Bulletin of the Technical Committee on

Data Engineering

March 2015 Vol. 38 No. 1

IEEE Computer Society

Letters

Special Issue on Data Consistency in the Cloud

Conference and Journal Notices

The TC on Data Engineering
Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is `http://tab.computer.org/tcde/index.html`.

The Data Engineering Bulletin
The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at
`http://tab.computer.org/tcde/bull_about.html`.

Letter from the Editor-in-Chief

TCDE Chair Election Result

Members of the Technical Committee on Data Engineering (TCDE) have voted for a new TCDE. The turnout for the election was higher than in past elections, which demonstrates, I think, two things. One, past TCDE Chair Kyu-Young Whang's initiative to enlarge membership has resulted in a larger overall TCDE membership, and hence a larger electorate. Two, we had two strong candidates in Xiaofang Zhou and Erich Neuhold, which generate more interest than in the past.

The outcome is that Xiaofang Zhou is the new chair of the TCDE. I want to congratulate Xiaofang on his election victory. I am confident that Xiaofang will be a fine TCDE chair, and I look forward to working with him going forward. Xiaofang's letter as TC Chair appears on page 2 of the current issue.

The TCDE Chair is the one empowered to appoint the TCDE Executive Committee. The new committee is shown on the inside front cover of this edition of the Bulletin. You can also find the committee on the TCDE web site at http://tab.computer.org/tcde/exec.html.

The Current Issue

Not so long ago, it was thought that CAP theorem meant that one had to give up consistency if one wanted high availability and partition tolerance. This led some members in our community to advocate using some form of reduced consistency, e.g. eventual consistency. While the CAP theorem has not been overturned, the community more recently has refocused efforts on providing transactional consistency even in the presence of high availability and partition tolerance requirements. For example, when all nodes of a distributed system are within one data center, the risk of partitioning is enormously reduced. Under such a setting, providing transactional consistency is once again feasible, though even then it can be difficult.

Giving up transactional consistency was a characteristic of, e.g., Amazon DynamoDB, which left reconciliation up to applications. While this might be "feasible" when both application and system are done in the same organization, it exports from platform to applications (users) an enormous burden. In the platform business, imposing this kind of a burden on users is a prescription for going out of business. Thus, and not for the first time, our technical community is exploring how to provide transactional consistency with high performance and low latency in a distributed setting where high availability is required.

This "consistency" is the topic that Bettina Kemme has tackled in the current issue. Bettina knows this subject exceptionally well, and has used her knowledge of the field to bring together articles by leading researchers in the area who are working at the technical "frontier". I am particularly excited and interested in this issue as this is one of my core research areas. I have already scanned the articles in the process of working on the issue, and am eager to read them more thoroughly. I think you will be very well rewarded if you undertake a similar plan and study this issue.

David Lomet
Microsoft Corporation

Letter from the New TCDE Chair

I am deeply humbled and honored for the opportunity to serve as the Chair of IEEE Technical Committee on Data Engineering (TCDE). TCDE is the flagship international society for data management and database systems researchers and practitioners. I thank the trust of all TCDE members.

Let me start to pay our tribute to my predecessor, Kyu-Young Whang. Under his leadership, the visibility of TCDE has been greatly increased, through the initiation of TCDE Awards program to recognize the people in our community who have made significant impact in data engineering research, system development, applications and education, and the TCDE Archive taskforce to maintain the cooperate memory of TCDE activities including ICDE conferences. The membership of TCDE has increased in the last a couple of years more than three times to now nearly 1000 members. TCDE continued to develop close collaborative relationship with SIGMOD and VLDB in order to build a healthier and stronger database community. TCDEs financial position has also been strengthened, that gives us more room to provide better support to our members and TCDE related activities. TCDE was evaluated as Excellent by IEEE Technical & Conference Activities Board in 2014.

Previous TCDE Chairs, Paul Larson, David Lomet and Kyu-Young Whang, have established an excellent framework for TCDE. As the new Chair of TCDE, I will lead TCDE under this framework. One of my priorities is to work with the ICDE steering committee as well as the VLDB Endowment and ACM SIGMOD to ensure that ICDE maintains its status as a top quality international database conference. The Data Engineering Bulletin is TCDEs flagship publication with a proud history of 37 years. I thank the Editor-in-Chief, David Lomet and his editorial board to their dedicated work to provide this invaluable service to our community. We will strengthen our support the Data Engineering Bulletin to make it continuously to be a high value information source for all TCDE members and the broader community.

TCDE to IEEE is similar to SIGMOD to ACM. During my term as the Chair, I will continue to increase the visibility of TCDE, to bring to the level of SIGMOD. We will continue to promote the TCDE Award program, to further increase the TCDE membership, and to encourage more people to engage with TCDE.

With the support of the TCDE Executive Committee members, previous TCDE chairs, and most importantly, all TCDE members, I am confident that I could lead TCDE to the next level.

Xiaofang Zhou
University of Queensland
Brisbane, Australia

Letter from the Special Issue Editor

Offering transactional properties has been one of the cornerstones of the success of database management systems. As database technology has evolved, covering new application domains, handling new data models, and supporting new hardware and distributed infrastructure, transaction management had to be redefined, redesigned and rethought.

This starts by understanding what are actually the requirements of the applications in terms of "keeping data consistent" and providing users with a "consistent view of the data". Do they require the traditional transactional properties such as atomicity and strong isolation levels or can they accept weaker levels of consistency? Furthermore, the distributed nature of current data stores comes with great promises but also challenges. Replication is a common mechanism to provide better performance and availability but comes with its own cost of keeping replicas consistent. While scaling up by adding new server nodes every time demand increases sounds promising, transaction management tasks have shown to be very difficult to scale either relying on complex coordinating protocols such as two phase commit or on a central component that is difficult to scale. Distribution and replication across geo-distributed data centres adds the additional cost of wide-area communication.

Many different research communities are actively looking for solutions each bringing their own expertise and background. This becomes obvious when looking at the venues in which work on consistency for distributed data is published. Contributions can be found in conferences and journals related to database systems, distributed systems, operating systems, storage systems, and computer systems in general. This exchange of ideas broadens the scope and has led to a whole new range of possibilities.

This diversity becomes apparent in the first three articles of this issue that explore and analyze the wide range of consistency models and definitions that have been proposed in the recent past, how they relate to each other, what are there trade-offs, and how they can be implemented. In the first article, Agrawal *et al.* explore the design space in regard to distribution and replication, how replica management is intertwined with transaction management, and what are the options to apply changes to replicas. In the second article, Faleiro and Abadi discuss the trade-off between fairness, isolation and throughput in transactional systems, arguing that only two out of the three can be achieved at the same time. Along a set of recently developed data stores, they show how each of them only supports two of these properties. In the third article, Bravo *et al.* discuss the importance of logical and physical clocks to track the order of events in a distributed system, their costs in term of space and coordination overhead, and how they they are used in a wide range of systems to implement both weak as well as strong consistency.

The second set of papers presents concrete solutions to support large-scale transactional applications in distributed environments. The first two papers present novel concurrency and recovery components. Bernstein and Das present an optimistic concurrency approach based on parallel certifiers that supports transactions that access more than one partition while keeping global synchronization to a minimum. They rely on causal message ordering to guarantee the proper serialization order across all partitions. Levandoski *et al.* present the Deuteronomy architecture which separates transaction functionality from storage functionality. Their transaction manager, based on mulit-version concurrency control, exploits modern hardware and specialized data structures, and uses caching and batching to minimize data transfer. The last two papers present cloud platforms specifically designed to provide scalability and elasticity for transactional applications. Salomie and Alonso present the Vela platform that exploits both virtualization and replication. Virtualization enables dynamic and elastic deployment of database instances within and across heterogeneous servers while replication offers scalability. Finally, Jimenez-Peris *et al.* present CumuloNimbo, an elastic fault-tolerant platform for multi-tier applications providing a standard SQL interface and full transactional support across the entire data set.

I hope you enjoy reading these articles as much as I did!

Bettina Kemme
McGill University
Montreal, Canada

A Taxonomy of Partitioned Replicated Cloud-based Database Systems

Divy Agrawal
University of California Santa Barbara

Amr El Abbadi
University of California Santa Barbara

Kenneth Salem
University of Waterloo

Abstract

The advent of the cloud computing paradigm has given rise to many innovative and novel proposals for managing large-scale, fault-tolerant and highly available data management systems. This paper proposes a taxonomy of large scale partitioned replicated transactional databases with the goal of providing a principled understanding of the growing space of scalable and highly available database systems. The taxonomy is based on the relationship between transaction management and replica management. We illustrate specific instances of the taxonomy using several recent partitioned replicated database systems.

1 Introduction

The advent of the cloud computing paradigm has given rise to many innovative and novel proposals for managing large-scale, fault-tolerant and highly available processing and data management systems. Cloud computing is premised on the availability of large data centers with thousands of processing devices and servers, which are connected by a network forming a large distributed system. To meet increasing demands for data storage and processing power, cloud services are scaled out across increasing numbers of servers. To ensure high availability in the face of server and network failures, cloud data management systems typically replicate data across servers. Furthermore, the need for fault-tolerance in the face of catastrophic failures has led to replication of data and services across data centers. Such geo-replication holds the promise of ensuring continuous global access.

This paper considers the problem of providing scalability and high availability for *transactional database systems*. These are database systems that support the on-going operations of many organizations and services, tracking sales, accounts, inventories, users, and a broad spectrum of similar entities and activities. A key feature of these systems is support for transactional access to the underlying database. Transactions simplify the development and operation of complex applications by hiding the effects of concurrency and failures.

To provide scalability and high availability, databases are typically partitioned, replicated, or both. Partitioning splits a database across multiple servers, allowing the database system to scale out by distributing load across more and more servers. Replication stores multiple copies of a partition on different servers. If the replicas are

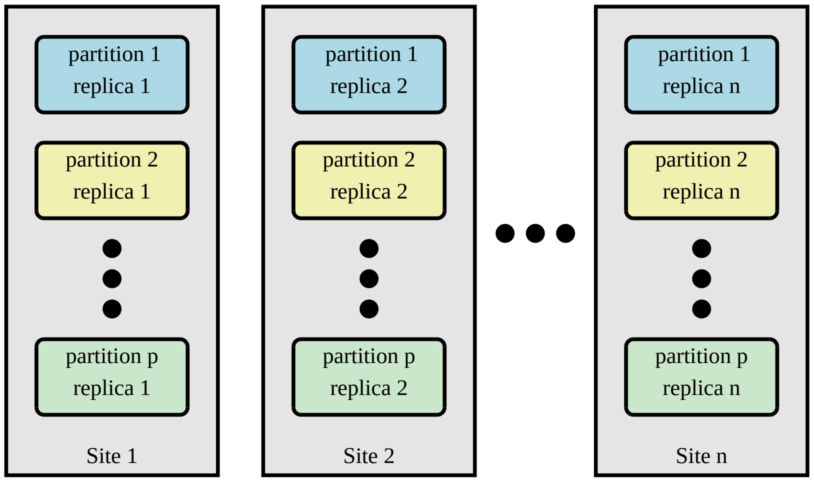


Figure 1: A Replicated Partitioned Database

distributed across multiple data centers, they can provide fault-tolerance even in the presence of catastrophic failures, such as earthquakes or hurricanes. Such replication is often referred to as geo-replication.

Many modern database systems operate over partitioned, replicated databases, and they use a variety of different techniques to provide transactional guarantees. Our objective in this paper is to present a simple taxonomy of such systems, which focuses on *how* they implement transactions. By doing so, we hope to provide a framework for understanding the growing space of scalable and highly available database systems. Of course, these database systems differ from each other in many ways. For example, they may support different kinds of database schemas and provide different languages for expressing queries and updates. They also target a variety of different settings, e.g., some systems geo-replicate so as to survive catastrophes, while others are designed to scale out within a single machine room or data center. We will ignore these distinctions as much as possible, thus allowing the development of a simple taxonomy that focuses on transaction mechanisms.

2 Replicated Partitioned Databases

We begin with an abstract model of a distributed, replicated transactional database system. This model will provide a common context for the subsequent presentation of the taxonomy. We assume that the system stores a large database. To allow the system to scale up, the database is divided into p non-overlapping *partitions*, so that each partition can be managed by a separate server. Partitioning is accomplished in different ways in different systems, but we will not be concerned here with the way that the partitions are defined.

For the sake of availability, each partition of the database is replicated n times. As shown in Figure 1, each complete copy of the database is referred to as a *site*. Because there are n complete copies of the database, there are n sites. In a geo-replication setting, these sites may be housed in geographically distributed data centers.

Applications perform queries and updates against the database. A transaction is an application-defined group of database operations (queries and updates) that can be executed as an indivisible atomic unit. The database servers that manage the database partition replicas work together as a distributed database management system to handle the application's transactions. The "gold standard" for the distributed database system is to provide a *one-copy serializability* [3] guarantee for application transactions. Such a guarantee says that transactions will behave as if they executed sequentially on a single (non-replicated) copy of the database. In practice, however, some of the systems that we will describe will fall short of this gold standard in one or more ways.

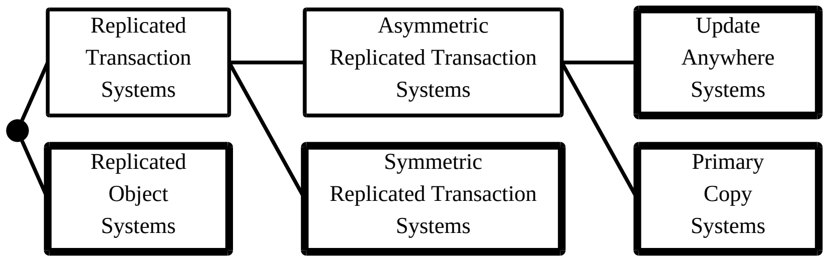

Replicated Transaction Systems
Asymmetric Replicated Transaction Systems
Update Anywhere Systems
Replicated Object Systems
Symmetric Replicated Transaction Systems
Primary Copy Systems

Figure 2: Taxonomy of Partitioned, Replicated Database Systems

3 Taxonomy

Database systems that manage replicated, distributed databases face a two-dimensional problem. First, even if the database were not replicated, it would be necessary to coordinate the database partitions to enforce transactional (ACID) guarantees. Many concurrency control, recovery, and commitment techniques exist to solve this *transaction management* problem. Second, the database system must somehow synchronize the database replicas so that replication can be hidden from the application, and so that the system can remain available even when some of the replicas are down. A variety of existing replica synchronization techniques can be used to solve this *replica management* problem.

Our taxonomy, which is shown in Figure 2, is based on the relationship between transaction management and replica management in these systems. Since there are well-known techniques for both transaction management and replica management, the challenge in designing a distributed replicated database system is how to combine techniques to arrive at an effective design that will address both problems. Thus, the relationship between transaction management and replica management is a natural basis for our taxonomy.

The top level of the taxonomy distinguishes between two general types of systems: *replicated object systems* and *replicated transaction systems*. In replicated object systems, replica management is used to make replicated database partitions look like non-replicated partitions. Transactions are then implemented over the logical, non-replicated partitions, using any of the well-known transaction management techniques. Thus, replicated object systems can be thought of as implementing transaction management on top of replica management.

Replicated transaction systems, in contrast, implement replica management on top of transaction management. In replicated transaction systems, transactions run over individual partition replicas, rather than logical partitions. That is, each *global transaction* is implemented using n separate *local transactions*, one at each site. Each site has a local transaction manager that ensures that its local transactions have ACID properties with respect to other local transactions at that site. (In contrast, replicated object systems have a single *global* transaction manager.) Each local transaction is responsible for applying the effects of its parent global transaction to the replicas at its own local site. We will refer to the local transactions as *children* of their (global) parent.

