N}_S$ continues executing transactions that were active at the start of this mode $(T_{Sk+1}, \ldots, T_{Sl})$. Since $\mathbb{N}_S$ and $\mathbb{N}_D$ share ownership of $\mathbb{P}_M$, they synchronize to ensure transaction correctness. Zephyr, however, requires minimal synchronization between these nodes.

When a transaction T_{Di} executing at $\mathbb{N}_D$ accesses a page P_i that is not owned by $\mathbb{N}_D$, it **pulls** P_i from $\mathbb{N}_S$ *on demand* (pull phase as shown in Figure 8.3(a)). This pull request is serviced only if P_i is not locked at $\mathbb{N}_S$, in which case the request is blocked. If P_i is available, or once it becomes available, $\mathbb{N}_S$ updates its index to relinquish ownership and P_i is migrated to $\mathbb{N}_D$ which becomes its unique owner. Zephyr migrates a page only once from $\mathbb{N}_S$ to $\mathbb{N}_D$. As the pages are migrated, both $\mathbb{N}_S$ and $\mathbb{N}_D$ update their ownership mapping. Once $\mathbb{N}_D$ receives P_i, it proceeds to execute T_{Di}. At $\mathbb{N}_S$, transactions execute normally using local index and page level locking, until a transaction T_{Sj} accesses a page P_j which has already been migrated. In our simple design, a database page is migrated only once. Therefore, such an access fails and the transaction is aborted at $\mathbb{N}_S$.

Apart from fetching missing pages from $\mathbb{N}_S$, transactions at $\mathbb{N}_S$ and $\mathbb{N}_D$ do not need to synchronize. Since the index wireframe is frozen, local locking of the index structure and pages is enough; a formal proof of this appears later in the chapter. This ensures minimal synchronization between $\mathbb{N}_S$ and $\mathbb{N}_D$ only during the short dual mode while ensuring serializable transaction execution.

When $\mathbb{N}_S$ has finished executing all transactions $T_{Sk+1}, \ldots, T_{Sl}$ that were active at the start of dual mode (i.e., $\mathbb{T}(\mathbb{N}_S) = \phi$), $\mathbb{N}_S$ initiates transfer of exclusive ownership to $\mathbb{N}_D$. This transfer is achieved through a handshake between $\mathbb{N}_S$ and $\mathbb{N}_D$ after which both nodes enter the finish mode for $\mathbb{P}_M$.

Finish Mode In the finish mode, $\mathbb{N}_D$ is the only node executing transactions on $\mathbb{P}_M$ $(T_{Dm+1}, \ldots, T_{Dn})$. However, $\mathbb{N}_D$ does not yet have ownership of all the database pages (Figure 8.3(b)). In this phase, $\mathbb{N}_S$ **pushes** the remaining database pages to $\mathbb{N}_D$. While the pages are migrated from $\mathbb{N}_S$, if a transaction T_{Di} accesses a page that is not yet owned by $\mathbb{N}_D$, the page is requested as a *pull* from $\mathbb{N}_S$ in a way similar to that in dual mode. The index metadata is used to detect duplicate pages that were pushed and pulled concurrently.

$\mathbb{N}_S$ can migrate the pages at the highest possible transfer rate such that the delays resulting from $\mathbb{N}_D$ fetching missing pages is minimized. However, such a high throughput push can impact other tenants co-located at $\mathbb{N}_S$ and $\mathbb{N}_D$. Therefore, the rate of transfer is a trade-off between the tenant SLAs and migration overhead. The page ownership information is also updated during this bulk transfer. When all the database pages have been moved to $\mathbb{N}_D$, $\mathbb{N}_S$ initiates the termination of migration so that operation switches back to the normal mode. This again involves a handshake between $\mathbb{N}_S$ and $\mathbb{N}_D$. On successful completion of this handshake, it is guaranteed that $\mathbb{N}_D$ has a persistent image of $\mathbb{P}_M$, and so $\mathbb{N}_S$ can safely relinquish all of $\mathbb{P}_M$'s resources. Once migration terminates, $\mathbb{N}_D$ executes transactions on $\mathbb{P}_M$ without any interaction with $\mathbb{N}_S$ and $\mathbb{N}_S$ notifies the system controller.

8.2.2 Migration Cost Analysis

Migration cost in Zephyr results from copying the initial wireframe, operation overhead during migration, and transactions or operations aborted during migration. In the wireframe transferred, the schema and authentication information is typically small. The indices for the tables however have a non-trivial size. A simple analysis provides an estimate of index sizes.

Let us assume 4 KB pages, 8 byte keys (integers or double precision floating point numbers), and 4 byte pointers. Each internal node in the tree can hold about $4096/12 \approx 340$ keys. Therefore, a three-level B+ tree can have up to $340^2 = 115600$ leaf nodes, which can index a $(115600 \times 4096 \times 0.8)/10^6 \approx 400$ MB database, assuming 80% page utilization. Similarly, a four-level tree can index a 125 GB database. For a three level tree, the size of the wireframe is a mere $340 \times 4096/10^6 \approx 1.4$ MB while for a 4-level tree, it is about 400 MB. For most multitenant databases whose representative sizes are in the range of hundreds of megabytes to a few gigabytes [88, 90], an index size of the order of tens of megabytes is a realistic conservative estimate. These index sizes add up for the multiple tables and indices maintained for the database.

Overhead during migration stems from creating the wireframe and fetching pages over the network. $\mathbb{N}_S$ uses standard *multi-granularity* locking [45] of the index to construct the index wireframe. This scan to create the wireframe needs intention read locks at the internal nodes which only conflict with write locks [15] on the internal node. Therefore, this scan can execute in parallel with any transaction T_{Si} executing at $\mathbb{N}_S$, only blocking update transactions that result in an update in the index structure that requires a conflicting write lock on an internal node.

On the other hand, on-demand pull of a page from $\mathbb{N}_S$ over the network is also not very expensive compared to fetches from the disk; disks have an access latency of about a millisecond while most data center networks have round trip latencies of less than a millisecond. In theory, the pull requests can result in random I/O at $\mathbb{N}_S$. However, due to temporal and spatial locality of data accesses observed in practice, most of these pages are served from the cache at $\mathbb{N}_S$. Hence, the cost incurred by a remote pull is therefore of the same order as a cache miss during normal operation resulting in a disk access. Assuming an OLTP workload with predominantly small transactions, the period for which $\mathbb{P}_M$ remains in dual mode is expected to be small. Therefore, the cost incurred in this short period in dual mode is expected to be small.

Another contributor to the migration cost is failed transactions at $\mathbb{N}_S$ that access pages already migrated. In its simplest form as described, Zephyr does not guarantee zero transaction failure; this however can be guaranteed by an extended design as shown later in Section 8.4. However, Zephyr guarantees no unavailability for $\mathbb{P}_M$ since at least one of $\mathbb{N}_S$ or $\mathbb{N}_D$ is available to execute transactions on $\mathbb{P}_M$.

8.2.3 Discussion

Zephyr's approach to migrating a database partition is radically different from Albatross's. We now discuss why a variant of Albatross, adapted to the shared nothing architecture, is not the best possible alternative and rationalize the design of Zephyr.

The key aspect of Albatross is to minimize the long unavailability of the *stop and copy* technique which arises due to the time taken to create the checkpoint and to copy it to the destination. It is possible to adapt Albatross for the shared nothing architecture, we refer to this adaptation as Iterative State Replication (**ISR**). ISR uses an iterative approach, similar to Albatross, where the checkpoint is created and iteratively copied. $\mathbb{N}_S$ checkpoints $\mathbb{P}_M$ and starts migrating the checkpoint to $\mathbb{N}_D$ while $\mathbb{N}_S$ continues serving requests. While $\mathbb{N}_D$ loads the checkpoint, $\mathbb{N}_S$ maintains the differential changes, which are then iteratively copied until the amount of change to be transferred is small enough or a maximum iteration count is reached. At this point, a final stop and copy is performed. The iterative copy can be performed using either page level copying or shipping the transaction log and replaying it at the destination.

In case ISR updates the cached pages in-place, ISR will create multiple checkpoints during migration, thus resulting in higher disk I/O at the source. In case the cache pages are copied-on-write, ISR will result in higher CPU and network overhead. Therefore, when migrating a tenant from a heavily loaded source node, this additional overhead, i.e., the additional disk and network I/O, can result in significant impact on co-located tenants which are potentially already I/O limited. However, since ISR replays the log at the destination, transactions executing at the destination will have lesser impact on response times due to a partially warm cache, thus minimizing the post migration overhead. On the other hand, Zephyr does not incur additional disk I/O at the source due to checkpointing, but the cold start at the destination results in higher post migration overhead and more I/O at the destination. Therefore, Zephyr results in less overhead at the source and is suitable for scale-out scenarios where the source is already heavily loaded, while ISR is attractive for consolidation during scale-down where it will result in lower impact on tenants co-located at the destination.

Furthermore, the iterative copying of differential updates in ISR can lead to more data being transferred during migration, because some pages are transferred more than once. This is especially true for update heavy workloads that result in more changes to the database state. Zephyr, on the other hand, migrates a database page only once and hence is expected to have lower data transfer overhead.