We further divide replicated transaction systems into two subclasses, *symmetric* and *asymmetric*. In symmetric replicated transaction systems, all of the local transactions of a given parent transaction are the same. Typically, they can run concurrently at the different sites. In contrast, in an asymmetric system, one local *master transaction* runs first at a single site. If that local transaction is able to commit, then the remaining local transactions are run at the other sites. We will refer to these remaining transactions as *update propagation transactions*. Typically, the update propagation transactions perform only the updates of the master transaction, and do not perform any reads.

Finally, we distinguish two types of asymmetric replicated transaction systems. In *primary copy* systems, all master transactions run at a single *primary* site. The remaining sites only run update propagation transactions. In *update anywhere* systems, master transactions may run at any site.

4 Examples

In this section, we briefly discuss several recent partitioned, replicated database systems, describing them in terms of the our taxonomy. We have included an example from each of the "leaf" categories of the taxonomy shown in Figure 2.

4.1 Replicated Object Systems

A prominent example of a replicated object system is Google's Spanner [5], which is designed to support geo-replication. Other systems that use the replicated object approach are the Multi-Data Center Consistency (MDCC) [7] system, which is also designed to allow geo-replicated data, and Granola [6], which is not.

Spanner stores keyed records, with related records grouped by the application into directories. A *tablet* in Spanner is a collection of directories or fragments of directories. Tablets are replicated. Thus, they correspond to partitions in our generic architecture (Figure 1) - we will refer to them here as partitions. Spanner uses Paxos [8] to synchronize the replicas of each partition across sites. Spanner uses a separate instance of Paxos, with a long-lived leader, for each partition.

To implement transactions, including transactions that involve multiple partitions, Spanner uses two-phase locking for concurrency control, and two-phase commit. The Paxos leader in each partition is responsible for participating in these protocols on behalf of that partition. It does so in much the same way that it would if its partition were not replicated, except that changes to the partition's state are replicated using Paxos instead of just being stored locally at the leader's server. The leaders serve as the link between the transaction protocols and Paxos, by ensuring that both database updates and changes in the state of the transaction are replicated.

The transaction protocol described here is a simplified version of the full protocol used by Spanner. The full protocol uses a system called TrueTime to associate timestamps with transactions, and ensures that transactions will be serialized in the order in which they occur in time. Spanner also uses TrueTime (and versioned objects) to support point-in-time read-only transactions, which "see" a snapshot of the database at a particular time. However, these features of the full Spanner protocol do not change its underlying nature, which is that of a replicated object system.

4.2 Symmetric Replicated Transaction Systems

An example of a symmetric replicated transaction approach is UCSB's replicated commit protocol [9]. Under the replicated commit protocol, each site includes a transaction manager capable of implementing transactions over the database replica local to that site. Each global transaction is implemented as a set of local child transactions, one at each site.

As is the case with Spanner, a global transaction's update operations are deferred until the client is ready to commit the transaction. To commit a transaction, the client chooses a *coordinating partition* for that transaction. It then contacts the replica of that partition at each site, and requests that the transaction commit locally at that site, providing a list of the transaction's updates.

At each site, the coordinating partition is responsible for determining whether that site is prepared to commit its local child of the global transaction. If so, the coordinating partition notifies the coordinators at other sites, and the client, that its site is prepared to commit the transaction. The global transaction will be committed if the local coordinators at a majority of the sites vote to commit it.

Under this protocol, it is possible that a transaction that commits globally may not commit at some sites, since only a majority of sites must agree on the commit decision. Thus, some transactions' updates may be missing at some sites. In order to ensure that it will observe all committed updates, a client that wishes to read data from a partition sends its read request to *all* replicas of that partition, waits for a majority of the replicas to respond, and chooses the latest version that it receives from that majority. To ensure global one-copy serializability, each local replica acquires a (local) read lock when it receives such a request.

4.3 Primary Copy Systems: Cloud SQL Server

A recent example of a primary copy system is Microsoft's Cloud SQL Server [2], which is the database management system behind the SQL Azure [4] cloud relational database service. Cloud SQL Server supports database partitioning (defined by application-specified partitioning attributes) as well as replication. However, transactions are limited to a single partition unless they are willing to run at the read committed SQL isolation level. Cloud SQL server is not designed for geo-replication, so all of database replicas (the "sites" shown in Figure 1) are located in the same datacenter.

In Cloud SQL Server, one replica of each partition is designated as the primary replica. Clients direct all transactions to the primary replica, which runs them locally. When a transaction is ready to commit at the primary site, the transaction manager at the primary assigns it a commit sequence number. It then sends a commit request to each of the secondary replicas. Each secondary replica then starts a local transaction to apply the updates locally at that replica, and sends an acknowledgment to the primary when it has succeeded. Thus, Cloud SQL Server is an asymmetric replicated transaction system since each transaction runs first (to the commit point) at the primary site before being started at the secondaries. Once the primary has received commit acknowledgments from a majority of the secondary replicas, it commits the transaction at the primary and acknowledges the commit of the global transaction to the client.

4.4 Update Anywhere Systems: Megastore

In primary copy asymmetric systems like Cloud SQL Server, the master children of all global transactions run at the same site. In update anywhere asymmetric systems, the master children of different global transactions may run at different sites. Google's Megastore [1] is an example of an asymmetric update anywhere systems. In Megastore, database partitions, which are called *entity groups*, are replicated and geographically distributed.

In Megastore, a client can initiate a single-partition transaction at any replica of that partition - typically, the client will use a nearby replica. For each partition, Megastore manages a transaction log, which is replicated to all sites using Paxos. The client reads the partition log to determine the sequence number of the log's next free "slot", then runs the transaction locally at its chosen replica, deferring updates until it is ready to commit the transaction. When the client is ready to commit the transaction, its chosen replica attempts to write a transaction record to the replicated log, using Paxos, into the slot that it read before starting the transaction. If this succeeds, the transaction is committed. The initiating site applies the updates to its replica, and all other sites read the transaction's updates from the shared log and apply them to their replicas using local transactions. Thus, the replicated transaction log serves to coordinate the sites, ensuring the transactions commit in the same order everywhere.

5 Conclusion

In this paper we proposed a taxonomy for partitioned replicated database systems that reside in the cloud. Our focus was on databases that support both transactions and replication. We used several state of the art systems to illustrate specific instances of the taxonomy. The taxonomy is skewed, in the sense that there are several subcategories of Replicated Transaction Systems, but only one category of Replicated Object Systems. This taxonomy thus represents a first principled attempt to understand and classify the transaction mechanisms of many of the proposed state of the art systems. We hope this will lead to a better understanding of the trade-offs offered, as well as a potentially more elaborate and extensive taxonomy in the future.

References

[1] Jason Baker, Chris Bond, James Corbett, J.J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. Conf. on Innovative Database Research (CIDR)*, January 2011.

[2] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kallan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL Server for cloud computing. In *Proc. IEEE Int'l Conf. on Data Engineering*, 2011.

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

[4] David G. Campbell, Gopal Kakivaya, and Nigel Ellis. Extreme scale with full SQL language support in Microsoft SQL Azure. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 1021–1024, 2010.

[5] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *USENIX Conference on Operating Systems Design and Implementation*, 2012.

[6] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *Proc. USENIX Annual Technical Conf.*, June 2012.

[7] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *Proc. EuroSys*, pages 113–126, April 2013.

[8] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169.

[9] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9):661–672, July 2013.

FIT: A Distributed Database Performance Tradeoff

Jose M. Faleiro
Yale University
jose.faleiro@yale.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

Abstract

Designing distributed database systems is an inherently complex process, involving many choices and tradeoffs that must be considered. It is impossible to build a distributed database that has all the features that users intuitively desire. In this article, we discuss the three-way relationship between three such desirable features — fairness, isolation, and throughput (FIT) — and argue that only two out of the three of them can be achieved simultaneously.

1 Introduction

A transaction that involves data which resides on more than one node in a distributed system is called a distributed transaction. At a minimum, distributed transactions must guarantee atomicity; the property that either all or none of a transaction's updates must be reflected in the database state. Atomicity prohibits a scenario where updates are selectively applied on only a subset of nodes. In order to guarantee atomicity, distributed transactions require some form of coordination among nodes; at the very least, each node must have a way of informing other nodes of its willingness to apply a transaction's updates.

As a consequence of the unavoidable coordination involved in distributed transactions, database lore dictates that there exists a tradeoff between *strong isolation* and *throughput*. The intuition for this tradeoff is that in a system which guarantees isolation among conflicting transactions, distributed coordination severely impacts the amount of concurrency achievable by the system by extending the time for which conflicting transactions are unable to make meaningful progress. The only way to get around this limitation is to provide weak isolation, which allows transactions to execute concurrently *despite the presence of conflicts*. The intuition is clear: a system which implements distributed transactions can provide either one of strong isolation or good throughput, but not both.

In this article, we put this intuition under the microscope, and find that there exists another variable, *fairness*, that influences the interplay between strong isolation and throughput. We make the case that weakening isolation is not the only way to "defeat" distributed coordination; if a system is given the license to selectively prioritize or delay transactions, it can find a way to pay the cost of distributed coordination without severely impacting throughput. Based on our intuition and an examination of existing distributed database systems, we propose that instead of an isolation-throughput tradeoff, there exists a *three-way* tradeoff between fairness, isolation, and throughput (FIT); a system that foregoes one of these properties can guarantee the other two. In this way,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

designers who are comfortable reasoning about the three-way tradeoff behind the CAP theorem [10] can use the same type of reasoning to comprehend the FIT tradeoff for distributed database systems.

The remainder of this article proceeds as follows: Section 2 outlines our assumptions and definitions. Section 3 describes the general principles underlying the FIT tradeoff. Section 4 describes FIT in action using examples of real systems. Section 5 discusses how the tradeoff may be more broadly applicable beyond distributed transactions. Section 6 concludes the paper.

2 Assumptions and Definitions

We assume that data is *sharded* across a set of nodes in a distributed system. Each node is referred to as a partition. A distributed transaction refers to a type of transaction whose read- and write-sets involve records on more than one partition. A transaction must be guaranteed to have one of two outcomes; commit or abort. We classify aborts as being either logic-induced or system-induced. A logic-induced abort occurs when a transaction encounters an explicit abort statement in its logic. A system-induced abort occurs when the database forcibly aborts a transaction due to events beyond the transaction's control such as deadlocks and node failures.

When processing distributed transactions, we require that a database satisfy *safety*, *liveness*, and *atomicity*; defined as follows:

- **Safety.** The partitions involved in a distributed transaction must agree to either commit or abort a transaction. A transaction is allowed to commit if *all* partitions can commit, otherwise, it must abort.
- **Liveness.** If a distributed transaction is always re-submitted whenever it experiences a system-induced abort, then it is guaranteed to eventually commit. [1]
- **Atomicity.** If a transaction commits, then *all* of its updates must be reflected in database state. Otherwise, none of its updates must be reflected in the database's state.

2.1 Fairness

We do not attempt to formally define fairness. Instead, we provide intuition for what it means for a system to be fair and then discuss our intuition in the context of two examples: group commit [8] and lazy transaction evaluation [9].

Fairness corresponds to the idea that a database system does not deliberately prioritize nor delay certain transactions. When a transaction is received by the database, the system immediately attempts to process the transaction, and never artificially adds latency to a transaction (i.e. blocking the transaction for reasons other than waiting for data or for conflicting transactions to finish) for the purpose of facilitating the execution of other transactions.

Database systems have a long history of trading off fairness in order to obtain good performance. One of the prominent examples, "group commit", was proposed by DeWitt et al. in the context of database systems whose records could reside entirely in main-memory [8]. In such a setting, the time to execute a transaction's logic is dramatically shortened, but the cost of writing log records to disk – for durability purposes – limits overall throughput. Accordingly, DeWitt et al. proposed that the database accumulate log records from several transactions into a single disk write. Writing log records in batches circumvented the disk write bottleneck at the expense of fairness; despite having executed their logic, certain transactions are unable to commit until the set of buffered log records exceeds an appropriate threshold.

In prior work, we proposed a mechanism for lazy transaction evaluation, one of whose goals was to exploit spatial locality in on-line transaction processing systems [9]. Lazy transaction evaluation exploits spatial locality

[1] Liveness guards against partitions trivially achieving safety by choosing to abort transactions via system-induced aborts.

by amortizing the cost of bringing database records into the processor cache and main-memory buffer pool over several transactions. In order to achieve this, we proposed that a lazy database system defer the execution of transactions in order to obtain batches of unexecuted transactions whose read- and write-sets overlapped. Leaving transactions unexecuted meant that obtaining the most up-to-date value of a record would involve executing deferred transactions. Hence, when a client issued a query to read the value of a record, the query would need to initiate and wait for the execution of the appropriate set of deferred transactions. Lazy evaluation ensured that data-dependent transactions were executed together, so that the cost of bringing records into the processor cache and main-memory buffer pool was amortized across those transactions. This led to throughput benefits but came at the expense of fairness to client issued queries.

2.2 Isolation

Synchronization independence is the property that one transaction cannot cause another transaction to block or abort, even in the presence of conflicting data accesses [1]. Synchronization independence effectively decouples the execution of concurrently executing transactions. As a consequence, systems which allow transactions to run with synchronization independence are susceptible to race conditions due to conflicts.

Intuitively, synchronization independence implies that a system is unable to guarantee any form of isolation among conflicting transactions. We thus classify a system as providing weak isolation if it allows transactions to run with synchronization independence. If a system does not allow transactions to run with synchronization independence, it is classified as providing strong isolation.

3 The FIT Tradeoff

The safety and liveness requirements of Section 2 *necessitate* some form of coordination in processing a distributed transaction. The partitions involved in a transaction need to agree on whether or not to commit a transaction (safety), and cannot cheat by deliberately aborting transactions (liveness). In order to reach agreement, partitions require some form of coordination. The coordination involved in distributed transaction processing is at the heart of the FIT tradeoff.

The coordination associated with distributed transactions entails a decrease in concurrency in systems which guarantee strong isolation. If a transaction is in the process of executing, then, in order to guarantee strong isolation, conflicting transactions must not be allowed to make any meaningful progress. Distributed coordination extends the duration for which conflicting transactions are unable to make meaningful progress, which limits the throughput achievable under strong isolation.

This suggests that one way to reduce the impact of coordination is to allow transactions to run with synchronization independence (Section 2.2). Synchronization independence implies that conflicting transactions do not interfere with each other's execution. The implication of synchronization independence is that the inherent coordination required to execute a distributed transaction has no impact on the execution of other transactions. In other words, coordination has no impact on concurrency. As a consequence, systems which run transactions with synchronization independence are able to achieve good throughput. However, this comes at the expense of strong isolation. Since synchronization independence implies that transactions do not block due to conflicts, a weak isolation system does not stand to gain any benefits from sacrificing fairness; there is no point delaying or prioritizing transactions because there exist no inter-transaction dependencies in the first place.

Unlike systems that offer weak isolation, strong isolation systems must ensure that only a single transaction among many concurrent conflicting transactions is allowed to make meaningful progress. Any kind of latency due to coordination in the middle of a transaction thus reduces concurrency. In order to achieve good throughput, a strong isolation system must find a way to circumvent the concurrency penalty due to coordination. This can be done by giving up fairness. In giving up fairness, the database gains the flexibility to pick the most opportune moment to pay the cost of coordination. *Which* moment is "most opportune" depends on the specific technique a system uses to circumvent the cost of coordination. For example, in some cases, it is possible to perform coordination outside of transaction boundaries so that the additional time required to do the coordination does not increase the time that conflicted transactions cannot run. In general, when the system does not need to guarantee fairness, it can deliberately prioritize or delay specific transactions in order to benefit overall throughput. A system which chooses to eschew fairness can guarantee isolation and good throughput.

A system which chooses fairness must make its best effort to quickly execute individual transactions. This constrains the system's ability to avoid the reduction in concurrency due to coordination. Therefore, systems which guarantee fairness and strong isolation cannot obtain good throughput.

In summary, the choice of how a system chooses to pay the cost of coordination entailed in supporting distributed transactions divides systems into three categories: 1) Systems that guarantee strong isolation and fairness at the expense of good throughput, 2) Systems that guarantee strong isolation and good throughput at the expense of fairness, and 3) Systems that guarantee good throughput and fairness at the expense of strong isolation.

4 FIT in Action

In this section, we examine several distributed database systems, and classify them according to which two properties of the FIT tradeoff they achieve.

4.1 Isolation-Throughput Systems

4.1.1 G-Store

G-Store is a system which extends a (non-transactional) distributed key-value store with support for multi-key transactions [5]. G-Store's design is tailored for applications that exhibit temporal locality, such as online multi-player games. During the course of a game instance, players interact with each other and modify each other's state. Transactions pertaining to a particular instance of a game will operate on only the keys corresponding to the game's participants.

In order to allow applications to exploit temporal locality, G-Store allows applications to specify a *KeyGroup*, which is a group of keys whose values are updated transactionally. G-Store requires that the scope of every transaction is restricted to the keys in a single KeyGroup. When an application creates a KeyGroup, G-Store moves the group's constituent keys from their respective partitions onto a single *leader* node. Transactions on a KeyGroup always execute on the group's leader node, which precludes the need for a distributed commit protocol.

By avoiding a distributed commit protocol, the period for which conflicting transactions on the same KeyGroup are blocked from making meaningful progress is significantly reduced. G-Store thus pays the cost of distributed coordination prior to transaction execution by moving all the keys in a KeyGroup to the leader node. KeyGroup creation involves an expensive grouping protocol, the cost of which is amortized across the transactions which execute on the KeyGroup.

The requirement that transactions restrict their scope to a single KeyGroup favors transactions that execute on keys which have already been grouped. On the flip side, this requirement is unfair to transactions that need to execute on a set of as yet ungrouped keys. Before such transactions can begin executing, G-Store must first disband existing KeyGroups to which some keys may belong, and then create the appropriate KeyGroup.

4.1.2 Calvin

Calvin is a database system designed to reduce the impact of coordination required while processing distributed transactions [14, 15]. Prior to their execution, Calvin runs transactions through a preprocessing layer, which imposes a total order on the transactions. The database's partitions then process transactions such that their serialization order is consistent with the total order, which implies that Calvin guarantees serializable isolation (satisfying our criteria for strong isolation).

In imposing a total order on transactions, the preprocessing layer solves two important problems:

- It eliminates deadlocks by pre-defining the order in which conflicting transactions must execute.
- It guarantees that a transaction's outcome is *deterministic*, despite the occurrence of failures. Each database partition is required to process transactions in a manner consistent with the total order, which implies that the total order of transactions is effectively a *redo log*; the state of the database is completely dependent on the total order of transactions.

The preprocessing layer is able to avoid deadlocks and ensure determinism because it is a logically (but not necessarily physically) *centralized* component[2] The use of a centralized component to disseminate information in a distributed system is a form of coordination. By funneling transactions through the preprocessing layer, Calvin pays some of the cost of distributed coordination up front, prior to the actual execution of transactions.

The guarantees of determinism and deadlock-freedom, obtained via coordination *prior* to transaction execution, keep the amount of coordination required *during* transaction execution down to a bare minimum. Deadlock-freedom eliminates any form of coordination required to deal with deadlocks. Determinism implies that, in the event of a failure, a partition can correctly re-construct its state from the total order of transactions. Therefore, partitions do not need to plan for failures using expensive operations such as forced log writes and synchronous replication. Minimizing the impact of coordination implies that the amount of time that conflicting transactions are blocked from making progress is also minimized, which helps Calvin achieve good throughput.

The preprocessing layer is replicated for fault-tolerance, and uses Paxos [11] to decide on a total ordering for transactions. Since it is logically centralized, the preprocessing layer must be able to totally order transactions at a rate which allows partitions to be fully utilized. If this were not the case, then Calvin's throughput would be bottlenecked by preprocessing layer. Accordingly, the preprocessing layer runs each Paxos instance over a large batch of transactions, amortizing the cost of consensus across the batch. In batching the input to each Paxos instance, the preprocessing layer sacrifices fairness; the transaction at the beginning of the batch must wait for enough transactions to accumulate before it can be processed. The *fair* strategy would be to run a Paxos instance on every transaction, however, its throughput would not be enough to fully utilize Calvin's partitions.

4.2 Fairness-Isolation Systems

4.2.1 Spanner

Spanner is Google's geo-scale distributed database system [4]. Spanner's data is partitioned for scale and each partition is replicated for fault-tolerance. Spanner uses multiversioning to support non-blocking read-only transactions, and a combination of two-phase locking (2PL) and two-phase commit (2PC) for transactions which perform updates. Spanner's concurrency control mechanisms guarantee that update transactions execute under strict serializability, which satisfies our criterion for strong isolation. Spanner processes transactions in a fair manner, a transaction begins executing as soon as it is submitted; Spanner does not deliberately delay transactions in order to benefit others.

[2]The predecessor of Calvin used a physically centralized preprocessor [13], while the more well-known versions of Calvin use a distributed preprocessor [14, 15].

Spanner commits transactions using 2PC. In a non-replicated implementation of 2PC, the coordinator must persist its commit/abort decision on disk, and each partition must similarly persist its prepare vote and the final commit decision. For correctness, some of these disk writes must be *forced*, which means that a node cannot take any further actions (related to the current instance of 2PC) until it is sure that the write is stable on disk. The coordinator's commit/abort decision and each partition's prepare vote fall into this category.

In a scenario where multiple copies of partitions exist, successful replication is analogous to a disk write. In order to correctly implement 2PC in a replicated setting, therefore, state must be "persisted" through replication. In keeping with the analogy, the equivalent of a forced write is *synchronous* replication. This means that before a distributed transaction can commit, it must pay the cost of replicating every partition's prepare vote (although per-partition replication occurs in parallel), and the cost of replicating the coordinator's final commit decision.

Spanner's 2PC protocol negatively impacts concurrency. In order to avoid cascading rollbacks and to guarantee strict serializable isolation, a transaction cannot release its locks until after it has obtained a commit decision, which implies that a transaction whose execution is blocked due to lock conflicts must wait for at least the duration of two synchronous replications, for *every* transaction that precedes it in the lock queue, for *every* lock it has not yet acquired. The end result is that Spanner's overall throughput is severely limited.

4.3 Fairness-Throughput Systems

4.3.1 Eventually Consistent Systems

Most eventually consistent systems do not support multi-record atomic transactions and therefore fall out of the scope of this article, which explores the fairness-isolation-throughput tradeoff involved in implementing distributed atomic transactions. For example, the original version of Dynamo supported only single-key updates [7][3]. As of the time of this article, Riak is in the same category [2].

However, Cassandra supports the notion of a "batch" — a "transaction" containing multiple INSERT, UPDATE, or DELETE statements that are performed as single logical operation — either the entire batch succeeds or none of it will [6]. Thus, a Cassandra batch supports the minimum level of atomicity to be within the scope of this article. However, there is no batch isolation. Other clients are able to read the partial results of the batch, even though these partial results may end up being undone if the batch were to abort. Thus, Cassandra's batch functionality clearly gives up isolation according to our definition in this article.

Since Cassandra makes no attempt to guarantee isolation of batches, the process of deciding whether a batch should succeed or abort can occur entirely independently of other batches or Cassandra operations. In general, throughput is unaffected by logical contention, and there is no need to delay or reorder transactions in order to improve throughput. Thus, by giving up isolation, the execution of Cassandra batches is both fair and occurs at high throughput.

4.3.2 RAMP Transactions

Bailis et al. propose a new isolation model, Read Atomic (RA) isolation, which ensures that either all or none of a transaction's updates are visible to others [1]. RA isolation's atomic visibility guarantee provides useful semantics on multi-partition operations where most weak isolation systems provide none.

RA isolation is implemented via Read Atomic Multi-Partition (RAMP) transactions. RAMP transactions guarantee *synchronization independence*, which means that a transaction *never* blocks others, even in the presence of read-write or write-write conflicts. Because conflicting transactions are allowed to concurrently update the same record(s), RAMP transactions cannot enforce value constraints (e.g., number of on call doctors must be greater than zero). We thus classify RAMP transactions as guaranteeing weak isolation.

[3]DynamoDB, a relatively recent Amazon Web Services product that is based partly on the design of Dynamo, supports multi-record atomic transactions. This OCC-based implementation gives up throughput to achieve isolation and fairness.

In order to provide atomicity (*not* the same as atomic visibility — see Section 2), RAMP transactions execute a commit protocol which resembles two-phase commit. Synchronization independence ensures that a transactions's commit protocol does not block the execution of conflicting transactions. As a consequence, the coordination entailed by RAMP transactions's commit protocol does not impact concurrency, which implies that RAMP transactions can achieve good throughput. Furthermore, because coordination does not impact concurrency, the system does not need to rely on unfairness to mask the cost of coordination. For this reason, RAMP transactions can guarantee good throughput *and* fairness simultaneously.

5 Applicability Beyond Distributed Transactions

One of the fundamental realities of modern multicore architectures is that contention on shared memory words can lead to severe scalability problems due to cache coherence overhead [3]. The impact of the cost of contention in multicore systems is analogous to the impact of distributed coordination in multi-partition transactions, it is an unavoidable cost that the database must mitigate in order to attain good performance. This suggests that the FIT tradeoff can also help guide the design of database systems on multicore machines. We now examine two recent serializable databases specifically designed to address the contention bottleneck on multicore machines by trading off fairness for overall system throughput.

Silo is a main-memory database system explicitly designed to reduce contention on shared memory words [16]. Like the main-memory system of DeWitt et al. [8], Silo buffers transactions's log records in memory prior to flushing them to disk. However, Silo's group commit technique is fundamentally different from that of DeWitt et al. because it is primarily designed to minimize synchronization across a machine's cores. Whenever a processing core appends log records to the in-memory buffer, it must synchronize with the other cores in the system (because the in-memory buffer is shared among all cores). In order to avoid this synchronization cost at the end of every transaction, each core tracks a transaction's log records in a *core-local* buffer. Cores periodically move the contents of their local buffers to the global buffer at the granularity of a large batch of transactions, amortizing the cost of synchronization over the batch. This reduction in synchronization comes at the cost of fairness; a transaction is unable to commit until its log records are moved to the global buffer and flushed to disk, even though it may have finished executing its logic much earlier.

Doppel is a main-memory database system which can exploit commutativity to increase the amount of concurrency in transaction processing [12]. Doppel exploits commutativity by periodically replicating high contention records across all cores in the system; when a record is replicated, commuting operations are allowed to execute on any given replica. In order to satisfy non-commuting operations, Doppel periodically aggregates the value at each replica into a single record. Doppel effectively cycles between two phases of operation: 1) a "joined-phase", in which only a single replica of a record exists, with no restrictions on which transactions are allowed to execute, 2) a "split-phase", in which high contention records are replicated across all cores, and only transactions with commuting operations on replicated records are allowed to execute (there exist no restrictions on operations on non-replicated records). Doppel is able to obtain concurrency in the split-phase by allowing commuting operations on replicated records. However, replicating records blocks the execution of transactions which include non-commuting operations on replicated records, which implies that Doppel is unable to guarantee fairness.

6 Conclusions

This article made the case for the existence of the fairness-isolation-throughput (FIT) tradeoff in distributed database systems. The FIT tradeoff dictates that a distributed database system can pick only two of these three properties; a system which foregoes one of these three properties can guarantee the other two. We believe that the FIT tradeoff is also applicable to the design of multicore database systems.

Although system implementers have long treated transaction isolation and throughput as “first-class citizens” of database design, we hope that explicitly thinking about the concept of “fairness” will help database architects more completely understand the tradeoffs involved in building distributed database systems.

7 Acknowledgments

We thank Peter Bailis and Alexander Thomson for providing insightful feedback on this article.

References

[1] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable atomic visibility with ramp transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 27–38. ACM, 2014.

[2] Basho. Riak. `http://basho.com/riak`.

[3] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[4] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google’s globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[5] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174. ACM, 2010.

[6] Datastax. Cql for cassandra 1.2. `http://www.datastax.com/documentation/cql/3.0/cql/cql_reference/batch_r.html`.

[7] 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 *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP ’07, pages 205–220. ACM, 2007.

[8] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 1–8. ACM, 1984.

[9] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 15–26. ACM, 2014.

[10] S. Gilbert and N. Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[11] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[12] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pages 511–524. USENIX Association, 2014.

[13] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, 2010.

[14] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[15] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Fast distributed transactions and strongly consistent replication for oltp database systems. *ACM Transactions on Database Systems (TODS)*, 39(2):11, 2014.

[16] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.

On the use of Clocks to Enforce Consistency in the Cloud

Manuel Bravo[*†], Nuno Diegues[*], Jingna Zeng[*‡], Paolo Romano[*], Luís Rodrigues[*]

[*]Instituto Superior Técnico, Universidade de Lisboa, Portugal
[†]Université Catholic de Louvain, Belgium
[‡]KTH/Royal Institute of Technology, Sweden

Abstract

It is well known that the ability to timestamp events, to keep track of the order in which they occur, or to make sure they are processed in a way that is meaningful to the application, is of paramount importance in a distributed system. The seminal work by Leslie Lamport[30] has laid the foundations to understand the trade-offs associated with the use of physically and logically synchronized clocks for establishing the order of events. Lamport's original work considered scalar clocks and has later been extended to consider richer forms of logical clocks such as vector clocks and matrix clocks. The purpose of this paper is to revisit these concepts in the concrete setting of developing distributed data store systems for the cloud. We look at how different clock mechanisms have been applied in practice to build different cloud data stores, with different consistency/performance trade-offs. Our goal is to gain a better understating of what are the concrete consequences of using different techniques to capture the passage of time, and timestamp events, and how these choices impact not only the performance of cloud data stores but also the consistency guarantees that these systems are able to provide. In this process we make a number of observations that may offer opportunities to enhance the next generations of cloud data stores.

1 Introduction

With the maturing of cloud computing, cloud data stores have become an indispensable building block of existing cloud services. The ideal cloud data store would offer low latency to clients operations (effectively hiding distribution), consistency (hiding concurrency), and high availability (masking node failures and network partitions). Unfortunately, it is today well understood that such an idealized data store is impossible to implement; a fact that has been captured by the following observation: in a distributed system subject to faults it is impossible to ensure simultaneously, and at all times, Consistency, Availability, and Partition tolerance, a fact that became known as the CAP theorem[22, 9]. In face of this impossibility, several different cloud data stores have been designed, targeting different regions of the design space in an attempt to find the sweetest spot among consistency, availability, and performance.