Consider applications such as shopping cart management or online games such as Farmville that represent workloads with a high percentage of reads followed by updates, and that require high availability for continued customer satisfaction. In ISR, $\mathbb{P}_M$ is unavailable to updates during the final stop phase. Even though the system can potentially serve read-only transactions during this window, *all* transactions with at least one update will be aborted during this *small* window. On the other hand, Zephyr does not render $\mathbb{P}_M$ unavailable by allowing concurrent transaction execution at both $\mathbb{N}_S$ and $\mathbb{N}_D$. However, during migration, Zephyr will abort a transaction in two cases: (i) if at $\mathbb{N}_S$, the transaction accesses an already migrated page, or (ii) if at either node, the transaction issues an update operation that modifies the structure of the index. Hence, Zephyr may abort a fraction of update transactions during migration. The exact impact of either technique on transaction execution will depend on the workload characteristics, and needs to be evaluated experimentally.

Finally, since ISR creates a replica of the tenant's state at another node, it can iteratively copy the updates to multiple nodes, thus creating replicas on the fly during migration. Zephyr however does not allow for this easy extension.

We believe that Zephyr provides a more lightweight alternative for the shared nothing database architectures. This chapter, therefore, focusses on Zephyr since it is expected to have minimal service interruption which is critical to ensure high availability.

8.3 Correctness and Fault Tolerance

Any migration technique should guarantee transaction correctness and migration safety in the presence of arbitrary failures. We first prove that Zephyr guarantees serializable isolation even during migration. We then prove the atomicity and durability properties of both transaction execution as well as the migration protocol.

8.3.1 Isolation guarantees

We assume that transactions executing with serializable isolation use strict 2PL [15] with multi-granularity locking [45]. We use strict 2PL where all locks for a transaction are held until it completes; most common RDBMSs implement this variant of 2PL. In the init mode and finish mode, only one of $\mathbb{N}_S$ and $\mathbb{N}_D$ is executing transactions on $\mathbb{P}_M$. The init mode is equivalent to normal operation while in finish mode, $\mathbb{N}_S$ acts as the storage node for the database serving pages on demand. Guaranteeing serializability is straightforward in these modes. Therefore, we only need to prove the isolation guarantees in dual mode where both $\mathbb{N}_S$ and $\mathbb{N}_D$ are executing transactions on $\mathbb{P}_M$.

In dual mode, $\mathbb{N}_S$ and $\mathbb{N}_D$ share the internal nodes of the index which are immutable in our design, while the leaf nodes (i.e., the data pages) are still uniquely owned by one of the two nodes. To guarantee serializability, we first prove that the phantom problem [40] is impossible. The phantom problem arises from predicate based accesses where a transaction inserts or deletes an item that matches the predicate of a concurrent transaction. We then prove that a serialization graph for transactions executing in dual mode can not have a cycle. A serialization graph is formed with transactions represented as vertices and a conflict between two transactions represented as an edge [15, 87].

Lemma 8.3.1. ***Impossibility of Phantoms:*** *Local predicate locking at the internal index nodes and exclusive page level locking between nodes is enough to ensure impossibility of phantoms.*

Proof. Assume for contradiction that a phantom is possible resulting in predicate instability. Let T_1 and T_2 be two transactions such that T_1 has a predicate and T_2 is inserting (or deleting) at least one element that matches T_1's predicate. T_1 and T_2 cannot be executing at the same node, since local predicate locking would prevent such a behavior. Therefore, these transactions must be executing on different nodes. Without loss of generality, assume that T_1 is executing at $\mathbb{N}_S$ and T_2 is executing at $\mathbb{N}_D$. Let T_1's predicate match pages $P_i, P_{i+1}, \ldots, P_j$ representing a range of keys. Since Zephyr does not allow a write operation that changes the index during migration, T_2 cannot insert to a newly created page at $\mathbb{N}_D$. Therefore, if T_2 was inserting to (or deleting from) one of the pages $P_i, P_{i+1}, \ldots, P_j$ while T_1 was executing, then it implies that both $\mathbb{N}_S$ and $\mathbb{N}_D$ have ownership of the page. This results in a contradiction, since a database page can be owned by at most one or $\mathbb{N}_S$ and $\mathbb{N}_D$. □

Lemma 8.3.2. ***Serializability at a node:*** *Transactions executing at the same node (either $\mathbb{N}_S$ or $\mathbb{N}_D$) cannot have a cycle in the serialization graph involving these transactions.*

The proof of Lemma 8.3.2 follows directly from the correctness of 2PL [40], since all transactions executing at the same node use 2PL for concurrency control.

Lemma 8.3.3. *Let T_{Sj} be a transaction executing at $\mathbb{N}_S$ and T_{Di} be a transaction executing at $\mathbb{N}_D$. It is impossible to have a conflict dependency $T_{Di} \rightarrow T_{Sj}$.*

Proof. Assume for contradiction that there exists a dependency of the form $T_{Di} \rightarrow T_{Sj}$. This implies that T_{Sj} makes a conflicting access to an item in page P_i at $\mathbb{N}_S$ after T_{Di}

accessed P_i at $\mathbb{N}_D$. This leads to a contradiction since in Zephyr, once P_i is migrated from $\mathbb{N}_S$ to $\mathbb{N}_D$, all subsequent accesses to P_i at $\mathbb{N}_S$ fail. □

Corollary 8.3.4 follows by applying induction on Lemma 8.3.3.

Corollary 8.3.4. *It is impossible to have a path $T_{Di} \rightarrow \ldots \rightarrow T_{Sj}$ in the serialization graph.*

Theorem 8.3.5. ***Serializability in dual mode.*** *It is impossible to have a cycle in the serialization graph of transactions executing in dual mode.*

Proof. Assume for contradiction that there exists a set of transactions $T_1, T_2, \ldots, T_k$ such that there is a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_k \rightarrow T_1$ in the serialization graph. If all transactions executed at the same node, then this contradicts Lemma 8.3.2. So suppose some transactions executed at $\mathbb{N}_S$ and some at $\mathbb{N}_D$. Let us first assume that T_1 executed at $\mathbb{N}_S$. Let T_i be the first transaction in the sequence that executed at $\mathbb{N}_D$. If $i = 1$, then all transactions in the cycle executed at $\mathbb{N}_D$. However, since the transactions use 2PL, the cycle is impossible. If not, then there exists a non-empty path $T_i \rightarrow \ldots \rightarrow T_1$ where T_i executed at $\mathbb{N}_D$ and T_1 executed at $\mathbb{N}_S$. This contradicts Corollary 8.3.4. If T_1 executed at $\mathbb{N}_D$, then there exists at least one transaction T_j which executed at $\mathbb{N}_S$, which implies a path of the form $T_1 \rightarrow \ldots \rightarrow T_j$, again a contradiction to Corollary 8.3.4. □

Snapshot Isolation (SI) [12] can also be guaranteed in Zephyr. A transaction T_i writing to a page P_i must have unique ownership of P_i, while a read can be performed from a snapshot shared by both nodes. This condition of unique page ownership is sufficient to ensure that during validation of transactions in SI, the transaction manager can detect two concurrent transactions writing to the same page and abort one. Zephyr therefore guarantees transactional isolation with minimal synchronization and without much migration overhead.

8.3.2 Fault tolerance

Our failure model assumes that all message transfers use reliable communication channels that guarantee in-order, at least once delivery. We consider node crash failures and network partitions; however, we do not consider malicious node behavior. We assume that a node failure does not lead to loss of the persistent data stored in a disk. In case of a failure during migration, our design first recovers the state of the committed transactions and then recovers the state of migration.

Transaction State Recovery

Transactions executing during migration use write ahead logging for transaction state recovery [15, 69]. Updates made by a transaction are *forced* to the log before it commits, thus resulting in a total order of transactions executing at the node. After a crash, a node recovers its transaction state using standard log replay techniques, ARIES [69] being an example.

In dual mode, $\mathbb{N}_S$ and $\mathbb{N}_D$ append transactions to their respective node's local transaction log. Log entries in a single log file have a local order. However, since the log for $\mathbb{P}_M$ is spread over $\mathbb{N}_S$ and $\mathbb{N}_D$, a logical global order of transactions on $\mathbb{P}_M$ is needed to ensure that the transactions from the two logs are applied in the correct order to recover from a failure during migration. The ordering of transactions is important only when there is a conflict between two transactions. If two transactions, T_S and T_D, executing on $\mathbb{N}_S$ and $\mathbb{N}_D$, conflict on item i, they must access the same database page P_i. Since at any instant of time only one of $\mathbb{N}_S$ and $\mathbb{N}_D$ is the owner of P_i, the two nodes must synchronize on P_i. This synchronization establishes a total order between the transactions. During migration, a **commit sequence number** (CSN) is assigned to every transaction at commit time, and is appended along with the commit record of the transaction. This CSN is a monotonically increasing sequence number maintained locally at the nodes and determines the order in which transactions commit. If P_i was owned by $\mathbb{N}_S$ and T_S was the last committed transaction before the migration request for P_i was made, then $CSN(T_S)$ is piggy-backed with P_i. On receipt of a page P_i, $\mathbb{N}_D$ sets its CSN as the maximum of its local CSN and that received with P_i such that at $\mathbb{N}_D$, $CSN(T_D) > CSN(T_S)$. This causal conflict ordering creates a global order per database page, where all transactions at $\mathbb{N}_S$ accessing P_i are ordered before all transactions at $\mathbb{N}_D$ that access P_i. We formally state this property as Theorem 8.3.6:

Theorem 8.3.6. *The transaction recovery and the conflict ordering protocol ensures that for every database page, conflicting transactions are replayed in the same order in which they committed.*

Migration State Recovery

Migration progress is logged to guarantee atomicity and consistency in the presence of failures. Migration safety is ensured by using rigorous recovery protocols. A failure of either $\mathbb{N}_S$ or $\mathbb{N}_D$ in dual mode or the finish mode requires coordinated recovery between the two nodes. We first discuss recovering from a failure during transition of migration modes and discuss recovery after failure in different migration modes.