On one extreme, some systems have favoured concurrency and availability over consistency. For instance, Dynamo[12] allows for concurrent, possibly conflicting, operations to be executed in parallel on different nodes.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

It then provides the tools to help the application developers to eventually merge in a coherent manner the updates performed independently, which is known as *eventual consistency*. Practice has shown that reconciling conflicting updates can be extremely hard, and that it poses a heavy burden on programmers, favouring ad hoc solutions that are hard to compose and maintain[6]. For this reason, some people claim that eventually consistency is, in practice, no consistency[33].

On the other extreme, there are also examples of systems that favour strong consistency, at the cost of lower availability and performance[11, 38]. Although, as stated by the CAP theorem, strong consistency prevents unconditioned availability, in practice, if enough replicas are used, such that it is unlikely that a majority becomes failed or disconnected, the availability concerns can be mitigated. On the other hand, the coordination required to ensure the consistent processing of concurrently submitted operations introduces delays that are hard to overcome and that become visible to the user experience.

Between these two extremes of the spectrum, other systems have proposed to use weaker forms of consistency, such as causal consistency, to allow for a reasonable degree of concurrency while still providing clear and meaningful semantics to the application[33]. In fact, causal consistency is often regarded as the strongest consistency level that a system can provide without compromising availability and incurring high latencies [34].

Not surprisingly, regardless of the type of consistency guarantees that all these systems provide, they still need to track the relative order of different read and write requests in order to ensure that the state of the data store remains consistent with the user intents and that users observe updates in an expectable order. Key to correctly tracking such ordering relations is to use some form of clock in order to timestamp events of interest. These clocks can be either logical (i.e., just based on the number of events observed by the nodes) or physical (i.e, synchronized with some external source of universal time). In a seminal paper[30], Leslie Lamport has laid the foundations to understand the trade-offs among the use of physically synchronized clocks and the use of logically synchronized clocks for establishing the order of events. Lamport's original work considered scalar clocks and has later been extended to consider richer forms of logical clocks such as vector clocks[21, 35] and matrix clocks[42, 49]. As their name implies, vector and matrix clocks do not use a single scalar value; instead, they use multiple scalar values to capture more precisely the causal dependencies among events. Therefore, vector and matrix clocks require to maintain, and exchange, more metadata. A non-informed outsider may assume that systems providing stronger consistency are required to maintain more sophisticated forms of clocks and that systems providing weak consistency can be implemented with cheaper scalar clocks. Interestingly, this is not the case and the trade-offs involved are subtler than may appear at first sight.

In this paper, we compare recent cloud data store implementations, focusing on this crucial aspect that is at the core of their design, i.e., the choice of the mechanisms used to capture the passage of time and track cause-effect relations among data manipulations. In particular, we analyze a set of recent cloud data store systems supporting both strong and weak consistency, and which use a variety of approaches based on physical clocks (either loosely [17] or tightly synchronized [5]), various forms of logical clocks (scalar [40], vector [38] and matrix [18]), as well as hybrid solutions combining physical and logical clocks [1]. Our study encompasses a variety of cloud data stores, ranging from academic solutions, such as COPS [33] and ClockSI [17], to industry systems, such as Spanner [5] and Cockroach [1]. We provide a comparative analysis of alternative clock implementations, and identify a set of trade-offs that emerge when using them to enforce different consistency criteria in large-scale cloud data stores. Our aim is to derive insights on the implications associated with the design choices of existing solutions that may help cloud service builders to choose which types of stores are more suitable for their customers and services.

The remainder of this paper is structured as follows. We start by providing a generic reference architecture of a cloud service, that we will use to illustrate some of our points. Then, we review some fundamental notions and techniques for tracking time and establishing the order of events in a distributed system. These techniques represent crucial building blocks of the mechanisms that are used to enforce consistency in existing cloud data stores. In the following two sections, we focus on analyzing the design of recent distributed data stores providing weak (i.e., some form of causality) and strong (i.e., general purpose transactions) consistency. Subsequently, we

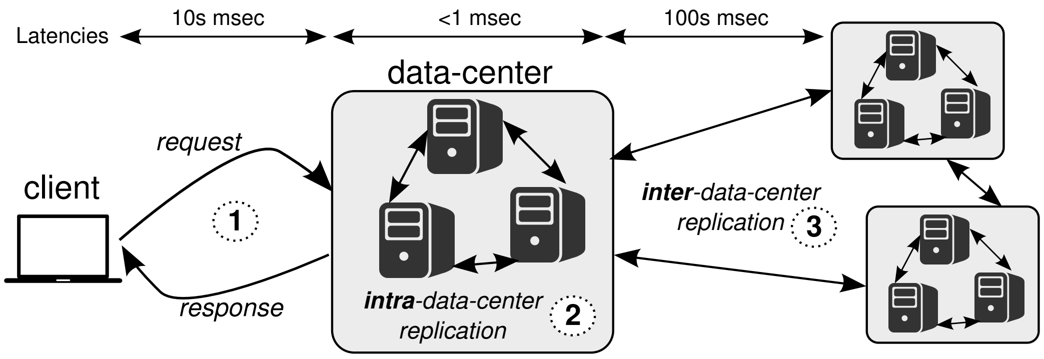


Figure 1: Generic Cloud Data Store Architecture.

draw a number of observations regarding the use of clocks for both weakly and strongly consistent systems. The last section offers some concluding remarks.

2 Architecture of a Cloud Data Store

We start by presenting a high level architecture of a cloud data store service, which we shall use as a generic framework for our subsequent discussion. An illustration of the architecture is presented in Figure 1. Typically, for fault-tolerance and locality, a data item is replicated in multiple nodes of the cloud data store. It is possible to replicate data just in a single data center, but this makes the service vulnerable to unavailability and data loss due to power outages, network problems, or natural catastrophes. Therefore, data is usually replicated in multiple, geographically replicated, data centers, which can also potentially reduce latencies.

In this setting, a client request is redirected to some data center, where it is served by some proxy node that abstracts the entire set of replicas. The role of the proxy is to initiate the required coordination to ensure that enough local and remote replicas are involved to provide the desired quality of service to the client. In theory, it is possible to implement a flat coordination scheme that treats all replicas alike, regardless of their location; for instance, one could use an algorithm that would apply an update by storing the data in a majority of all replicas (including replicas at local and remote data centers). However, the delays involved in contacting replicas within the same data center are several orders of magnitude smaller than the delays involved in the communication among different data centers. As a result, many geo-replicated data stores use some form of hierarchical coordination mechanism, where the algorithms used to maintain intra-data center replica consistency are different from the algorithms used to maintain inter-data center replica consistency.

Needless to say, it is cheaper to enforce strong consistency within a single data center than across multiple data centers. Therefore, a common pattern in many implementations (but not all) is that, when a client contacts a given data center, some local protocol is immediately executed to ensure strong consistency among the replicas within that data center. This means that, as long as the client keeps contacting the same data center, it can observe consistent results without incurring significant delays. On the other hand, synchronizations among data centers is often performed asynchronously. This means that updates performed in one data center may be observed with noticeable staleness in remote data centers. This also means that if a client is redirected to another data center it may either observe inconsistent results or be forced to wait for pending updates to be propagated. In the later case, clocks play a fundamental role to understand when consistency can be ensured.

Finally, different clients may try to update shared data concurrently at different data centers. To coordinate such updates is hard. In the general case, to avoid conflicting updates, coordination among data servers must be performed synchronously, i.e., finalized before replying to the client. However, as previously stated, many systems avoid this approach due to the potentially long delays involved in the process. This choice is supported by studies that have demonstrated the sensitiveness of users to even very small latency increases [44, 26, 32]. The consequence is that, in many systems, conflicting concurrent updates are accepted, and the resulting conflicts need to be resolved in background. In any case, either to perform eager coordination (for instance to run consensus among the data centers[37, 31, 5]), or just to order conflicting updates in a meaningful way, clocks play a role of paramount importance.

In this paper, we discuss the role of clocks in both the intra and inter data center coordination and also for both strong and weak consistency protocols.

3 Tracking the Ordering of Events in Distributed Systems

For self-containment, we provide a brief summary of different techniques that have been proposed to keep track of the ordering of events in distributed systems.

We assume that the nodes of the system are fully connected and that each node keeps one or more values to track the passage of time and to assign timestamps to events; these timestamps can be later used to order the corresponding events. We call this set of values a *clock*. If the clock maintains a single value we say that the system uses a *scalar clock*. If the clock maintains one value per node in the system we say that the system uses a *vector clock*[21, 35]. Finally, a clock that maintains a value for each link between pairs of nodes in the system is denoted a *matrix clock*[42, 49].

Orthogonal to the number of values in the clock is whether the values are supposed to capture the real time instant at which the events occurred or if the values are just logical values that count how many relevant events have been observed. In the first case we say the system uses *physical* values and in the second case *logical* values. More recently, some works have proposed to consider values that are actually a tuple, containing both a physical clock and a logical clock[28]. In this later case, we say that the system uses *hybrid* values[27].

When the two dimensions above are combined, we obtain data structures that have been used in different systems. For instance, if a system uses a scalar clock with physical values we say that it uses (scalar) physical clocks. Conversely, if it uses a scalar clock with logical values we say that it uses (scalar) logical clocks, also known as Lamport's clocks [30]. Finally, a system that uses a scalar clock with hybrid values is said to use a (scalar) hybrid clock. Similar combinations can be obtained for vector and matrix clocks.

In abstract, clocks based on physical values can be considered as potentially superior to logical clocks in the sense that, in addition to tracking potential cause-effect relations, they can allow to correlate the timestamp of events with the actual (real-time) instant at which a given event occurred. Unfortunately, in practice, physical clocks are subject to unavoidable drifts and are, as such, never perfectly synchronized. One way to address this issue is to rely on distributed clock synchronization algorithms, such as the Network Time Protocol (NTP) [36]: nonetheless, the quality of the synchronization achievable via these algorithms may vary significantly (e.g., depending on the network load), creating uncertainty intervals that can vary from only a few to hundreds of milliseconds. An alternative to address the synchronization problems is to rely on specialized hardware devices (such as GPS and atomic clocks)[5]: these mechanisms do allow for ensuring small and bounded uncertainty intervals (typically less than 10 msec), but these devices have a non-negligible cost and are not commonly available in public cloud infrastructures.

It is also obvious that scalar clocks are more space efficient than vector or matrix clocks. For instance, a system that uses vector clocks has to maintain metadata whose size is linear with the number of entities (namely, nodes across all the data centers) while a system that maintains scalar clocks maintains metadata whose size is constant with the number of entities. On the other hand, with a scalar clock it is, in general, impossible to assess if two events are concurrent or causally related. For instance consider events e and e' with timespamps t and t' respectively, where $\mathit{value}(t) < \mathit{value}(t')$. With scalar clocks we know that e' can causally depend on e or that e and e' are concurrent (but that, for sure, e does not causally depend on e'). With a vector (and hence a matrix) clock it is always possible to be sure if e and e' are causally dependent or concurrent.

4 Clocks for Weak Consistency

We now discuss the use of clocks in systems that trade consistency for availability; we mainly focus on systems that provide causal consistency. The common features of these systems are: (i) no support for general purpose update transactions (i.e., update operations address a single object) and sometimes the availability of read-only transactions (i.e., transaction that read multiple objects observe a consistent snapshot of the system) (ii) geo-replication across multiple data centers, (iii) asynchronous propagation of updates, (iv) causal consistency for operations that may access different objects, and (v) client support for ensuring causality by tracking dependences via dedicated metadata.

Dynamo Probably one of the most representative systems which rests on the extreme of the spectrum is Dynamo[12]. This system borrows multiple ideas and techniques previously proposed in the literature to build a cloud data store designed to be highly available in the presence of both temporary and permanent failures. This is achieved by relying on a weak form of quorums, namely sloppy quorums and a hinted hand-off protocol. Sloppy quorums allow the system to make progress on write operations even in the presence of failures in the replication group. Thus, if a client issues a write operation but there are not sufficient available replicas to complete the operation, nodes that do not replicate the data being updated are used to complete the operation. Eventually, Dynamo will move the data from the non-responsible nodes to the actual replicas using a hinted hand-off protocol. The Dynamo replication scheme is flat. Dynamo uses vector clocks to track causal dependencies within the replication group of each key. A vector clock contains one entry for each replica (thus the size of clocks grows linearly with the number of replicas). This metadata is kept in the data store, close to the data object, but is never transmitted to the clients. Therefore, when a client contacts a replica it has no way to assess if the replica has a state that is consistent with its past observations. Thus, it is possible for a client to issue a read operation and obtain a version that does not include the client's previous write operations (thus lacking session guarantees [48]). The purpose of the metadata is to detect conflicting updates and to be used by programmers in coding the reconciliation function, using some ad-hoc, application specific, approach. This very weak form of consistency, known as *eventual consistency*, makes it difficult to reason about and can lead to some known anomalies, such as the Amazon's shopping cart anomalies[12]. Nevertheless, its usability has been widely proven for some applications, namely through several commercial systems, such as Riak[2], which have successfully created a business around Dynamo's ideas.

COPS Due to the difficulty of building applications on top of eventually consistent data stores, the community has tried to achieve stronger consistency guarantees, such as session guarantees [48] or causal consistency[41], without hindering availability. More recently, in the context of geo-replicated cloud data stores, there has been a noticeable effort on providing causal consistency, which is widely regarded as the strongest consistency level that a system can provide without compromising availability and incurring high latencies[34]. COPS[33], a scalable causally consistent geo-replicated data store, revived the interest of the community for causally consistent cloud data stores (in fact, COPS offers a slightly stronger consistency criteria, dubbed causal+ consistency but this difference is not critical to our discussion; the reader may refer to [33] for a rigorous definition). COPS uses a hierarchical approach to manage replication: it assumes that some intra data center replication scheme ensures linearizability, such that replication within the data center can be completely masked from the inter data center protocol; COPS addresses the latter. To avoid the inconsistency issues of Dynamo, COPS requires clients to keep metadata and to exchange this metadata every time they access the cloud store. For this purpose, COPS assigns a scalar clock to each object. Clients are required to maintain in their metadata the last clock value of all objects read in the causal past. This metadata, representing their causal history, is purged after every update operation, due to the transitivity property of the happens-before relationship [30]. Thus, the dependencies of an update operation are those immediately previous to the update operation just issued, and the subsequent reads up to the newly issued update operation. This set of dependencies is known as *nearest dependencies*. As long as the client sticks to the same data center, its causal dependencies are automatically satisfied. Nevertheless, updates piggyback their dependencies when being propagated to other data centers, similarly to lazy replication[29]. When a data center receives an update propagated by another data center, it only makes it visible when its dependencies are satisfied. In addition, COPS provides a partial form of transactions called causally consistent read-only transactions. As its name clearly suggests, these are transactions with only read accesses that return versions of the read objects that belong to a causally consistent snapshot. To implement this construct, the authors propose a two-round protocol. In the first round, the client issues a concurrent read operation per key in the read set. Each retrieved object version has a list of dependencies associated. In case any conflicting version is returned (whose dependencies are not satisfied consistently), a second round of read operations is used to retrieve non-conflicting versions. Two object versions are free of conflicts if the objects do not depend on one another, or, if they do so, the version retrieved is equal or greater than the one in the dependencies list. Notice that the conflicting versions may appear only due to concurrent updates to the read-only transaction. The authors argue that in order to correctly identify conflicting versions, the *nearest dependencies* are not sufficient. Therefore, the system needs to store the full list of dependencies and piggyback them when clients issue reads within transactions. This clearly slows down the system and may even make the system unusable due to high latencies in processing and sending large chunks of dependencies over the network.

Chain Reaction In COPS, the fact that causal consistency is provided to the clients is not exploited to increase concurrency within a data center, given that the protocols assume linearizability. ChainReaction [4] exploits this opportunity by proposing a modified chain replication technique that allows for concurrent causally consistent reads within a data center. For this purpose ChainReaction slightly augments the metadata maintained by clients, with two logical clock values per data object: the first value is a global Lamport's clock and the second value is a hint on which local replicas can provide consistent reads. ChainReaction also explores an interesting trade-off when offering causally consistent read-only transactions: it introduces a logical serialization point in each data center (slightly reducing the concurrency of updates) to make it easier to identify consistent snapshots and avoid the need for multiple read-phases when implementing read-transactions.

GentleRain A recently proposed system that uses physical clocks in its design is GentleRain [19]. The main contributions of GentleRain are to reduce the metadata piggybacked on updates propagation to almost zero and to eliminate dependency checking procedures. The idea is to only allow a data center to make a remote update visible once all partitions (whithin the data center) have seen all updates up to the remote update time stamp. Thus, a client that reads a version is automatically ensured to read causally consistent versions in subsequent reads without the need of explicitly checking dependencies or being forced to wait until a causally consistent version is ready.

GentleRain uses physical clocks in its implementation for several purposes: (i) timestamp updates, (ii) detect conflicting versions, (iii) generate read snapshot time for read-only transactions, and (iv) compute the global stable clock. Clients only need to keep track of the latest global stable clock seen in order to ensure across-objects causality. The global stable clock is the minimum physical clock known among all the partitions and data centers, and it is local to each partition. Since, two partitions may have different global stable clocks, GentleRain relies on the clients to convey sufficient information to avoid causality violations. GentleRain is able to achieve performance results similar to eventually consistent systems. Notice that physical clocks could be substituted by simple scalar logical clocks without compromising correctness. Nevertheless, this could cause clocks to advance at very different rates in different partitions, causing the global stable clock to move slowly. This would increase the latency for the visibility of updates to data. Furthermore, it would generate staler read snapshot times for read-only transactions

ORBE ORBE [18] is a partitioned geo-replicated data store that proposes a solution based on the joint use of physical and logical clocks. ORBE uses vector clocks, organized as a matrix, to represent dependencies. The vector clock has an entry per partition and data center. Thus, multiple dependencies on data items of the same partition and data center are represented by only one scalar, which allows for reducing the size of the metadata piggybacked to messages. In addition, similarly to ChainReaction, ORBE is able to optimize the read-only transactional protocol to complete in only one round by relying on physical clocks. Physical clocks are used to generate read snapshot times. Thus, upon receiving a transaction request, the coordinator of the transaction sets the transaction's snapshot time to its physical clock. In consequence, the reads of the transaction have a version upper bound, the snapshot time, which is used to avoid retrieving versions that would violate causality due to concurrent update operations, as in COPS.

5 Clocks for Strong Consistency

In this section, we address systems that provide strong semantics with general purpose transactions, supporting read and write operations to arbitrary objects in the data store, with the guarantee that they will be accessed atomically and isolated from concurrent accesses.

GMU and SCORe The ability of scaling out data stores in large cloud deployments is highly affected by the cost of synchronizing data among the different replicas. This is particularly true for the case of strongly consistent data stores, which require synchronous interactions among all the replicas maintaining data manipulated by a transaction. In order to reduce this cost, a widely adopted strategy is to use partial replication techniques, i.e., to limit the nodes r that replicate each data item to a subset of all the nodes m (with $r \ll m$). One additional characteristic that is particularly desirable in large-scale systems is that of Genuine Partial Replication (GPR) [43], i.e., ensuring that a transaction involves coordination of only the nodes s that replicate the data items accessed (where typically $s \ll m$). We note that the GPR property rules out non-scalable designs that rely on a centralized time source[7], or that force the synchronous advancement of clocks on all nodes of the system upon the commit of any (update) transaction [46]. Two recent examples of GPR data stores that rely on logical clocks to ensure strong consistency are GMU [38] and SCORe [40]. Both systems implement a fully-decentralized multiversion concurrency control algorithms, which spares read-only transactions from the cost of remote validations and from the possibility of aborting. As for update transactions, GMU and SCORe can be characterized as Deferred Update Replication [45] protocols, since writes are buffered at the transaction's coordinator and sent out during the commit operation to be applied and made visible to other transactions in an atomic step.

The two systems differ, however, in the way they reify time. GMU uses a vector clock of size m (one entry per node) whereas SCORe uses a single scalar clock. These clocks are used to assign the read and commit timestamps to transactions, as well as to tag the various versions of each data item. In both systems, the read timestamp of a transaction is used to establish a consistent snapshot (i.e., the set of visible data item versions) over the partially replicated data set. In SCORe, the start timestamp is assigned to a transaction upon its first read operation in some node. From that moment on, any subsequent read operation is allowed to observe the most recent committed version of the requested datum having timestamp less than or equal to the start timestamp (as in classical multiversion concurrency control algorithms). In contrast, for GMU, whenever a transaction T reads for the *first* time on a node, it tries to advance the start timestamp (a vector clock) to extend the visible snapshot for the transaction. This can allow, for instance, to include in T's snapshot the versions created by a transaction T' that have involved a node n not contacted so far by T, and whose commit events would not be visible using T's initial read-timestamp (as the n-th entry of the commit timestamp of T' is higher than the corresponding entry of T's read-timestamp).

Another relevant difference between these two systems lies in the consistency guarantees that they provide: SCORe provides 1-Copy Serializability (1CS), whereas GMU only Update Serializability (US) [3], a consistency criterion that allows read-only transactions to serialize the commit events of non-conflicting update transactions in different orders — an anomaly only noticeable by users of the systems if they communicate using some external channel, which still makes US strictly weaker than classic 1CS. In fact, both protocols guarantee also the serializability of the snapshot observed by transactions that have to be aborted (because their updates are not reconcilable with those of already committed transactions). This additional property is deemed as important in scenarios in which applications process the data retrieved by the store in an non-sandboxed environment (e.g., in a Transactional Memory [15]), in which exposing data anomalies to applications may lead to unpredictable behaviors or faults.

Clock-SI Clock-SI[17] assumes loosely synchronized clocks that only move forward, but that can be subject to possibly arbitrary skews. Clock-SI[17] provides the so-called Snapshot Isolation[8] consistency criterion, in which read-only transactions read from a consistent (possibly versioned) snapshot, and other transactions commit if no object written by them was also written concurrently. Each transaction identifies its read timestamp simply by using its local physical clock. In order to ensure safety in spite of clocks skews, Clock-SI resorts to delaying read operations in two scenarios: i) the read operation is originated by a remote node whose clock is in the future with respect to the physical clock of the node that processes the read request; ii) the read operation targets a data item for which a pending pre-committed version exists and whose commit timestamp may be smaller than the timestamp of the read operation.

Spanner Spanner [5] provides a stronger consistency guarantee than the data stores discussed so far, namely external consistency (also known as strict serializability). The key enabler behind Spanner is the TrueTime API. Briefly, clock skew is exposed explicitly in TrueTime API by representing the current time as an interval. This interval is guaranteed to contain the absolute time at which the clock request was invoked. In fact, multiple clock references (GPS and atomic clocks) are used to ensure strict, and generally small, bounds on clock uncertainty (less than 10 ms).

Spanner assigns a timestamp s to (possibly read-only) transactions by reading the local physical clock. Similarly, it also assigns a commit timestamp c to update transactions once that they acquire all the locks over the write-set w, which is then used to version the new data items. Spanner blocks read operations associated with a physical timestamp t in case it cannot yet be guaranteed that no new update transaction will be able to commit with a timestamp $t' < t$. This is achieved by determining a, so called, *safe time* t_{safe} that is lower bounded by the physical timestamp of the earliest prepared transaction at that node. Unlike Clock-SI, though, Spanner also introduces delays in commit operations: the write-set w of a transaction with commit timestamp c is made visible only after having waited out the uncertainty window associated with c. This is sufficient to ensure that the side-effects of a transaction are visible solely in case timestamp c is definitely in the past, and that any transaction that reads or over-writes data in w is guaranteed to be assigned a timestamp greater than c.

CockroachDB The waiting phases needed by the data stores based on physical clocks can have a detrimental effect on performance, especially in the presence of large skews/uncertainties in the clock synchronization. This has motivated the investigation of techniques that use in conjunction both logical and physical clocks, with the aim to allow the correct ordering of causally related transactions without the need of injecting delays for compensating possible clock skews. Augmented-Time (AT) [13], for instance, uses a vector clock that tracks the latest known physical clock values of any node with which communication took place (even transitively) in the current uncertainty window. The assumption on the existence of a bounded skew among physical clocks allows to trim down the size of the vector clocks stored and disseminated by nodes. Also, by relying on vector clocks to track causality relations within the uncertainty interval, AT ensures external consistency for causally related transactions (e.g,. if a transaction T2 reads data updated by transaction T1, T2's commit timestamp will be greater than T1's) without the need for incurring Spanner's commit wait phases.

Real-time order (or external consistency) cannot however be reliably enforced between transactions whose uncertainty intervals overlap and which do not develop any (possibly transitive) causal dependency. In such cases, waiting out the uncertainty interval (provided that reasonable bounds can be assumed for it) remains necessary to preserve external consistency. Hybrid Logical Clocks (HLC) [28] represent a recent evolution of the same idea, in which physical clocks are coupled with a scalar logical clock with the similar goal of efficiently ordering causally related transactions whose uncertainty intervals overlap. Roughly speaking[1], with HLC, whenever a node n receives a message timestamped with a physical clock value, say pt', greater than the maximum currently heard of among the nodes, say pt, n sets $pt = pt'$ and uses the updated value of pt as if it were a classic scalar logical clock in order to propagate causality information among transactions.

HLC is being integrated in CockroachDB[1], an open-source cloud data store (still under development at the time of writing) that, similarly to Spanner, relies on physical clocks to serialize transactions (ensuring either Snapshot Isolation or Serializable Snapshot Isolation[20]). Unlike Spanner, however, CockroachDB does not assume the availability of specialized hardware to ensure narrow bounds on clock synchronization, but relies on conventional NTP-based clock synchronization that frequently imposes clock skews of several tens of milliseconds. HLC is hence particularly beneficial in this case, at it allows for ensuring external consistency across causally related transactions while sparing from the costs of commit waits.

6 Discussion

We now draw some observations, based on the systems analyzed in the previous sections.

6.1 It is Unclear How Well External Consistency May Scale

Ensuring external consistency in a distributed data store without sacrificing scalability (e.g., using a centralized time source, or tracking the advancement of logical time at all nodes whenever a transaction commits) is a challenging problem. GPR systems based on logical clocks, like SCORe or GMU, can accurately track real-time order relations *only* between causally related transactions — a property that goes also under the name of witnessable real-time order [39]). Analogous considerations apply to system, like Clock-SI, that rely on physical clocks and assume unbounded clock-skews (and monotonic clock updates). The only GPR system we know of that guarantees external consistency is Spanner, but this ability comes with two non-negligible costs: 1) the reliance on specialized hardware to ensure tight bounds on clock uncertainty; and 2) the introduction of delays upon the critical path of execution of transactions, which can have a detrimental impact on performance. As already mentioned, the usage of hybrid clocks, like HLC or AT, can be beneficial to reduce the likelihood of blocking due to clock uncertainty. However, we are not aware of any GPR data store that uses hybrid clocks and loosely-synchronized physical clocks (i.e., subject to unbounded skew), which can consistently capture real-time order among non-causally related transactions, e.g., two transactions that execute sequentially on different nodes and update disjoint data items. In the light of these considerations, there appears to exist a trade-off between the strictness of the real-time ordering guarantees that can be ensured by a scalable transactional data store and the necessity of introducing strong assumptions on clock synchronization.

More in general, the analysis of these systems raises interesting theoretical questions such as: which forms of real-time order can be guaranteed by scalable (i.e., GPR) cloud data store implementations that use exclusively logical clocks? Does the use of loosely synchronized physical clocks, possibly complemented by logical

[1]A more sophisticated algorithm using two scalar logical clocks is also proposed in order to guarantee the boundedness of the information stored in the logical timestamps and that the divergence between physical and logical clocks is bounded — which allows for using logical clocks as a surrogate of physical clocks, e.g., to use logical clocks to obtain consistent snapshots at "close" physical times.

clocks, allow for providing stronger real-time guarantees? Would relaxing the progress guarantees for read-only transactions (e.g., lock-freedom in systems like Spanner or GMU) allow for strengthening the real-time ordering guarantees that a GPR system can guarantee? One may argue that the above questions appear related to some of the theoretical problems studied by the parallel/concurrent computing literature, which investigated the possibility of building scalable Transactional Memory implementations [23, 47, 14, 24] that guarantee a property akin to GPR, namely Disjoint Access Parallelism (DAP) [25, 5, 10, 39]: roughly speaking, in a DAP system two transactions that access disjoint data items cannot contend for any shared object (like logical clocks). However these works considered a different system model (shared-memory systems) and neglected the usage of physical and hybrid clocks. Understanding whether those results apply to the case of distributed data stores (i.e., message-passing model) using different clock types remains an interesting open research problem.

6.2 The Trade-offs Between Logical and Physical Clocks Are Not Completely Understood

A key trade-off that emerges when using logical clocks in a strongly consistent GPR cloud data store is that the advancement of clocks across the various nodes of the system is dependent on the communication patterns that are induced by the data accesses issued by the user-level applications. As such, systems based on logical clocks may be subject to pathological scenarios in which the logical clocks of a subset s of the platform's nodes can lag significantly behind with respect to the clocks of other nodes that commit update transactions without accessing data maintained on s. In this case, the transactions originated on the nodes in s are likely to observe stale data and incur higher abort rates. The mechanism typically suggested to tackle this issue is to rely on some asynchronous, epidemic dissemination of updates to the logical clocks. Such techniques can at least partially address the issue, but they introduce additional complexity (as the optimal frequency of dissemination is a non-trivial function of the platform's scale and workload) and consume additional network bandwidth.

Clearly, physical clocks — which advance spontaneously — avoid this issue. On the other hand, in order to compensate for clock skews, data stores that rely on physical clocks have to inject delays proportional to the uncertainty intervals in critical phases of the transactions' execution. As already discussed in various works[5, 1, 28, 13], these delays can severely hamper performance unless specialized, expensive hardware is used to ensure tight bounds on the uncertainty intervals (as in Spanner). In this sense, the idea at the basis of hybrid clocks, and in particular of approaches like HLC [28], appears quite interesting, since it attempts to combine the best of both worlds. On the other hand, research in this area is still at its infancy and the only cloud data store we are aware of that is using hybrid clocks is CockroachDB, which is still in its early development stages. It is hence unclear to what extent hybrid clocks can alleviate the need for injecting artificial delays in solutions based on physical clocks. Indeed, we advocate the need for conducting a rigorous and systematic experimental analysis aimed at identifying under which conditions (e.g., platform's scale, maximum clock synchronization skew, workloads) each of the various approaches provides optimal efficiency. The results of such a study would be valuable to guide practitioners in the selection of the data store best suited to their application needs. They may also provide motivations to foster research in the design of autonomic systems aimed at adapting the type of clock implementation to pursue optimal efficiency in face of dynamic workload conditions and/or platform's scale (e.g., following an elastic re-scaling of the platform).

6.3 Time is a Resource That Can be Traded for Other Resources

As we have noted earlier, the end-user experience can be negatively affected by the latency of the cloud service. Therefore, solutions that require waiting, as some of the previously described protocols based on the use of physical clocks, have to be used carefully. On the other hand, time is just one of the many resources that can be a source of bottlenecks in the operation of a cloud data store. Large metadata structures consume disk space and increase network bandwidth utilization, within the data center, across data centers, and between the data center and the client. In turn, heavy disk and network utilization are also a major source of delays and instability.