Transitions of Migration Modes During migration, a transition from one state to another is logged. Except for the transition from the init mode to dual mode, which involves the query router metadata in addition to $\mathbb{N}_S$ and $\mathbb{N}_D$, all other transitions involve only $\mathbb{N}_S$ and $\mathbb{N}_D$. Such transitions occur through a one-phase handshake between $\mathbb{N}_S$ and $\mathbb{N}_D$. To transition between migration modes, $\mathbb{N}_S$ forces an entry logging the initiation of the transition and sends a message to $\mathbb{N}_D$. On receipt of the message, $\mathbb{N}_D$ moves to the next migration mode, forces a log entry for this change, and sends an acknowledgment to $\mathbb{N}_S$. Receipt of this acknowledgment completes this transition and $\mathbb{N}_S$ forces an entry logging the completion of the transition.

If $\mathbb{N}_S$ fails before sending the message to $\mathbb{N}_D$, the mode remains unchanged when $\mathbb{N}_S$ recovers, and $\mathbb{N}_S$ re-initiates the transition. If $\mathbb{N}_S$ fails after sending the message, then it knows about the message after it recovers and establishes contact with $\mathbb{N}_D$. Therefore, a state transition results in two messages and three writes to the log. Logging of messages at $\mathbb{N}_S$ and $\mathbb{N}_D$ provides message idempotence, i.e., detect and reject duplicate messages resulting from failure of $\mathbb{N}_S$ or $\mathbb{N}_D$, and guarantees safety with repeating failures.

Atomic Handover A transition from the init mode to dual mode involves three participants ($\mathbb{N}_S$, $\mathbb{N}_D$, and the query router metadata) that must together change the state atomically. That is, they must either all be in dual mode or all in init mode. A one-phase handshake is therefore not enough. We use the two-phase commit (2PC) [44] protocol which is a standard protocol for atomic commitment over multiple sites. Once $\mathbb{N}_D$ has acknowledged the initialization of $\mathbb{P}_M$, $\mathbb{N}_S$ initiates the transition and serves as the coordinator of 2PC. First, $\mathbb{N}_S$ sends a message to the router to direct all future transactions accessing $\mathbb{P}_M$ to $\mathbb{N}_D$, and a message to $\mathbb{N}_D$ to start accepting new transactions for $\mathbb{P}_M$ whose ownership is shared with $\mathbb{N}_S$. $\mathbb{N}_S$ forces an entry logging the initiation of this handover; though a force is not required if assuming presumed-abort [87]. On receipt of the messages, both $\mathbb{N}_D$ and the router log their messages and reply back to $\mathbb{N}_S$. Logging at the router enables it to recover from a failure independent of $\mathbb{N}_S$. Once $\mathbb{N}_S$ has received messages from both $\mathbb{N}_D$ and the router, it logs the successful handover in its own log which commits the change of state from init mode to dual mode, and sends acknowledgments to $\mathbb{N}_D$ and the router which then update their respective states. Atomicity of this handover process follows directly from the atomicity proof of 2PC [44]. This protocol also exhibits the blocking behavior of 2PC when $\mathbb{N}_S$ (the coordinator) fails. However, this blocking only affects $\mathbb{P}_M$ which is anyway unavailable as a result of $\mathbb{N}_S$'s failure.

Recovering Migration Progress The page ownership information is critical for migration progress as well as safety. A simple fault-tolerant design is to make this ownership information durable; any page (P_i) transferred from $\mathbb{N}_S$ is immediately flushed to the disk at $\mathbb{N}_D$. $\mathbb{N}_S$ also makes this transfer persistent, either by logging the transfer or by updating P_i's parent page in the index, and flushing it to the disk. This simple solution will guarantee resilience to failure but will introduce a lot of disk I/O which considerably increases migration cost and impacts other co-located tenants.

An optimized solution uses the semantics of the operation that resulted in P_i's on-demand migration. When P_i is migrated, $\mathbb{N}_S$ has a persistent (or at least recoverable) image of P_i. After the migration of P_i, if a committed transaction at $\mathbb{N}_D$ updated P_i, then the update will be in $\mathbb{N}_D$'s transaction log. Therefore, after a failure, $\mathbb{N}_D$ recovers P_i from its log and the persistent image of P_i that it can obtain from $\mathbb{N}_S$. The presence of a log entry accessing P_i at $\mathbb{N}_D$ implies that $\mathbb{N}_D$ owns P_i, thus preserving the ownership information after $\mathbb{N}_D$'s recovery.

In case P_i was migrated only for a read operation or if an update transaction at $\mathbb{N}_D$ did not commit, then this migration of P_i is not persistent at $\mathbb{N}_D$, though $\mathbb{N}_S$ has already migrated P_i's ownership. That is, $\mathbb{N}_D$ will not recover as the owner of P_i. Therefore, in case of $\mathbb{N}_D$'s failure, P_i can potentially become an *orphan page*, i.e., without an owner. However, synchronization between $\mathbb{N}_S$ and$\mathbb{N}_D$ ensures that such *orphan pages* are not left without an owner for indefinite periods. After $\mathbb{N}_D$'s recovery completes, it synchronizes its page ownership information with that of $\mathbb{N}_S$. Any orphan page P_i is detected during this synchronization, after which $\mathbb{N}_D$ assumes ownership of P_i and copies P_i's from the persistent image at $\mathbb{N}_S$.

Similarly, if $\mathbb{N}_S$ fails after migrating P_i, it recovers assuming it is the owner of P_i, even though the ownership was migrated to $\mathbb{N}_D$. However, this dual ownership of P_i is detected when $\mathbb{N}_S$ synchronizes its page ownership information with $\mathbb{N}_D$; $\mathbb{N}_S$ updates its ownership information after this synchronization completes.

Failure of both $\mathbb{N}_S$ and $\mathbb{N}_D$ immediately following P_i's transfer is equivalent to the failure of $\mathbb{N}_D$ without P_i making it to the disk at $\mathbb{N}_D$, and $\mathbb{N}_D$ becomes the owner of the orphan pages.

Logging the pages at $\mathbb{N}_D$ guarantees idempotence of page transfers, thus allowing migration to deal with repeated failures and prevent lost updates at $\mathbb{N}_D$.

Therefore, the optimized version of the page transfer protocol reduces the logging necessary during normal operation and hence considerably reduces the disk I/O during dual mode. In the finish mode, since pages are transferred in bulk, the pages transferred can be immediately flushed to the disk; the large number of pages per flush amortizes the disk I/O.

Since the transfer of pages to $\mathbb{N}_D$ does not force an immediate flush, after migration completes, $\mathbb{N}_D$ must ensure a flush before $\mathbb{P}_M$'s information can be purged at $\mathbb{N}_S$. This is achieved using a fuzzy checkpoint at $\mathbb{N}_D$ [15]. A fuzzy checkpoint is used by a DBMS during normal operation to reduce the recovery time after a failure. It causes minimal disruption to transaction processing, as a background thread scans through the database cache and flushes modified pages, while the database can continue to process updates. As part of the final state transition all transferred pages are marked dirty and $\mathbb{N}_D$ initiates a fuzzy checkpoint. After the checkpoint, $\mathbb{N}_D$ can independently recover and $\mathbb{N}_S$ can safely purge $\mathbb{P}_M$'s state. This recovery protocol guarantees that in the presence of a failure, migration recovers to a consistent point before the crash. Theorem 8.3.7 formalizes this recovery guarantee.

Theorem 8.3.7. ***Migration recovery:*** *At any instant during migration, its progress is recoverable, i.e., after transaction state recovery is complete, database page ownership information is restored to a consistent state and every page has exactly one owner.*

Failure and Availability A failure during migration results in partial or complete unavailability of $\mathbb{P}_M$.

- **Init Mode.** In the init mode, $\mathbb{N}_S$ is still the exclusive owner of $\mathbb{P}_M$.
  - $\mathbb{N}_S$ **fails:** $\mathbb{P}_M$ becomes unavailable and this state is equivalent to $\mathbb{N}_S$'s failure during normal operation. $\mathbb{N}_D$ can either abort migration or wait until $\mathbb{N}_S$ recovers and resumes migration. If $\mathbb{N}_D$ aborts migration, $\mathbb{N}_S$ detects this outcome after recovery and notifies the controller.
  - $\mathbb{N}_D$ **fails:** $\mathbb{N}_S$ has two options: it can unilaterally abort the migration, or it can continue processing new transactions until $\mathbb{N}_D$ recovers and resumes migration. If migration is aborted in the init mode, $\mathbb{N}_S$ notifies the controller which might select a new destination and re-initiate migration.
- **Dual Mode.** In the dual mode, $\mathbb{N}_S$ and $\mathbb{N}_D$ share ownership of $\mathbb{P}_M$. Failure of one of $\mathbb{N}_S$ or $\mathbb{N}_D$ does not render $\mathbb{P}_M$ completely unavailable.
  - $\mathbb{N}_S$ **fails:** $\mathbb{N}_D$ can only process transactions that access pages whose ownership was migrated to $\mathbb{N}_D$ before $\mathbb{N}_S$ failed. This is equivalent to a disk failing, making parts of the database unavailable.
  - $\mathbb{N}_D$ **fails:** $\mathbb{N}_S$ can only process transactions that do not access the pages that have already been migrated.
- **Finish Mode.** In the finish mode, $\mathbb{N}_D$ is the exclusive owner of $\mathbb{P}_M$.
  - $\mathbb{N}_S$ **fails:** Similar to $\mathbb{N}_S$'s failure in the dual mode, $\mathbb{N}_D$ can only process transactions that access pages whose ownership was migrated to $\mathbb{N}_D$ before $\mathbb{N}_S$ failed.
  - $\mathbb{N}_D$ **fails:** $\mathbb{P}_M$ becomes unavailable since $\mathbb{N}_D$ is now the exclusive owner of $\mathbb{P}_M$. This failure is equivalent to $\mathbb{N}_D$'s failure during normal operation.

8.3.3 Migration Safety and Liveness

Migration safety ensures correctness in the presence of a failure, while liveness ensures that ‘something good’ will eventually happen. We first establish formal definitions for safety and liveness, and then show how Zephyr guarantees these properties.

Definition 8.3.8. ***Safety*** *of migration requires the following conditions:* (*i*) Transactional isolation: *serializability is guaranteed for transactions executing during migration;* (*ii*) Transaction durability: *updates from committed transactions are never lost; and* (*iii*) Migration consistency: *a failure during migration does not leave the system’s state and data inconsistent.*

Definition 8.3.9. ***Liveness*** *of migration requires the following conditions to be met:* (*i*) Termination: *if $\mathbb{N}_S$ and $\mathbb{N}_D$ are not faulty and can communicate with each other for a sufficiently long period during migration, this process will terminate; and* (*ii*) Starvation Freedom: *in the presence of one or more failures, $\mathbb{P}_M$ will eventually have at least one node that can execute its transactions.*

Transaction correctness follows from Theorem 8.3.5. We now prove transaction durability and migration consistency.

Theorem 8.3.10. ***Transaction durability:*** *Changes made by a committed transaction are never lost, even in the presence of an arbitrary sequence of failures.*

Proof. The proof follows from the following two conditions: (*i*) during normal operation, transactions force their updates to the log before commit, making them durable; and (*ii*) on successful termination of migration, $\mathbb{N}_S$ purges its transaction log and the database image only after the fuzzy checkpoint at $\mathbb{N}_D$ completes, ensuring that changes at $\mathbb{N}_S$ and $\mathbb{N}_D$ during migration are durable. □

Theorem 8.3.11. ***Migration consistency:*** *In the presence of arbitrary or repeated failures, Zephyr ensures:* (*i*) *updates made to data pages are consistent even in the presence of failures;* (*ii*) *a failure does not leave a page P_i of $\mathbb{P}_M$ without an owner; and* (*iii*) *both $\mathbb{N}_S$ and $\mathbb{N}_D$ are in the same migration mode.*

The condition for exclusive page ownership along with Theorem 8.3.5 and 8.3.6 ensures that updates to the database pages are always consistent, both during normal operation and after a failure. Theorem 8.3.7 guarantees that no database page is without an owner, while the atomicity of the atomic handover and other state transition protocols discussed in Section 8.3.2 guarantee that both $\mathbb{N}_S$ and $\mathbb{N}_D$ are in the same migration mode. Theorem 8.3.5, 8.3.10, and 8.3.11 therefore guarantee migration safety.

Theorem 8.3.12. ***Migration termination:*** *If $\mathbb{N}_S$ and $\mathbb{N}_D$ are not faulty and can communicate for a long enough period, Zephyr guarantees progress and termination.*

Proof. Zephyr successfully terminates if: (*i*) the set of active transactions ($\mathbb{T}$) at $\mathbb{N}_S$ at the start of dual mode have completed, i.e., $\mathbb{T} = \phi$; and (*ii*) the persistent image of $\mathbb{P}_M$ is migrated to $\mathbb{N}_D$ and is recoverable. If $\mathbb{N}_S$ is not faulty in dual mode, all transactions in $\mathbb{T}$ will eventually complete, whether $\mathbb{N}_D$ has failed or not. If there is a failure of $\mathbb{N}_S$ at any point during migration, then after recovery it is guaranteed that $\mathbb{T} = \phi$. Therefore, the first condition is guaranteed to be satisfied eventually. After $\mathbb{T} = \phi$, if $\mathbb{N}_S$ and $\mathbb{N}_D$ can communicate long enough, all the pages of $\mathbb{P}_M$ at $\mathbb{N}_S$ will be migrated and recoverable at $\mathbb{N}_D$. □

Theorem 8.3.13. ***Starvation freedom:*** *Even after an arbitrary sequence of failures, there will be at least one node that can execute transactions on* $\mathbb{P}_M$.

The proof of Theorem 8.3.13 follows from Theorem 8.3.11 which ensures that $\mathbb{N}_S$ and $\mathbb{N}_D$ are in the same migration mode, and hence have a consistent view of $\mathbb{P}_M$'s ownership.

Theorem 8.3.12 and 8.3.13 together guarantee liveness. Zephyr guarantees safety in the presence of repeated failures or a network partition between $\mathbb{N}_S$ and $\mathbb{N}_D$, though progress is not guaranteed. Even though such failures are rare, proven guarantees in such scenarios improves the users' reliance on the system.

8.4 Optimizations and Extensions

We now discuss some extensions that relax some of the assumptions made to simplify our initial description of Zephyr.

8.4.1 Replicated Tenants

In our discussion so far, we assume that the destination of migration does not have any prior information about $\mathbb{P}_M$. Many production database installations however use some form of replication for fault-tolerance and availability. In such a scenario, $\mathbb{P}_M$ can be migrated to a node which already has its replica. Since most DBMS implementations use lazy replication techniques to circumvent the high cost of synchronous replication [15], replicas often lag behind the master. Zephyr can be adapted to leverage this form of replication. Since $\mathbb{N}_D$ already has a replica, there is no need for init mode. When $\mathbb{N}_S$ is notified to initiate migration, it executes the atomic handover protocol to enter dual mode. Since $\mathbb{N}_D$'s copy of the database is potentially stale, when a transaction T_{Di} accesses a page P_i, similar to the original design, $\mathbb{N}_D$ synchronizes with $\mathbb{N}_S$ to transfer ownership. $\mathbb{N}_D$ sends the sequence number associated with its version of P_i to determine if it has the latest version of P_i; P_i is transferred only if $\mathbb{N}_D$'s version is stale. Furthermore, in finish mode, $\mathbb{N}_S$ only needs to send a small number of pages that were not replicated to $\mathbb{N}_D$ due to a lag in replication. Replication can therefore considerably improve the performance of Zephyr.

8.4.2 Sharded Tenants

Our initial description assumes that a tenant is small and is served from a single node, i.e., a single partition tenant. However, Zephyr can also handle a large tenant that is sharded across multiple nodes, primarily due to the fact that $\mathbb{N}_S$ completes the execution of all transactions that were active when migration was initiated. Let $\mathbb{P}_M$ consist of partitions $\mathbb{P}_{M1}, \ldots, \mathbb{P}_{Mp}$ and assume that we are migrating $\mathbb{P}_{Mi}$ from $\mathbb{N}_S$ to $\mathbb{N}_D$. Transactions accessing only $\mathbb{P}_{Mi}$ are handled similar to the case of a single partition tenant. Let T_i be a multi-partition transaction where $\mathbb{P}_{Mi}$ is a participant. If T_i was active at the start of migration, then $\mathbb{N}_S$ is the node that executes T_i, and $\mathbb{P}_{Mi}$ will transition to finish mode only when all such T_i's have completed. On the other hand, if T_i started after $\mathbb{P}_{Mi}$ transitioned to dual mode, then $\mathbb{N}_D$ is the node executing T_i. At any given node, T_i is executed in the same way as in a single partition tenant.

8.4.3 Data Sharing in Dual Mode

In Dual Mode, both $\mathbb{N}_S$ and $\mathbb{N}_D$ are executing update transactions on $\mathbb{P}_M$. This design is reminiscent of data sharing systems [23], the difference being that our design does not use a shared lock manager. However, our design can be augmented to use a shared lock manager to support a larger set of operations during migration, including arbitrary updates and minimizing transaction aborts at $\mathbb{N}_S$.

In the modified design, we replace the concept of page ownership with page level locking, allowing the locks to be shared when both $\mathbb{N}_S$ and $\mathbb{N}_D$ are reading a page. Every node in the system has a **Local Lock Manager (LLM)** and a **Global Lock Manager (GLM)**. The LLM is responsible for the local locking of pages while the GLM is responsible for arbitrating locks for remote pages. In all migration modes except dual mode, locks are local and hence serviced by the LLM. However, in dual mode, $\mathbb{N}_S$ and $\mathbb{N}_D$ must synchronize through the GLMs. The only change needed is in the page ownership transfer; the rest of the algorithm remains unchanged. Note that scalability limitations of a shared lock manager are not significant since any instance of the lock manager is shared by only two nodes. We now describe how this extended design can remove some limitations of the original design. Details have been omitted due to space constraints.