Since solutions based on physical clock often allow for significant metadata savings, the overall performance of a given algorithm is hard to predict, and strongly depends on which resources the system is more constrained.

For instance, although GentleRain is able to reduce dependencies metadata to almost zero, it also has a disadvantage: clients are likely to read stale data. Since remote updates visibility is delayed, we may end up with latencies of up to seconds (depending on the geo-replicated deployment and network conditions) before an update is visible to other clients. On the other hand, the system will have performance almost similar to an eventually consistent data store since the metadata is almost negligible and no dependence checking mechanism is required. Thus, depending on application requirements, one could give preference to freshness of data or performance.

Another example is how ORBE uses time to avoid multiple rounds of messages to implement read-only transactions. In this case, loosely synchronized physical clocks are used as a decentralized solution for implementing one round causally consistent read-only transactions. However, this may affect negatively the latency of operations. A partition that receives a read operation with snapshot time st has to first wait until its physical clock catches up with st, and also to make sure that updates up to that time from other replicas have been received.

GMU and SCORe also exemplify this trade-off. By using vector clocks, GMU can advance the transaction's read snapshot more frequently than SCORe, which relies on a scalar clock. The intuition is that, since SCORe needs to capture causal dependencies among transactions via a single scalar clock, it is sometimes forced to over-approximate such relations, which leads to observing obsolete snapshots. This impacts the freshness of the data snapshots observed by transactions, an issue that is particularly relevant for update transactions, which can be subject to spurious aborts as a consequence of unnecessarily reading obsolete data versions. On the other side of the coin, in both systems, logical timestamps are piggybacked to every message, both for read operations (to identify the read timestamp), as well as during the distributed commit phase (to obtain the final commit timestamp). As such, SCORe naturally incurs less overheads by sending a single scalar clock with each message, in contrast with GMU's vector clocks that grow linearly with the number of nodes in the deployment. Experiments comparing both systems in a similar infra-structure (both in terms of software as well as in hardware) confirmed that, at least in low conflict workloads, SCORe tends to perform even slightly better than GMU [40], despite providing a stronger consistency criterion — we note that their scalability trends are similar, as expected in workloads with low contention to data. Other workloads may be found on another work relying on SCORe [16].

6.4 Total Order is Expensive but Concurrency Also Comes with a Cost

Most strong consistency models, such as serializability, require concurrent requests to be serialized and this, in practice, means that some form of total order/consensus protocol needs to be implemented in the system. By definition, such protocols materialize a logic serialization point for the system, which can easily turn into a bottleneck. The use of weaker consistency models, which allow for more concurrency, has therefore great appeal. We have already discussed that weak consistency imposes a burden on the programmer of applications that use the cloud data store. Maybe not so obvious is the fact that weak consistency also imposes a metadata burden on the protocols that support the additional level of concurrency. For instance, consider an idealized system where all operations, for all objects, are executed by a single "primary" node, replicated using Paxos [31]. The entire state of the system at a given point can be referred to by a single scalar, the number of the last operation executed by the primary node. On the one hand, the use of a single primary node to serialize all operations is inherently non-scalable, and hence undesirable in a large-scale system. On the other hand, an idealized fully concurrent system supporting partial replication, with no constraints on how updates are propagated among replicas, requires to maintain a matrix clock of size m^2 where m is the total number of nodes in the entire system. Such timestamps would be impossible to maintain and exchange in large scale.

The discussion above shows that fully decentralized (and inherently fully concurrent) systems may be as little scalable as fully centralized (and inherently fully sequential) systems. It is therefore no surprise that, as with the use of different clock implementations, the right trade-offs may heavily depend on concrete factors such as the workload characteristics, the hardware deployment, etc. In consequence, it is hard to define, in general, when one approach outperforms the other.

Consider for instance the different approaches taken by COPS and ChainReaction for different aspects of the system. ChainReaction is geared towards efficient read operations while COPS aims at more balanced workloads. For instance, ChainReaction is willing to slightly increase the metadata size to allow for additional concurrency while reading objects within a data center, contrary to COPS that enforces linearizabilty at each data center. Also ChainReaction serializes updates within the data center to optimize read-only transactions while COPS does not serialize updates to distinct objects but may pay the cost of multi-phase read-only transactions. Similarly, ORBE implements a number of metadata reduction techniques, that may create false dependencies among operations, thus, achieving metadata savings at the cost of concurrency loss.

7 Summary

In this work we focused on one common aspect that is crucial to any distributed cloud data store, i.e., the mechanisms employed to capture the passage of time and to keep track of the cause-effect relations among different data manipulations.

We analyzed existing literature in the area of cloud data stores from two antagonistic perspectives, those of low latency intra data center communication and geo-replicated systems. These two complementary deployments of machines create the opportunity to explore both strongly consistent and weakly consistent systems. This is, in part, a consequence of the design of modern data centers that allow strong consistent systems to perform fast, whereas for inter data center replication we can effectively use weak consistent systems (in particular with causal consistency) while still providing meaningful semantics and guarantees to the programmer.

Our analysis allowed us to derive some observations on the implications of various design choices at the basis of existing cloud data stores. These insights may not only help system architects to select the solutions that best fit their needs, but also identify open research questions that may offer opportunities to enhance the next generations of cloud data stores.

References

[1] Cockroach. https://github.com/cockroachdb/cockroach.

[2] Riak. http://basho.com/riak/.

[3] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, 1999. AAI0800775.

[4] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[5] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory of Computing Systems*, 49(4):698–719, 2011.

[6] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32, March 2013.

[7] Roberto Baldoni, Carlo Marchetti, and Sara Tucci Piergiovanni. A fault-tolerant sequencer for timed asynchronous systems. In *Euro-Par 2002 Parallel Processing*, pages 578–588. Springer, 2002.

[8] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[9] Eric A Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[10] Victor Bushkov, Dmytro Dziuma, Panagiota Fatourou, and Rachid Guerraoui. Snapshot isolation does not scale either. Technical report, Technical Report TR-437, Foundation of Research and Technology–Hellas (FORTH), 2013.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[13] Murat Demirbas and Sandeep Kulkarni. Beyond truetime: Using augmented time for improving Google Spanner, 2013.

[14] Nuno Diegues and Paolo Romano. Time-warp: lightweight abort minimization in transactional memory. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*,167–178. 2014.

[15] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 3–14, 2014.

[16] Nuno Lourenço Diegues and Paolo Romano. Bumper: Sheltering transactions from conflicts. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems*, SRDS '13, pages 185–194, Washington, DC, USA, 2013. IEEE Computer Society.

[17] Jiaqing Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 173–184, Sept 2013.

[18] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[19] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13, New York, NY, USA, 2014. ACM.

[20] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[21] Colin J Fidge. *Timestamps in message-passing systems that preserve the partial ordering*. Australian National University. Department of Computer Science, 1987.

[22] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[23] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[24] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[25] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, 151–160. ACM, 1994.

[26] Caroline Jay, Mashhuda Glencross, and Roger Hubbold. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Trans. Comput.-Hum. Interact.*, 14(2), August 2007.

[27] Sandeep S Kulkarni et al. Stabilizing causal deterministic merge. In *Self-Stabilizing Systems*, pages 183–199. Springer, 2001.

[28] SandeepS. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In MarcosK. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems*, volume 8878 of *Lecture Notes in Computer Science*, pages 17–32. Springer International Publishing, 2014.

[29] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, November 1992.

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

[31] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[32] Greg Linden. Make data useful. *Presentation, Amazon, November*, 2006.

[33] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[34] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. *University of Texas at Austin Tech Report*, 11, 2011.

[35] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[36] David L Mills. A brief history of ntp time: Memoirs of an internet timekeeper. *ACM SIGCOMM Computer Communication Review*, 33(2):9–21, 2003.

[37] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[38] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 455–465, June 2012.

[39] Sebastiano Peluso, Roberto Palmieri, Paolo Romano, Binoy Ravindran, and Francesco Quaglia. Brief announcement: Breaching the wall of impossibility results on disjoint-access parallel tm. *Distributed*, page 548.

[40] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 456–475, New York, NY, USA, 2012. Springer-Verlag New York, Inc.

[41] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 288–301, New York, NY, USA, 1997. ACM.

[42] Sunil K. Sarin and Nancy A. Lynch. Discarding obsolete information in a replicated database system. *Software Engineering, IEEE Transactions on*, (1):39–47, 1987.

[43] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine partial replication in wide area networks. In *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, pages 214–224, Washington, DC, USA, 2010. IEEE Computer Society.

[44] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*, June 2009.

[45] Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. Scalable deferred update replication. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[46] Damián Serrano, Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Bettina Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 290–297. IEEE, 2007.

[47] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[48] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*, pages 140–149. IEEE, 1994.

[49] Gene TJ Wuu and Arthur J Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 233–242. ACM, 1984.

Scaling Optimistic Concurrency Control by Approximately Partitioning the Certifier and Log

Philip A. Bernstein
Microsoft Research
Redmond, WA, USA
phil.bernstein@microsoft.com

Sudipto Das
Microsoft Research
Redmond, WA, USA
sudipto.das@microsoft.com

Abstract

In optimistic concurrency control, a certifier algorithm processes a log of transaction operations to determine whether each transaction satisfies a given isolation level and therefore should commit or abort. This logging and certification of transactions is often sequential and can become a bottleneck. To improve transaction throughput, it is beneficial to parallelize or scale out the certifier and the log. One common technique for such parallelization is to partition the database. If the database is perfectly partitioned such that transactions only access data from a single partition, then both the log and the certifier can be parallelized such that each partition has its own independent log and certifier. However, for many applications, partitioning is only approximate, i.e., a transaction can access multiple partitions. Parallelization using such approximate partitioning requires synchronization between the certifiers and logs to ensure correctness. In this paper, we present the design of a parallel certifier and a partitioned log that uses minimal synchronization to obtain the benefits of parallelization using approximate partitioning. Our parallel certifier algorithm dynamically assigns constraints to each certifier. Certifiers enforce constraints using only atomic writes and reads on shared variables, thus avoiding expensive synchronization primitives such as locks. Our partitioned log uses a lightweight causal messaging protocol to ensure that transactions accessing the same partition appear in the same relative order in all logs where they both appear. We describe the techniques applied to an abstract certifier algorithm and log protocol, making them applicable to a variety of systems. We also show how both techniques can be used in Hyder, a scale-out log-structured indexed record manager.

1 Introduction

Optimistic concurrency control (OCC) is a technique to analyze transactions that access shared data to determine which transactions commit or abort [14]. Instead of delaying certain operations that might lead to an incorrect execution, OCC allows a transaction to execute its operations as soon as it issues them. After the transaction finishes, OCC determines whether the transaction commits or aborts. A *certifier* is the component that makes this determination. It is a sequential algorithm that analyzes descriptions of the transaction one-by-one in a given total order. Each transaction description, called an *intention*, is a record that describes the operations that the transaction performed on shared data, such as read and write. One way to determine the total order of intentions is to store them in a *log*. In that case, the certifier analyzes intentions in the order they appear in the log.

A certifier algorithm has throughput limits imposed by the underlying hardware [7]. This limits the scalability of a system that uses it. To improve the throughput, it is worthwhile to parallelize the algorithm. One way to do this is to split the set of transactions into partitions such that for every pair of transactions from different partitions, there are no conflicts between them. Then the certifier can run independently on each partition. However, it is often infeasible to partition transactions in this way. In that case, the certifier algorithm needs to handle transactions that span more than one partition. This paper presents such an algorithm.

The log also has throughput limits imposed by the hardware. Thus, a second opportunity for improving throughput is to partition the log, such that each partition includes updates that apply to a distinct database partition. This enables the log to be distributed over independent storage devices to provide higher aggregate throughput of read and append operations to the log. However, if the partitioning is imperfect, some transactions need to appear in two or more partitions. In this case, the log partitioning must ensure that conflicting transactions appear in the same relative order in all logs where they both appear. This paper presents a way of generating a log partitioning that satisfies this property.

The goal of these these two techniques—parallelizing a certifier and partitioning a log—is to increase transaction throughput. Our motivation for designing these techniques is to increase the throughput of our Hyder system, a database architecture that scales out without partitioning [8]. In Hyder, the log *is* the database, which is represented as a multi-version binary search tree. Each transaction T executes on a snapshot of the database and generates an intention record that contains T's writeset and, depending on the isolation level, its readset. The intention is stored in the log. A certification algorithm, called *meld* [9], reads intentions from the log and sequentially processes them in log order to determine whether a transaction committed or aborted. If a transaction commits, meld does one more step beyond OCC certification, namely, it merges the transaction's updates into the server's locally-cached copy of the database. Since all servers receive the same log, meld makes the same commit and abort decisions for every transaction. Therefore, for any two servers, their locally-cached copies of the database are identical for any data that is stored in both of them. Since there is no synchronization between the servers apart from appending to and reading from the shared log, the system scales out. That is, throughput increases as more servers are added, until the log, network, or meld algorithm is saturated. Often, the meld algorithm is the bottleneck. This was demonstrated in [6] by experiments with a distributed implementation of Hyder on a cluster of enterprise-grade commodity servers. It is therefore important to parallelize meld to increase transaction throughput. Bernstein et al. [6] describes two approaches that use pipeline parallelism to speed up meld; it introduces two preliminary stages that reduce the work done by the final sequential meld algorithm. In this paper, we leverage database partitioning to parallelize the meld algorithm itself.

Organization: We formally define the problem in Section 2 and then present the algorithms for parallel certification (Section 3) and log partitioning (Section 4). In Section 5, we revisit the question of how to apply these parallel solutions to Hyder. Section 6 summarizes related work and Section 7 is the conclusion.

2 Problem Definition

The certifier's analysis relies on the notion of conflicting operations. Two operations *conflict* if the relative order in which they execute affects the value of a shared data item or the value returned by one of them. The most common examples of conflicting operations are read and write, where a write operation on a data item conflicts with a read or write operation on the same data item. Two transactions conflict if one transaction has an operation that conflicts with at least one operation of the other transaction.

To determine whether a transaction T commits or aborts, a certifier analyzes whether any of T's operations conflict with operations issued by other concurrent transactions that it previously analyzed. For example, if two transactions executed concurrently and have conflicting accesses to the same data, such as independent writes of a data item x or concurrent reads and writes of x, then the algorithm might conclude that one of the transactions must abort. Different certifiers use different rules to reach their decision. However, all certifiers have one property in common: their decision depends in part on the relative order of conflicting transactions.

We define a *database partitioning* to be a set of partition names, such as $\{P_1, P_2, \ldots\}$, and an assignment of every data item in the database to one of the partitions. A database partitioning is *perfect* with respect to a set of transactions $T = \{T_1, T_2, \ldots\}$ if every transaction in T reads and writes data in at most one partition. That is, the database partitioning induces a transaction partitioning. If a database is perfectly partitioned, then it is trivial to parallelize the certifier and partition the log: For each partition P_i, create a separate log L_i and an independent execution C_i of the certifier algorithm. All transactions that access P_i append their intentions to L_i, and C_i takes L_i as its input. Since transactions in different logs do not conflict, there is no need for shared data or synchronization between the logs or between executions of the certifier on different partitions.

A perfect partitioning is not possible in many practical situations, so this simple parallelization approach is not robust. Instead, suppose we can define a database partitioning that is *approximate* with respect to a set of transactions T, meaning that most transactions in T read and write data in at most one partition. That is, some transactions in T access data in two or more partitions (so the partitioning is not perfect), but most do not.