In the original design of Zephyr, when a transaction T_{Di} requests access for a page P_i, $\mathbb{N}_D$ transfers ownership from $\mathbb{N}_S$. Therefore, future accesses to P_i (even reads) must fail to ensure serializable isolation. In this extended design, if $\mathbb{N}_D$ only needs a shared lock on P_i to service reads, then $\mathbb{N}_S$ can also continue processing reads that access P_i. Furthermore, even if $\mathbb{N}_D$ had acquired an exclusive lock, $\mathbb{N}_S$ can request a lock to $\mathbb{N}_D$'s GLM for the desired lock on P_i. This allows processing transactions at $\mathbb{N}_S$ that access a migrated page; the request to migrate the page back to $\mathbb{N}_S$ might be blocked in case it is locked at $\mathbb{N}_D$. The trade-off associated with this flexibility is the cost of additional synchronization between $\mathbb{N}_S$ and $\mathbb{N}_D$ to arbitrate shared locks, and the higher network overhead arising from copying P_i multiple times, while in the initial design, P_i was migrated exactly once.

The original design made the index structure at both $\mathbb{N}_S$ and $\mathbb{N}_D$ immutable during migration and did not allow insertions or deletions that required a change in the index structure. The shared lock manager in the modified design circumvents this limitation by sharing locks at the index level as well, such that normal index traversal will use shared intention locks while an update to the index will acquire an exclusive lock on the index nodes being updated.

Zephyr, adapted to the data sharing architecture, allows more flexibility by allowing arbitrary updates and minimizing transactions or operations aborted due to migration. The effect on the correctness is straightforward. Since page ownership can be transferred back to $\mathbb{N}_S$, Lemma 8.3.3 does not hold any longer. However, Theorem 8.3.5 still holds since page level locking is done in a two phase manner using the shared lock managers, which ensures that a cycle in the serialization graph is impossible. Similarly, the proof for Lemma 8.3.1 has to be augmented with the case for index changes. However, since a transaction inserting an item (T_2 in Lemma 8.3.1) needs to acquire an exclusive on the index page being modified, it will be blocked by the predicate lock acquired by the transaction with the predicate (T_1 in Lemma 8.3.1) on the index pages. Therefore, transactional correctness is still satisfied in the modified design; the other correctness arguments remain unchanged.

8.5 Implementation Details

Our prototype implementation of Zephyr extends an open source OLTP database H2 [46]. H2 is a lightweight relational database with a small footprint. H2 is built entirely in Java and supports both embedded and server mode operation. Though primarily designed for embedded operation, one of the major applications of H2 is as a replacement of commercial RDBMS servers for development and testing. It supports a standard SQL/JDBC API, serializable and read-committed isolation levels [12], tree indices, a relational data model, and referential integrity constraints [15].

H2's architecture resembles the shared process multitenancy model where an H2 instance can have multiple independent databases with different schemas. Each database maintains its independent database cache, transaction manager, transaction log, and recovery manager. In H2, a database is stored as a file on disk which is internally organized as a collection of fixed size database pages. The first four pages store the database's metadata. The data definitions and user authentication information is stored as a metadata table (called `INFORMATION_SCHEMA`) which is part of the database. Every table in H2 is organized as a tree index. If a table is defined with a primary key which is of type integer or real number, then the primary key index stores data for the table. If the primary key is of another type (such as `VARCHAR`) or was not specified at table creation, then the table's data is stored in a tree index whose key is auto-generated by the system. A table can have multiple indices which are maintained separate from the primary key index. The fourth page in the database file stores a pointer to the root of the `INFORMATION_SCHEMA` table, which in turn stores pointers to the other user tables. H2 supports classic multi-step transactions with serializable and read-committed isolation.

We use SQL Router[3], an open source package, to implement the query router. It is a JDBC wrapper that transparently migrates JDBC connections from $\mathbb{N}_S$ to $\mathbb{N}_D$. This SQL router runs a server listener that is notified when $\mathbb{P}_M$'s location changes. When migration is initiated, $\mathbb{N}_S$ spawns a migration thread T. In *init mode*, T transfers the database metadata pages, the entire `INFORMATION_SCHEMA` table of H2, and the internal nodes of the indices. Conceptually, this wireframe can be constructed by traversing the index trees to determine the internal index nodes. However, this might incur a large number of random disk accesses for infrequently accessed parts of the index, which increases migration overhead. We therefore use an optimization in the implementation where T sequentially scans the database file and transfers only the internal nodes of the indices. When processing a database index page, it synchronizes with any concurrent transactions and obtains the latest version from the cache, if needed. Since the index structure is frozen during migration, this scan uses shared locking, allowing other update transactions to proceed. T notifies $\mathbb{N}_D$ of the number of pages skipped, which is used to update page ownership information at $\mathbb{N}_D$.

In dual mode, $\mathbb{N}_D$ pulls pages from $\mathbb{N}_S$ on-demand while $\mathbb{N}_S$ continues transaction execution. Before a page is migrated, $\mathbb{N}_S$ obtains an exclusive lock on the page, updates the ownership mapping, and then sends it to $\mathbb{N}_D$. This ensures that the page is migrated only if it is not locked by any concurrent transaction. In *finish mode*, $\mathbb{N}_S$ pushes all remaining pages that were not migrated in dual mode, while serving any page fetch request from $\mathbb{N}_D$; pages transferred twice as a result of both the push from $\mathbb{N}_S$ and pull from $\mathbb{N}_D$ are detected at $\mathbb{N}_D$ and duplicate pages are rejected. Since $\mathbb{N}_S$ does not execute any transactions in finish mode, this push does not require any synchronization at $\mathbb{N}_S$.

[3] `http://www.continuent.com/community/tungsten-sql-router`

Parameter	**Default value**
Transaction size	10 operations
Read/Write distribution	80% reads, 15% updates, 5% inserts
Database size	250 MB
Transaction load	50 transactions per second (TPS)
Cache size	32 MB
Database page size	16 KB

Table 8.1: Default values for parameters used in Zephyr's evaluation.

8.6 Experimental Evaluation

We now present a thorough experimental evaluation of Zephyr for live database migration using our prototype implementation. We compare Zephyr with the off-the-shelf **stop and copy** technique that stops the database at $\mathbb{N}_S$, flushes all changes, copies over the persistent data, and restarts the database at $\mathbb{N}_D$. We measure the migration cost as the number of failed client interactions (or failed requests), the amount of data transferred during migration, and the impact on transaction latency during and after migration.

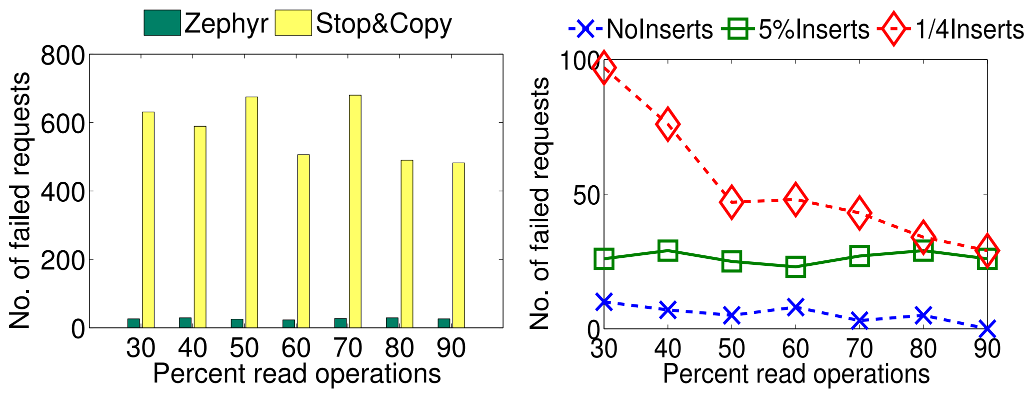
8.6.1 Experimental Setup

Our evaluation uses two server nodes that run the database instances and a separate set of client machines that generate load on the database. Each server node has a 2.4 GHz Intel Core 2 Quad processor, 8 GB RAM, a 7200 RPM SATA hard drive with 32 MB Cache, and runs a 64-bit Ubuntu Server Edition with Java 1.6. The nodes are connected via a gigabit switch. Workload is generated from a different set of client machines. Since migration only involves $\mathbb{N}_S$ and $\mathbb{N}_D$, our evaluation focusses only on these two nodes and is oblivious to other nodes.

8.6.2 Methodology

In our evaluation, we use the modified Yahoo! cloud serving benchmark (YCSB) presented in Chapter 4 (see Section 4.5.2). In our experiments, we consider tenant applications with small databases, where every tenant is assigned a partition. We therefore use the terms tenant and partition interchangeably.

The workload emulates multiple user sessions where a user connects to a tenant's database, executes hundred transactions and then disconnects. A workload consists of



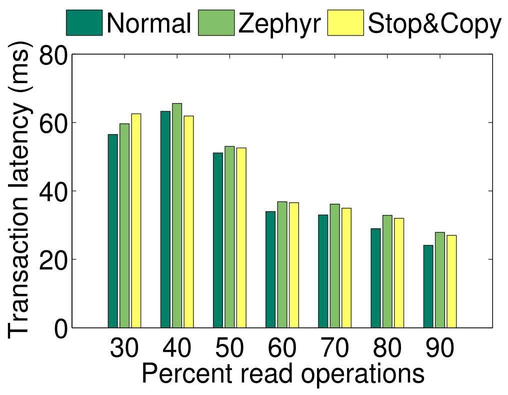
(a) Failed requests (b) Different insert ratios

Figure 8.5: Impact of the distribution of reads, updates, and inserts on the number of failed requests. We also vary the insert ratios–5% inserts correspond to a fixed percentage of inserts, while $1/4$ inserts correspond to a distribution where a fourth of the write operations are inserts. The benchmark executes 60,000 operations.