In an approximate partitioning, the transactions that access only one partition can be processed in the same way as a perfect partitioning. However, transactions that access two or more partitions make it problematic to partition the certifier. The problem is that such multi-partition transactions might conflict with transactions that are being analyzed by different executions of the certifier algorithm, which creates dependencies between these executions. For example, suppose data items x and y are assigned to different partitions P_1 and P_2, and suppose transaction T_i writes x and y. Then T_i must be evaluated by C_1 to determine whether it conflicts with concurrent transactions that accessed x and by C_2 to determine whether it conflicts with concurrent transactions that accessed y. These evaluations are not independent. For example, if C_1 determines that T_i must abort, then that information is needed by C_2, since C_2 no longer has the option to commit T_i. When multiple transactions access different combinations of partitions, such scenarios can become quite complex.

A transaction that accesses two or more partitions also makes it problematic to partition the log, because its intentions need to be ordered in the logs relative to all conflicting transactions. Continuing with the example of transaction T_i above, should its intention be logged on L_1, L_2, or some other log? Wherever it is logged, it must be ordered relative to all other transactions that have conflicting accesses to x and y before it is fed to the OCC algorithm. The problem we address is how to parallelize the certifier and partition the log relative to an approximate database partitioning. Our solution takes an approximate database partitioning, an OCC algorithm, and an algorithm to atomically append entries to the log as input. It has three components:

1. Given an approximate database partitioning $P = \{P_1, P_2, \ldots, P_n\}$, we define an additional *logical partition* P_0. Each transaction that accesses only one partition is assigned to the partition that it accesses. Each transaction that accesses two or more partitions is assigned to the master logical partition P_0.
2. We parallelize the certifier algorithm into $n + 1$ parallel executions $\{C_0, C_1, C_2, \ldots, C_n\}$, one for each partition, including the logical partition. Each single-partition transaction is processed by the certifier execution assigned to its partition. Each multi-partition transaction is processed by the logical partition's execution of the certifier algorithm. We define synchronization constraints between the logical partition's certifier execution and the partition-specific certifier executions so they reach consistent decisions.
3. We partition the log into $n + 1$ distinct logs $\{L_0, L_1, L_2, \ldots, L_n\}$, one associated with each partition and one associated with the logical partition. We show how to synchronize the logs so that the set of all intentions across all logs is partially ordered and every pair of conflicting transactions appears in the same relative order in all logs where they both appear. Our solution is a low-overhead sequencing scheme based on vector clocks.

Our solution works with any approximate database partitioning. Since multi-partition transactions are more

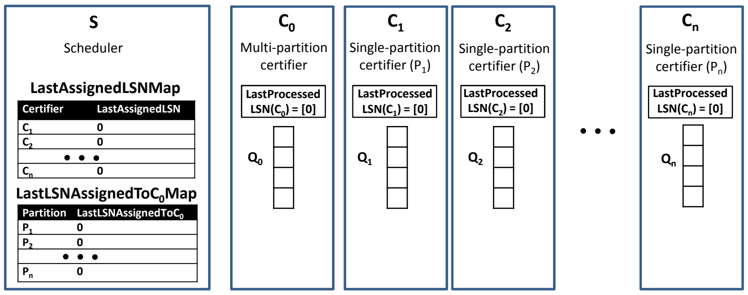

S
Scheduler
LastAssignedLSNMap
Certifier LastAssignedLSN
C1 0
C2 0
Cn 0
LastLSNAssignedToC0Map
Partition LastLSNAssignedToC0
P1 0
P2 0
Pn 0
C0
Multi-partition certifier
LastProcessed LSN(C0) = [0]
Q0
C1
Single-partition certifier (P1)
LastProcessed LSN(C1) = [0]
Q1
C2
Single-partition certifier (P2)
LastProcessed LSN(C2) = [0]
Q2
Cn
Single-partition certifier (Pn)
LastProcessed LSN(Cn) = [0]
Qn

Figure 1: Design overview of parallel certification showing the different certifiers and the data structures used.

expensive than single-partition transactions, the fewer multi-partition transactions that are induced by the database partitioning, the better. The synchronization performed between parallel executions of the certifier algorithm is external to the certifier algorithm. Therefore, our solution works with any certifier algorithm. The same is true for the synchronization performed between parallel logs.

3 Parallel Certification

We now explain the design of a parallel certifier assuming a single totally-ordered log. In this section, we use the term certifier to refer to a certifier execution. A certifier can be parallelized using multiple threads within a single process, multiple processes co-located on the same machine, or multiple processes distributed across different machines; our discussion encompasses all such scenarios. Section 4, entitled "Partitioned Log," explains how the parallel certification of this section can use a partitioned log.

3.1 Design

We dedicate one certifier C_i to process intentions from single-partition transactions on partition P_i, and dedicate one certifier C_0 to process intentions from multi-partition transactions. A single scheduler S processes intentions in log order, assigning each intention to one of the certifiers. The certifiers can process non-conflicting intentions in parallel. However, they must process conflicting intentions in log order.

Our design uses constraints that capture the log order. S passes these constraints to each C_i. The certifiers validate the constraints using atomic reads and writes on shared variables, so the synchronization is efficient. Figure 1 illustrates the design of a parallel certifier showing the different variables and data structures maintained by each C_i, and the data structures used by S to determine synchronization constraints passed to each C_i.

In what follows, for succinctness we frequently use the word "transaction" to mean the intention produced by the transaction. Each intention in the log has a unique location, called its log sequence number, or LSN, which reflects the relative order of intentions in the log. That is, intention Int$_i$ precedes intention Int$_k$ in the log if and only if the LSN of Int$_i$ is less than the LSN of Int$_k$.

Every certifier $C_i(\forall i \in [0, n])$ maintains a variable **LastProcessedLSN(C_i)** that stores the LSN of the last transaction processed by C_i. After C_i processes a transaction T_k, it sets LastProcessedLSN(C_i) equal to T_k's LSN; C_i performs this update irrespective of whether T_k committed or aborted. Every other certifier $C_j(\forall j \neq i)$ can atomically read LastProcessedLSN(C_i) but cannot update it. In our algorithm, each LastProcessedLSN(C_i), $i \in [1, n]$, is read only by C_0 and LastProcessedLSN(C_0) is read by all C_i, $i \in [1, n]$. Each C_i ($i \in [0, n]$) also has an associated producer-consumer queue Q_i where S enqueues the transactions C_i needs to process (i.e., S is the producer for Q_i). Each C_i dequeues the next transaction from Q_i when it completes processing its previous transaction (i.e., C_i is the consumer for Q_i). The scheduler S maintains a local structure, **LastAssignedLSN-Map**, that maps each C_i, $i \in [1, n]$), to the LSN of the last single-partition transaction it assigned to C_i. S

maintains another local structure, **LastLSNAssignedToC_0Map**, that stores a map of each partition P_i to the LSN of the last multi-partition transaction that it assigned to C_0 and that accessed P_i.

Each certifier C_i needs to behave as if it were processing all single-partition and multi-partition transactions that access P_i in log order. This requires that certifiers satisfy the following synchronization constraint:

> **Parallel Certification Constraint:** Before certifying a transaction T that accessed partition P_i, all transactions that precede T in the log and accessed P_i must have been certified.

This condition is trivially satisfied by a sequential certifier. Threads in a parallel certifier must synchronize to ensure that the condition holds. For each transaction T, S determines which certifiers C_i will process T. S uses its two local data structures, LastAssignedLSNMap and LastLSNAssignedToC_0Map, to determine and provide each such C_i with the synchronization constraints it must satisfy before C_i can process T. Note that this constraint is conservative since this strict ordering is essential only for conflicting transactions. However, in the absence of finer-grained tracking of conflicts, this conservative constraint guarantees correctness.

3.2 Synchronizing the Certifier Threads

Let T_i denote the transaction that S is currently processing. We now describe how S generates the synchronization constraints for T_i. Once S determines the constraints, it enqueues the transaction and the constraints to the queue corresponding to the certifier.

Single-partition transactions: If T_i accessed a single partition P_i, then T_i is assigned to the single-partition certifier C_i. C_i must synchronize with C_0 before processing T_i to ensure that the parallel certification constraint is satisfied. Let T_k be the last transaction that S assigned to C_0, that is, LastLSNAssignedToC_0Map(P_i) = k. S passes the synchronization constraint LastProcessedLSN(C_0)$\geq k$ to C_i along with T_i. The constraint tells C_i that it can process T_i only after C_0 has finished processing T_k. When C_i starts processing T_i's intention, it accesses the variable LastProcessedLSN(C_0). If the constraint is satisfied, C_i can start processing T_i. If the constraint is not satisfied, then C_i either polls the variable LastProcessedLSN(C_0) until the constraint is satisfied or uses an event mechanism to be notified when LastProcessedLSN(C_0)$\geq k$.

Multi-partition transactions: If T_i accessed multiple partitions $\{P_{i1}, P_{i2}, \ldots\}$, then S assigns T_i to C_0. C_0 must synchronize with the certifiers $\{C_{i1}, C_{i2}, \ldots\}$ of all partitions $\{P_{i1}, P_{i2}, \ldots\}$ accessed by T_i. Let T_{k_j} be the last transaction assigned to $P_j \in \{P_{i1}, P_{i2}, \ldots\}$, that is, LastAssignedLSNMap(C_j) = k_j. S passes the following synchronization constraint to C_0:

$$\bigwedge_{\forall j : P_j \in \{P_{i1}, P_{i2}, \ldots\}} \text{LastProcessedLSN}(C_j) \geq k_j,$$

The constraint tells C_0 that it can process T_i only after all C_j in $\{C_{i1}, C_{i2}, \ldots\}$ have finished processing their corresponding T_{k_j}'s, which are the last transactions that precede T_i and accessed a partition that T_i accessed. When C_0 starts processing T_i's intention, it reads the variables LastProcessedLSN(C_j) $\forall j : P_j \in \{P_{i1}, P_{i2}, \ldots\}$. If the constraint is satisfied, C_0 can start processing T_i. Otherwise, C_0 either polls the variables LastProcessedLSN(C_j) $\forall j : P_j \in \{P_{i1}, P_{i2}, \ldots\}$ until the constraint is satisfied or uses an event mechanism to be notified when the constraint is satified.

Notice that for all j such that $P_j \in \{P_{i1}, P_{i2}, \ldots\}$, the value of the variable LastProcessedLSN(C_j) increases monotonically over time. Thus, once the constraint LastProcessedLSN(C_j)$\geq k_j$ becomes true, it will be true forever. Therefore, C_0 can read each variable LastProcessedLSN(C_j) independently, with no synchronization. For example, it does not need to read all of the variables LastProcessedLSN(C_j) within a critical section.

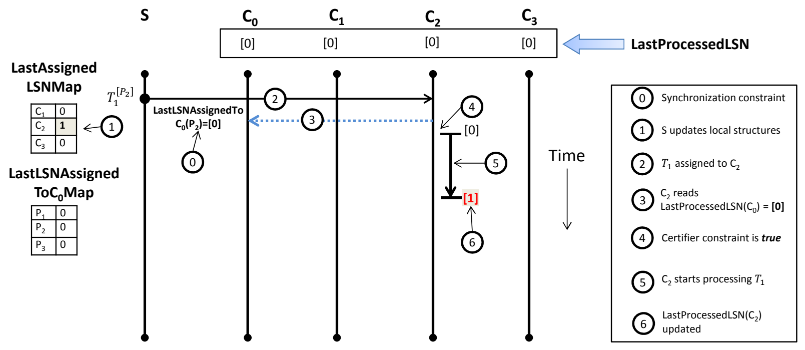

S
C0
C1
C2
C3
[0]
[0]
[0]
[0]
LastProcessedLSN
LastAssigned LSNMap
C1 0
C2 1
C3 0
LastLSNAssigned ToC0Map
P1 0
P2 0
P3 0
LastLSNAssignedTo C0(P2)=[0]
[0]
[1]
Time
0 Synchronization constraint
1 S updates local structures
2 T1 assigned to C2
3 C2 reads LastProcessedLSN(C0) = [0]
4 Certifier constraint is true
5 C2 starts processing T1
6 LastProcessedLSN(C2) updated

Figure 2: An example of the parallel certifier processing a single-partition transaction that accessed partition P_2.

3.3 An Example

Consider a database with three partitions P_1, P_2, P_3. Let C_1, C_2, C_3 be the parallel certifiers assigned to P_1, P_2, P_3 respectively, and let C_0 be the certifier responsible for multi-partition transactions. In this example, we consider the following sequence of transactions:

$$T_1^{[P_2]}, T_2^{[P_1]}, T_3^{[P_2]}, T_4^{[P_3]}, T_5^{[P_1,P_2]}, T_6^{[P_2]}, T_7^{[P_3]}, T_8^{[P_1,P_3]}, T_9^{[P_2]}$$

A transaction is represented in the form $T_i^{[P_j]}$ where i is the transaction's unique identifier and $[P_j]$ is the set of partitions that T_i accesses. In this example, we use the transaction's identifier i also as its LSN. That is, we assume T_1 appears in position 1 in the log, T_2 in position 2, and so on.

S processes the transactions (i.e., intentions) in log order. For each transaction, it determines which certifiers will process the intention and determines the synchronization constraint it needs to pass to the certifiers to enforce the parallel certification constraint. The sequence of figures 2– 8 illustrate the parallel certifier in action while it is processing the above sequence of transactions, showing how the certifiers synchronize. In each figure, we emphasize the transaction(s) at the tail of the log being processed by S; time progresses from top to bottom. The LastProcessedLSN at the top of the figure shows the variable's value for each certifier before it has started processing the recently-arrived transactions, i.e., the values after processing the transactions from the previous figure in the sequence. The vertical arrows beside each vertical line shows the processing time of each intention at a certifier. The values updated as a result of processing an intention are highlighted in red. To avoid cluttering the figure, we show minimal information about the previous transactions.

Figure 2 shows a single-partition transaction T_1 accessing P_2. The numbers ①–⑥ identify points in the execution. At ⓪, S determines the synchronization constraint it must pass to C_2, namely, that C_0 must have at least finished processing the last multi-partition transaction that accessed P_2. S reads this value in LastLSNAssignedToC_0Map(P_2). Since S has not processed any multi-partition transaction before T_1, the constraint is LastProcessedLSN(C_0)≥ 0. At ①, S updates LastAssignedLSNMap(C_2)$= 1$ to reflect its assignment of T_1 to C_2. At ②, S assigns T_1 to C_2, and then moves to the next transaction in the log. At ③, C_2 reads LastProcessedLSN(C_0) as 0 and hence determines at ④ that the constraint is satisfied. Therefore, at ⑤ C_2 starts processing T_1. After C_2 completes processing T_1, at ⑥ it updates LastProcessedLSN(C_2) to 1.

Figure 3 shows the processing of the next three single-partition transactions—T_2, T_3, T_4—using steps similar to those in Figure 2. As shown in Figure 4, whenever possible, the certifiers process the transactions in parallel. In the state shown in Figure 3, at ② C_1 is still processing T_2, at ③ C_2 completed processing T_3 and updated its variable LastProcessedLSN(C_2) to 3, and at ④ C_3 completed processing T_4 and updated its variable LastProcessedLSN(C_3) to 4.