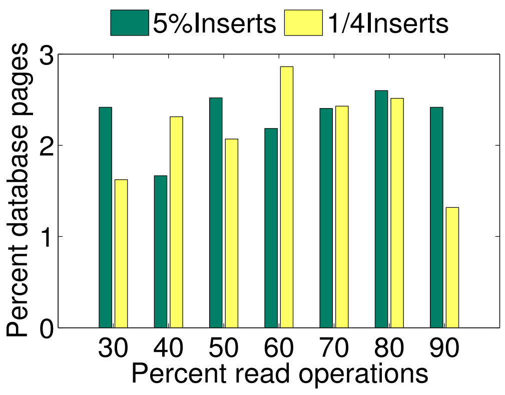
sixty such sessions, i.e., a total of $6,000$ transactions. We vary different YCSB parameters to cover a wide spectrum of workloads. These parameters include the percentage of read operations in a transaction, number of operations in a transaction (or **transaction size**), a tenant's database size, load offered on a partition, cache size, and the page size used by a database. We use Zipfian distribution to select the data items accessed with the co-efficient set to 1.0. In an experiment, we vary one of these parameters while using the default values for the rest of the parameters. The default values of these parameters are provided in Table 8.1, which are representative of medium sized tenants [88, 90].

8.6.3 Evaluating Migration Cost

Our first experiment analyzes the impact on migration cost when varying the percentage read operations in a transaction. Figure 8.5(a) plots the number of client requests that failed during migration; clients continue issuing requests on $\mathbb{P}_M$ even during migration. A client thread sequentially issues the operations of a transaction. All client requests are well-formed, and any error reported by the database server after a request has been issued account for a failed request. As is evident from Figure 8.5(a), the number of failed requests in Zephyr is one to two orders of magnitude lower than that of stop and copy. Two reasons contribute to more failed requests in stop and copy: (i) aborts of all transactions active at the start of migration, and (ii) aborts of all new transactions that access the tenant when it is unavailable during migration. Zephyr does



(a) Average latency



(b) Pages pulled in dual mode

Figure 8.6: Impact of the distribution of reads, updates, and inserts on the average transaction latency and the percentage of database pages pulled in dual mode.

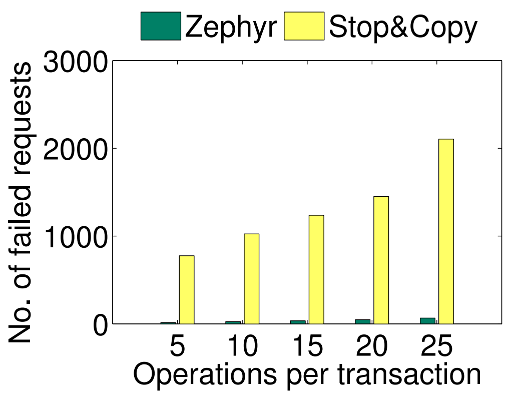
not incur any unavailability; requests fail only when they result in a change to the index structure during migration.

Figure 8.5(b) plots the number of failed requests for Zephyr when using workloads with different insert ratios. Zephyr results in only a few tens of failed requests when the workload does not have a high percentage of inserts, even for cases with a high update proportion. As the workload becomes predominantly read-only, the probability of a request resulting in a change in the index structure decreases. This results in a decrease in the number of failed requests in Zephyr. Stop and copy also results in fewer failed requests for higher values of read percentages, due to the smaller unavailability window resulting from fewer updates that need to be flushed before migration.

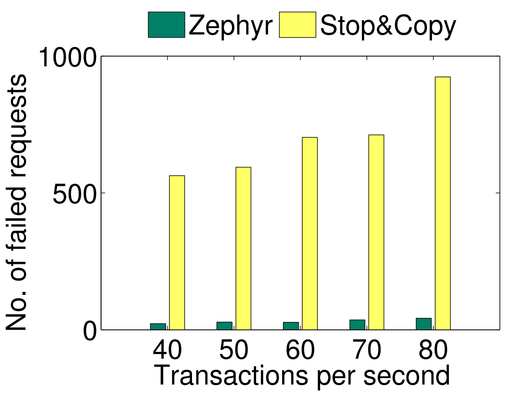
Figure 8.6(a) plots the average transaction latency observed by a client during normal operation (i.e., when no migration is performed) and with a migration occurring midway. We report latency averaged over all the $6,000$ transactions in the workload. We only report latency of committed transactions; aborted transactions are ignored. Compared to normal operation, the increased latency in stop and copy results from the cost of warming up the cache at $\mathbb{N}_D$ and the cost of clients re-establishing the database connections after migration. In addition to the aforementioned costs, Zephyr fetches pages from $\mathbb{N}_S$ on-demand during migration; the page can be fetched from $\mathbb{N}_S$'s cache or from its disk. This adds latency overhead in Zephyr compared to stop and copy.

Figure 8.6(b) shows the percentage of database pages pulled during dual mode. Since dual mode runs for a very short period, only a small fraction of pages are pulled on demand. Therefore, Zephyr incurs low data transfer overhead.

In our experiments, stop and copy took 3 to 8 seconds to migrate a tenant. Since all transactions in the workload have at least one update operation, when using stop



(a) Transaction size

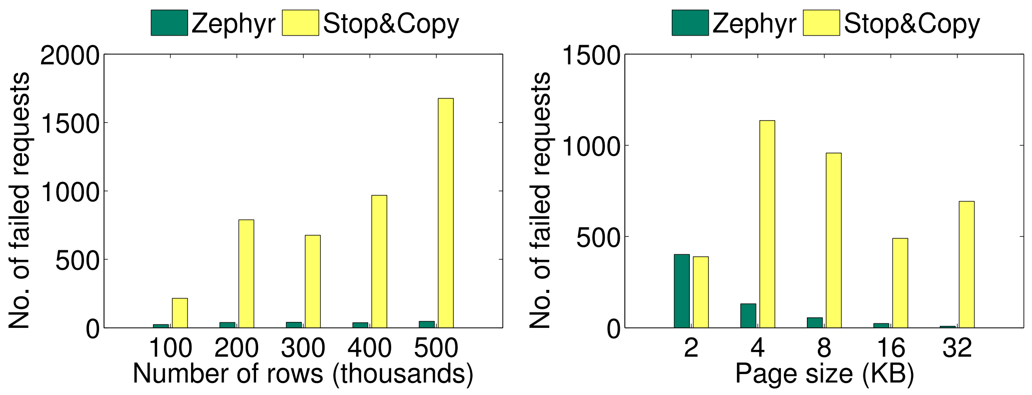


(b) Load

Figure 8.7: Impact of varying the transaction size and load on number of failed transactions.

and copy, all transactions issued during migration are aborted. On the other hand, even though Zephyr requires about 10 to 18 seconds to migrate the tenant, there is no downtime. As a result, the tenants observe few failed requests. Zephyr also incurs minimal messaging overhead beyond that needed to migrate the persistent database image. Every page transferred is preceded by its unique identifier; a pull request in dual mode requires one round trip of messaging to fetch the page from $\mathbb{N}_S$. Stop and copy only requires the persistent data for $\mathbb{P}_M$ to be migrated and does not incur any additional data transfer/messaging overhead.

We now evaluate the impact of transaction sizes and load (see Figure 8.7). Varying the transaction size implies varying the number of operations in a transaction. Since the load is kept constant at 50 TPS, a higher number of operations per transaction implies more operations issued per unit time. Varying the load implies varying the number of transactions issued. Therefore, higher load also implies more operations issued per unit time. Since the percentage of updates is kept constant, more operations result in more updates. For stop and copy, more updates result in more data to be flushed before migration. This results in a longer unavailability window which in turn results in more operations failing. On the other hand, for Zephyr, more updates imply a higher probability of changes to the index structure during migration, resulting in more failed requests. However, the rate of increase in failed requests is lower in Zephyr compared to stop and copy. This is evident from the slope of an approximate linear fit of the data points in Figure 8.7; the linear fit for Zephyr has a considerably smaller slope than that for stop and copy. The effect on transaction latency is similar and hence is omitted. We also varied the cache size allocated to the tenants, but, the impact on service interruption was not significant. Even though a large cache size potentially results in more changes



(a) Database size (b) Database page size

Figure 8.8: Impact of the database page size and database size on number of failed requests.

to be flushed to the disk, the Zipfian access distribution coupled with a high percentage of read operations results in very few changed objects in the cache.

Figure 8.8(a) plots the impact of database size on failed requests. In this experiment, we increase the database size up to 500K rows (about 1.3 GB). As the database size increases, more time is needed to copy its persistent data, resulting in a longer unavailability window for stop and copy. On the other hand, for Zephyr, a larger database implies a longer finish mode. However, since Zephyr does not result in any unavailability, the database size has almost no impact on the number of failed requests. Therefore, Zephyr is more robust for larger databases when compared to stop and copy.

Figure 8.8(b) shows an interesting interplay between the database page size on the number of failed requests. As the database page size increases, the number of failed requests decreases considerably for Zephyr, while that of stop and copy is almost unaffected. When the page size is small, each page contains only a few rows. For instance, in our setting, each row is close to a kilobyte, and a 2K page only holds two rows. As a result, a majority of inserts result in structural changes to the index, which result in a lot of these inserts failing during migration. If we consider the experiment with 2K page size, more than 95% of the failed requests were due to inserts. However, as the page size increases, the leaf pages have more unused capacity. Therefore, fewer inserts result in a change to the index structure. Since stop and copy is oblivious to the page size and transfers the raw bytes of the database file, its performance is almost unaffected by a change in the page size. However, when the page size is increased beyond the block size of the underlying file system, reading a page from the disk becomes more expensive, resulting in an increase in the transaction latency when the page size is larger than the file system block size.

In summary, Zephyr results in minimal service interruption. In a cloud platform, high availability is extremely critical for customer satisfaction, thus making Zephyr more attractive. In spite of Zephyr not allowing changes to the index structure during migration, it results in very few requests failing. A significant failure rate was observed only with a high ratio of row-size to page size. Zephyr is therefore more robust to variances in read-write ratios, database sizes, and transaction sizes when compared to stop and copy, thus making it suitable for a variety of workloads and applications.

8.7 Summary

In this chapter, we presented Zephyr, an efficient technique to migrate a live database (or partition) in a shared nothing architecture. Zephyr uses a combination of on-demand pull and asynchronous push to migrate a tenant with minimal service interruption. Using lightweight synchronization, Zephyr minimizes the number of failed operations during migration, while reducing the amount of data transferred during migration. We presented a detailed analysis of the guarantees provided and proved the safety and liveness of Zephyr. Our technique relies on generic structures such as lock managers, standard B+ tree indices, and minimal changes to write-ahead logging, thus making it suitable for most standard database engines with minimal changes to the existing code base. Our implementation in a lightweight open source RDBMS showed that Zephyr allows lightweight migration of a live database partition with minimal service interruption, thus allowing migration to be effectively used for elastic load balancing.

Zephyr is the first end-to-end solution for live database migration in a shared nothing architecture. We therefore focussed on a simple design to demonstrate feasibility and guarantee correctness. Section 8.4 discussed some optimization and extensions. Various other extensions are also possible. For instance, to handle inserts resulting in page splits in an index, Zephyr can be augmented to store the newly inserted data items in *overflow buckets* similar to B-link trees [64]. As with Albatross, it will also be useful to *predict* the migration cost of Zephyr so that the system controller can effectively use migration without violating the SLAs. These extensions, in addition to a detailed discussion of the ones presented in Section 8.4, are interesting directions for future work.

Part III

Concluding Remarks

Chapter 9

Conclusion and Future Directions

"Reasoning draws a conclusion, but does not make the conclusion certain, unless the mind discovers it by the path of experience."

– Roger Bacon.

9.1 Concluding Discussion

Over the past few years, cloud computing has emerged as a multi-billion dollar industry and as a successful paradigm for web application deployment. Irrespective of the cloud provider or the cloud abstraction, data is central to applications deployed in the cloud. Since DBMSs store and serve an application's critical data, they form a mission critical component in the cloud software stack.

DBMSs deployed in a cloud infrastructure and supporting diverse applications face unique challenges. The overarching goal of this dissertation was to enable DBMSs to scale-out while efficiently supporting transactional semantics and being elastic without introducing high performance overhead. On one hand, the ability to scale-out using clusters of commodity servers allows the DBMSs to leverage from the economies of scale, and the ability to efficiently support transactional semantics simplifies application design. On the other hand, the ability to dynamically scale-up and scale-down the number of nodes in a live DBMS allows the system to consolidate to fewer nodes during periods of low load and to add nodes when the load increases. This elastic scaling leverages the underlying pay-per-use cloud infrastructure to minimize the system's operating cost and ensures good performance. This dissertation makes fundamental contributions in the two thrust areas of scale-out transaction processing and lightweight elasticity. These advances are critical to the design of DBMSs for cloud computing infrastructure and significantly advances the state-of-the-art in that field.

In the area of scale-out transaction processing, we proposed the design and implementation of two systems that guarantee transactional access to database partitions where the partitions can be statically or dynamically defined. The key insight for both designs was to co-locate data frequently accessed data items within a database partition and limit transactions to access only a single partition, thus allowing efficient transaction execution, high scalability, and high availability.

We proposed ElasTraS, an elastically scalable transaction processing system to support applications whose access patterns allow the database to be partitioned statically. ElasTraS operates at the granularity of database partitions which are the units of assignment, load balancing, and transactional access. ElasTraS can effectively serve large numbers of small tenants and scale-out large tenants using schema level partitioning. ElasTraS is one of the first systems to allow efficient scale-out transaction processing while leveraging schema-level partitioning to support rich transactional semantics.

To serve applications whose access patterns evolve rapidly, we proposed the Key Group abstraction that allows applications to dynamically specify the group of data items on which it wants transactional access. We proposed the Key Grouping protocol to dynamically co-locate the read/write access (or ownership) of the data items forming a Key Group. The protocol ensures safe-ownership transfer within an operational system even in the presence of node or message failures. Ours is the first approach to allow a lightweight dynamic re-organization of ownership of the data items to provide the benefits of co-location even when the access patterns can not be statically partitioned.

In the area of lightweight elasticity in DBMSs, we proposed the design and implementation of two techniques for live database migration in the decoupled storage and shared nothing database architectures. The key insight for both techniques was to leverage the semantics of the DBMS internals to migrate a database partition with minimal disruption and performance overhead while ensuring transactional guarantees and correctness even in the presence of failures during migration. Our techniques are the first published end-to-end solutions for live database migration for elastic load balancing.

We proposed Albatross, a lightweight live migration technique for decoupled storage database architectures where the persistent data is stored in network-addressable storage and does not need migration. Albatross focusses on migrating the database cache and the state of active transactions to allow the partition being migrated to start warm at the destination.

We proposed Zephyr, a lightweight live migration technique for shared nothing database architectures where the persistent data is stored on disks locally attached to the database servers and hence must be migrated to the destination. Zephyr uses a combination of on-demand pull and asynchronous push to migrate a database partition with minimal service interruption.

9.2 Future Directions

The continued growth of data sizes, advent of novel applications, and evolution of the infrastructure ensures that the area of data management in the cloud has many interesting research challenges. While some of these future research directions are direct extensions of the techniques presented in this dissertation, others are more radical.

Access driven database partitioning techniques rely on the application's access patterns to partition data to avoid distributed transactions. However, as the application's access patterns change, it might need re-partitioning. Most web-applications strive for high availability. Traditional approaches rely on long unavailability windows to re-partition the database and are therefore not amenable to support regular re-partitioning. The challenge is to re-partition the database in a live system while minimizing service interruption. We envision two sub-problems towards this goal: techniques to incrementally determine the partitions based on changes in access patterns, and techniques to dynamically re-organize data in a live system without any downtime. Mining the transactions' access logs to discover access patterns, and extensions to the Key Group abstraction and the Key Grouping protocol are possible directions towards this goal.

In traditional enterprise settings, transaction processing and data analysis systems were typically managed as separate systems. The rationale behind this separation was that OLTP and analysis workloads have very different characteristics and requirements. Therefore, in terms of performance, it is prudent to separate the two types of systems [79]. However, the growing need for *real-time analysis* and the costs involved in managing two different systems have resulted in the compelling need for the convergence of the transaction processing and data analysis systems, especially in cloud infrastructures. In this dissertation, we focussed on the design of OLTP systems and presented the design principles and architectures for such systems. One major challenge in the design of these hybrid systems is to find the suitable design principles and architectures that will allow scale-out, elasticity, and augmented functionality. Existing approaches, such as Agrawal et al. [6], Nishimura et al. [71], and Cao et al. [21], present interesting trade-offs. A thorough analysis of the design space and the candidate systems, similar to those presented in Chapter 2, is essential in distilling the design principles the on-line transaction and analytical processing (**OLTAP**) systems.

Administering large scale database systems is expensive and labor intensive. Automatic administration of large DBMSs minimizes the need for human intervention for resource orchestration. The responsibilities of such a self-managing controller include monitoring the behavior and performance of the system, elastic scaling and load balancing based on dynamic usage patterns, modeling behavior to forecast workload spikes and take pro-active measures to handle such spikes. The design of a self-managing system controller for such large scale systems is an important area of future work. The goal is to ensure that the performance guarantees are met while ensuring effective resource utilization. As the scale of such systems increases, there is a super-linear growth in complexity of the problem, so the challenge is to make this problem tractable while ensuring competitive bounds. Leveraging machine learning techniques for the controller's design is a possible direction for future exploration [1, 38].

The current cloud infrastructure consists of a static collection of powerful data centers (or cores). This model misses out on the substantial computing power that resides outside the data centers. We envision a *dynamic cloud* [4] that will be formed of the static cloud that forms the nucleus of the infrastructure and a collection of cores that dynamically join the cloud from time to time. Such an infrastructure presents challenges beyond the current generation of cloud infrastructures. Examples of some challenges are: how to provide a consistent and uniform namespace spanning the dynamic collection of cloud cores, what are the practical consistency models and abstractions for such large scale dynamic environments, how to efficiently integrate surplus capacity as and when they become available, how to effectively migrate load and data and efficiently replicate state across the cores, and how to monitor and model such large scale systems. Extending the designs of elastic, self-managing, and scalable systems to this dynamic cloud infrastructure spanning larger scale operations, higher network latency, and lower network bandwidth is a worthwhile direction of future work.

Bibliography

[1] D. Agrawal, A. E. Abbadi, S. Das, and A. J. Elmore. Database Scalability, Elasticity, and Autonomy in the Cloud - (Extended Abstract). In *DASFAA (1)*, pages 2–15, 2011.

[2] D. Agrawal, S. Das, and A. E. Abbadi. Big data and cloud computing: New wine or just new bottles? *PVLDB*, 3(2):1647–1648, 2010.

[3] D. Agrawal, S. Das, and A. E. Abbadi. Big data and cloud computing: current state and future opportunities. In *EDBT*, pages 530–533, 2011.

[4] D. Agrawal, S. Das, and A. El Abbadi. From a Virtualized Computing Nucleus to a Cloud Computing Universe: A Case for Dynamic Clouds. Technical Report 2011-03, CS, UCSB, 2011.

[5] D. Agrawal, A. El Abbadi, S. Antony, and S. Das. Data Management Challenges in Cloud Computing Infrastructures. In *DNIS*, pages 1–10, 2010.

[6] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous view maintenance for VLSD databases. In *SIGMOD Conference*, pages 179–192, 2009.

[7] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, pages 159–174, 2007.

[8] Google App Engine SLA. `http://code.google.com/appengine/sla.html`, 2011. Retreived: October 2011.

[9] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *SIGMOD Conference*, pages 1195–1206, 2008.

[10] F. R. Bach and M. I. Jordan. Kernel independent component analysis. *J. Mach. Learn. Res.*, 3:1–48, 2003.

[11] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, pages 223–234, 2011.

[12] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD Conference*, pages 1–10, 1995.

[13] P. Bernstein, C. Reid, and S. Das. Hyder - A Transactional Record Manager for Shared Flash. In *CIDR*, pages 9–20, 2011.

[14] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manner, L. Novik, and T. Talius. Adapting Microsoft SQL Server for Cloud Computing. In *ICDE*, pages 1255–1263, 2011.

[15] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. MK Publishers Inc., second edition, 2009.

[16] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5:47–76, January 1987.

[17] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *VEE*, pages 169–179, 2007.

[18] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD Conference*, pages 251–264, 2008.

[19] E. A. Brewer. Towards robust distributed systems (Invited Talk). In *PODC*, page 7, 2000.

[20] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI*, pages 335–350, 2006.

[21] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. ES^2: A cloud data storage system for supporting both OLTP and OLAP. In *ICDE*, pages 291–302, 2011.

[22] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, pages 398–407, 2007.

[23] S. Chandrasekaran and R. Bamford. Shared cache - the future of parallel databases. In *ICDE*, pages 840–850, 2003.

[24] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218, 2006.

[25] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, pages 273–286, 2005.

[26] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[27] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM SoCC*, pages 143–154, 2010.

[28] C. Curino, E. Jones, R. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational Cloud: A Database Service for the Cloud. In *CIDR*, pages 235–240, 2011.

[29] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD Conference*, pages 313–324, 2011.

[30] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[31] S. Das, S. Agarwal, D. Agrawal, and A. El Abbadi. ElasTraS: An Elastic, Scalable, and Self Managing Transactional Database for the Cloud. Technical Report 2010-04, CS, UCSB, 2010.

[32] S. Das, D. Agrawal, and A. El Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. In *USENIX HotCloud*, 2009.

[33] S. Das, D. Agrawal, and A. El Abbadi. G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud. In *ACM SoCC*, pages 163–174, 2010.

[34] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration. *PVLDB*, 4(8):494–505, May 2011.

[35] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[36] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[37] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *VLDB*, pages 228–237, 1986.

[38] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Towards an Elastic and Autonomic Multitenant Database. In *NetDB*, 2011.

[39] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *SIGMOD Conference*, pages 301–312, 2011.

[40] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[41] D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *SIGMOD Record*, 38(1):43–48, 2009.

[42] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In *VLDB*, pages 209–219, 1986.

[43] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, pages 29–43, 2003.

[44] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[45] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In *VLDB*, pages 428–451, 1975.

[46] H2 Database Engine. `http://www.h2database.com/html/main.html`, 2011. Retreived: October 2011.

[47] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

[48] J. Hamilton. I love eventual consistency but... `http://bit.ly/hamilton-eventual`, April 2010. Retreived: October 2011.

[49] HBase: Bigtable-like structured storage for Hadoop HDFS. `http://hbase.apache.org/`, 2011. Retreived: October 2011.

[50] HDFS: A distributed file system that provides high throughput access to application data. `http://hadoop.apache.org/hdfs/`, 2011. Retreived: October 2011.

[51] P. Helland. Life beyond Distributed Transactions: An Apostate's Opinion. In *CIDR*, pages 132–141, 2007.

[52] A. R. Hickey. Cloud Computing Services Market To Near $150 Billion In 2014. `http://bit.ly/cloud_computing_value`, June 2010. Retreived: October 2011.

[53] A. Hirsch. Cool Facebook Application Game – Scrabulous – Facebook's Scrabble. `http://bit.ly/scrabulous`, 2007. Retreived: October 2011.

[54] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, 2010.

[55] M. Indelicato. Scalability Strategies Primer: Database Sharding. `http://bit.ly/sharding_primer`, December 2008. Retreived: October 2011.

[56] D. Jacobs and S. Aulbach. Ruminations on multi-tenant databases. In *BTW*, pages 514–521, 2007.

[57] J. W. Josten, C. Mohan, I. Narang, and J. Z. Teng. DB2's use of the coupling facility for data sharing. *IBM Syst. J.*, 36:327–351, April 1997.

[58] F. P. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, pages 245 – 256, 2011.

[59] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, 1997.

[60] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. *PVLDB*, 2(1):253–264, 2009.

[61] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[62] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[63] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[64] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6:650–670, December 1981.

[65] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, pages 123–133, 2011.

[66] B. G. Lindsay, L. M. Haas, C. Mohan, P. F. Wilms, and R. A. Yost. Computation and communication in R*: A distributed database manager. *ACM Trans. Comput. Syst.*, 2(1):24–38, 1984.

[67] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu. Live migration of virtual machine based on full system trace and replay. In *HPDC*, pages 101–110, 2009.

[68] D. Lomet, R. Anderson, T. K. Rengarajan, and P. Spiro. How the Rdb/VMS Data Sharing System Became Fast. Technical Report CRL 92/4, Digital Equipment Corporation, May 1992. `http://www.hpl.hp.com/techreports/Compaq-DEC/CRL-92-4.pdf`.

[69] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1):94–162, 1992.

[70] S. Neuvonen, A. Wolski, M. manner, and V. Raatikka. Telecommunication application transaction processing (tatp) benchmark description 1.0. `http://tatpbenchmark.sourceforge.net/TATP_Description.pdf`, March 2009. Retreived: October 2011.

[71] S. Nishimura, S. Das, D. Agrawal, and A. El Abbadi. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *MDM*, pages 7–16, 2011.

[72] D. Obasanjo. When databases lie: Consistency vs. availability in distributed systems. `http://bit.ly/obasanjo_CAP`, October 2009. Retreived: October 2011.

[73] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1991.

[74] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[75] L. Rao. One database to rule the cloud: Salesforce debuts database.com for the enterprise. `http://tcrn.ch/Database_Com`, December 2010. Retreived: October 2011.

[76] R. Rawson and J. Gray. HBase at Hadoop World NYC. `http://bit.ly/HBase_HWNYC09`, 2009. Retreived: October 2011.

[77] J. B. Rothnie Jr., P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. L. Reeve, D. W. Shipman, and E. Wong. Introduction to a System for Distributed Databases (SDD-1). *ACM Trans. Database Syst.*, 5(1):1–17, 1980.

[78] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD Conference*, pages 953–966, 2008.

[79] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose time has come and gone. In *ICDE*, pages 2–11, 2005.

[80] D. Tankel. Scalability of the Hadoop Distributed File System. `http://yhoo.it/HDFS_Perf`, May 2010. Retreived: October 2011.

[81] The Transaction Processing Performance Council. TPC-C benchmark (Version 5.11). `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`, February 2010. Retreived: October 2011.

[82] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the v-system. In *SOSP*, pages 2–12, 1985.

[83] H. T. Vo, C. Chen, and B. C. Ooi. Towards elastic transactional cloud storage with range query support. *PVLDB*, 3(1):506–517, 2010.

[84] W. Vogels. Data access patterns in the amazon.com technology platform. In *VLDB*, pages 1–1. VLDB Endowment, 2007.

[85] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[86] T. von Eicken. Righscale Blog: Animoto's Facebook Scale-up. `http://bit.ly/animoto_elasticity`, April 2008. Retreived: October 2011.

[87] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., 2001.

[88] C. D. Weissman and S. Bobrowski. The design of the force.com multitenant internet application development platform. In *SIGMOD Conference*, pages 889–896, 2009.

[89] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacigumus. Intelligent management of virtualized resources for database systems in cloud environment. In *ICDE*, pages 87–98, 2011.

[90] F. Yang, J. Shanmugasundaram, and R. Yerneni. A scalable data platform for a large number of small applications. In *CIDR*, 2009.