Figure 4 shows the processing of the first multi-partition transaction, T_5, which accesses partitions P_1 and P_2. S assigns T_5 to C_0. At ⓪, S specifies the required synchronization constraint, which ensures that

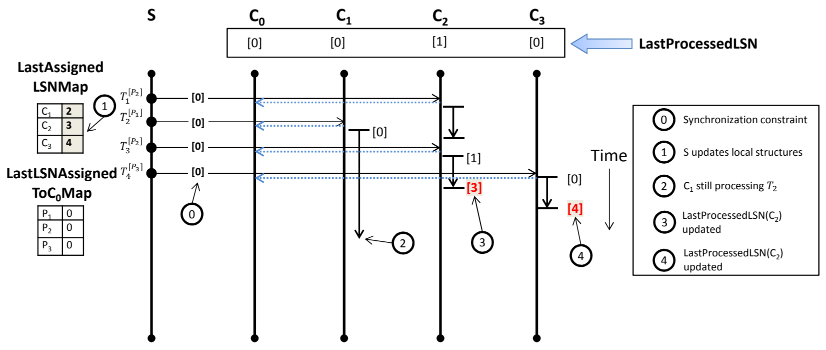


Figure 3: S processes transactions in log order and updates its local structures. Each certifier processes the transactions that S assigns to it.

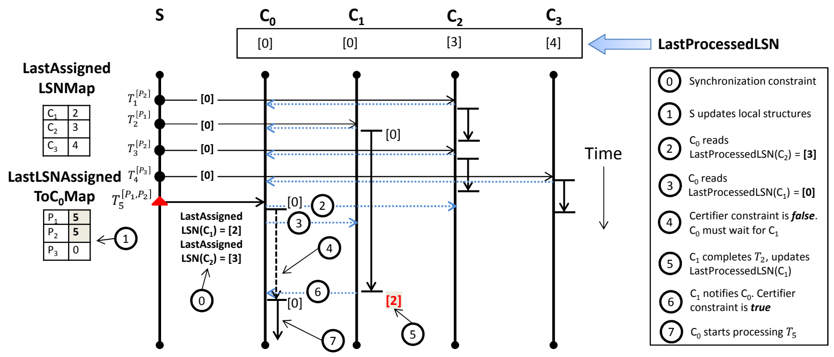
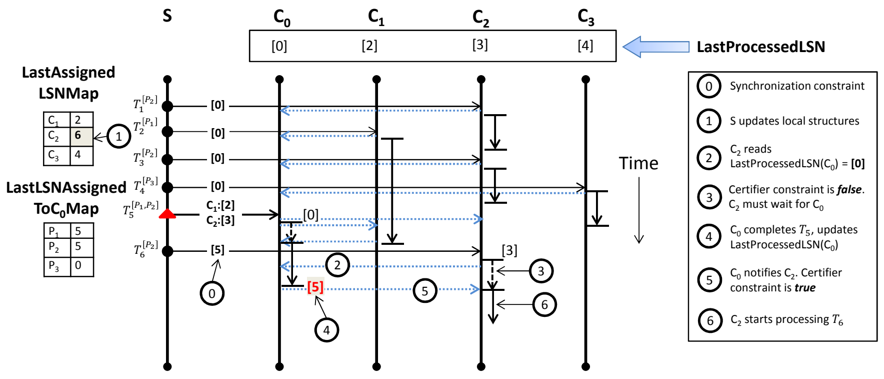

S
C0
C1
C2
C3
[0]
[0]
[3]
[4]
LastProcessedLSN
LastAssigned LSNMap
LastLSNAssigned ToC0Map
LastAssigned LSN(C1) = [2] LastAssigned LSN(C2) = [3]
Time
[2]
Synchronization constraint
S updates local structures
C0 reads LastProcessedLSN(C2) = [3]
C0 reads LastProcessedLSN(C1) = [0]
Certifier constraint is false. C0 must wait for C1
C1 completes T2, updates LastProcessedLSN(C1)
C1 notifies C0. Certifier constraint is true
C0 starts processing T5

Figure 4: For multi-partition transactions, S determines the synchronization constraints and assigns the transaction to C_0.

T_5 is processed after T_2 (the last single-partition transaction accessing P_1) and T_3 (the last single-partition transaction accessing P_2). S reads LastAssignedLSNMap(P_1) and LastAssignedLSNMap(P_2) to determine the LSNs of the last single-partition transactions for P_1 and P_2, respectively. The synchronization constraint shown at ⓪ corresponds to this requirement, i.e., LastProcessedLSN(C_1)$\geq 2 \bigwedge$ LastProcessedLSN(C_2)≥ 3. S passes the constraint to C_0 along with T_5. Then, at ①, S updates LastLSNAssignedToC_0Map(P_1)$= 5$ and LastLSNAssignedToC_0Map(P_2)$= 5$ to reflect that T_5 is the last multi-partition transaction accessing P_1 and P_2. Any subsequent single-partition transaction accessing P_1 or P_2 must now follow the processing of T_5. At ② and ③ C_0 reads LastProcessedLSN(C_2) and LastProcessedLSN(C_1) respectively to evaluate the constraint. At this point in time, C_1 is still processing T_2 and hence at ④ the constraint evaluates to false. Therefore, even though C_2 has finished processing T_3, C_0 waits for C_1 to finish processing T_2. This occurs at ⑤, where it updates LastProcessedLSN(C_1) to 2. Now, at ⑥ C_1 notifies C_0 about this update. So C_0 checks its constraint again and sees that it is satisfied. Therefore, at ⑦ it starts processing T_5.

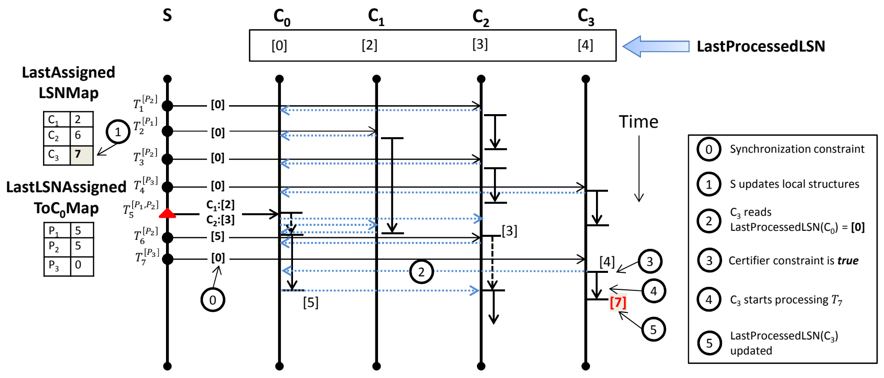
Figure 5 shows processing of the next transaction T_6, a single-partition transaction that accesses P_2. Since both T_5 and T_6 access P_2, C_2 can process T_6 only after C_0 has finished processing T_5. Similar to other single-partition transactions, S constructs this constraint by looking up LastLSNAssignedToC_0Map(P_2) which is 5. Therefore, at ⓪ S passes the constraint LastProcessedLSN(C_0)≥ 5 to C_2 along with T_6, and at ① sets LastLSNAssignedToC_0Map(P_2)$= 6$. At ② C_2 reads LastProcessedLSN(C_0)$= 0$. So its evaluation of the constraint at ③ yields false. C_0 finishes processing T_5 at ④ and sets LastProcessedLSN(C_0)$= 5$. At ⑤, C_0 notifies C_2 that it updated LastProcessedLSN(C_0), so C_2 checks the constraint again and finds it true. Therefore, at ⑥ it starts processing T_6.

While C_2 is waiting for C_0, other certifiers can process subsequent transactions if the constraints allow



S
C0
C1
C2
C3
[0]
[2]
[3]
[4]
LastProcessedLSN
LastAssigned LSNMap
LastLSNAssigned ToC0Map
Time
Synchronization constraint
S updates local structures
C2 reads LastProcessedLSN(C0) = [0]
Certifier constraint is false. C2 must wait for C0
C0 completes T5, updates LastProcessedLSN(C0)
C0 notifies C2. Certifier constraint is true
C2 starts processing T6

Figure 5: Synchronization constraints to order single-partition transactions after a multi-partition transaction.



S
C0
C1
C2
C3
LastProcessedLSN
LastAssigned LSNMap
LastLSNAssigned ToC0Map
Time
Synchronization constraint
S updates local structures
C3 reads LastProcessedLSN(C0) = [0]
Certifier constraint is true
C3 starts processing T7
LastProcessedLSN(C3) updated

Figure 6: Benefits of parallelization for single-partition transactions. C_3 can start processing T_7 while T_6 is waiting for T_5 to complete on C_0.

it. Figure 6 illustrates this scenario where the next transaction in the log, T_7, is a single-partition transaction accessing P_3. Since no multi-partition transaction preceding T_7 has accessed P_3, at ⓪ the constraint passed to C_3 is LastProcessedLSN(C_0)$\geq$ 0. The constraint is trivially satisfied, which C_3 observes at ③. Therefore, while C_2 is waiting, at ④ C_3 starts processing T_7 in parallel with C_0's processing of T_5 and C_2's processing of T_6, thus demonstrating the benefit of parallelizing the certifiers.

Figure 7 illustrates that if the synchronization constraints allow, even a multi-partition transaction can be processed in parallel with other single-partition transactions without any waits. Transaction T_8 accesses P_1 and P_3. At ⓪, based on LastAssignedLSNMap, S generates a constraint of LastProcessedLSN(C_1)$\geq$ 2 $\bigwedge$ LastProcessedLSN(C_3)$\geq$ 7 and passes it along with T_8 to C_0. By the time C_0 starts evaluating its constraint, both C_1 and C_3 have completed processing the transactions of interest to C_0. Therefore, at ② and ③ C_0 reads LastProcessedLSN(C_1)= 2 and LastProcessedLSN(C_3)= 7. So at ④ C_0 finds that the constraint LastProcessedLSN(C_1)$\geq$ 2 $\bigwedge$ LastProcessedLSN(C_3)$\geq$ 7 is satisfied. Thus, it can immediately start processing T_8 at ⑤, even though C_2 is still processing T_6. This is another example demonstrating the benefits of parallelism.

As shown in Figure 8, S processes the next transaction, T_9, which accesses only one partition, P_2. Although T_8 is still active at C_0 and hence blocking further activity on C_1 and C_3, by this time T_7 has finished running at C_2. Therefore, when S assigns T_9 to C_2 at ⓪, C_2's constraint is already satisfied at ③, so C_2 can immediately start processing T_9 at ④, in parallel with C_0's processing of T_8. Later, T_8 finishes at ⑤ and T_9 finishes at ⑥, thereby completing the execution.

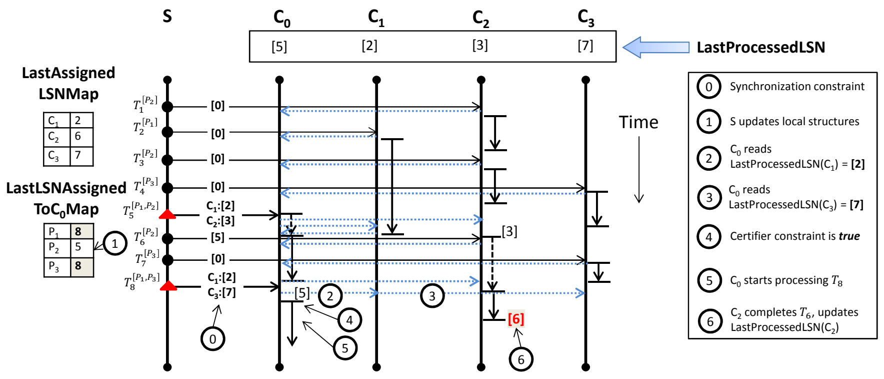

S
C0
C1
C2
C3
[5]
[2]
[3]
[7]
LastProcessedLSN
LastAssigned LSNMap
LastLSNAssigned ToC0Map
Time
[0]
C1:[2]
C2:[3]
[5]
C3:[7]
[3]
[6]
Synchronization constraint
S updates local structures
C0 reads LastProcessedLSN(C1) = [2]
C0 reads LastProcessedLSN(C3) = [7]
Certifier constraint is true
C0 starts processing T8
C2 completes T6, updates LastProcessedLSN(C2)

Figure 7: Benefits of parallelization for multi-partition transaction. C_0 can start processing T_8 while C_2 continues processing T_6.

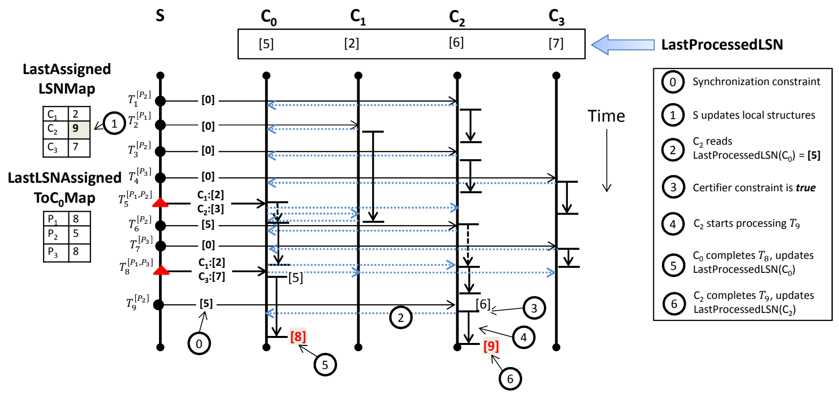

S
C0
C1
C2
C3
[5]
[2]
[6]
[7]
LastProcessedLSN
LastAssigned LSNMap
LastLSNAssigned ToC0Map
Time
[0]
C1:[2]
C2:[3]
[5]
C3:[7]
[8]
[9]
Synchronization constraint
S updates local structures
C2 reads LastProcessedLSN(C0) = [5]
Certifier constraint is true
C2 starts processing T9
C0 completes T8, updates LastProcessedLSN(C0)
C2 completes T9, updates LastProcessedLSN(C2)

Figure 8: Parallel certifier continues processing the transactions in log order and the synchronization constraints ensure correctness of the parallel design.

3.4 Discussion

Correctness requires that for each partition P_i, all transactions that access P_i are certified in log order. There are two cases, single-partition and multi-partition transactions.

- The constraint on a single-partition transaction T_i ensures that T_i is certified after all multi-partition transactions that precede it in the log and that accessed P_i. Synchronization conditions on multi-partition transactions ensure that T_i is certified before all multi-partition transactions that follow it in the log and that accessed P_i.
- The constraint on a multi-partition transaction T_i ensures that T_i is certified after all single-partition transactions that precede it in the log and that accessed partitions $\{P_{i1}, P_{i2}, \ldots\}$ that T_i accessed. Synchronization conditions on single-partition transactions ensure that for each $P_j \in \{P_{i1}, P_{i2}, \ldots\}$, T_i is certified before all single-partition transactions that follow it in the log and that accessed P_j.

Note that transactions that modify a given partition P_i will be certified by C_i or C_0 (but not both), depending on whether it is single-partition or multi-partition.

The extent of parallelism achieved by the proposed parallel certifier depends on designing a partitioning that ensures most transactions access a single partition and that spreads transaction workload uniformly across the partitions. With a perfect partitioning, each certifier can have a dedicated core. So with n partitions, a parallel certifier will run up to n times faster than a single sequential certifier.

Each of the variables that is used in a synchronization constraint—LastAssignedLSNMap, LastProcessedLSN, and LastLSNAssignedToC_0Map—is updatable by only one certifier. Therefore, there are no race conditions on these variables that require synchronization between certifiers. The only synchronization points are the constraints on individual certifiers which can be validated with atomic read operations.

3.5 Finer-Grained Conflict Testing

The parallelized certifier algorithm generates constraints under the assumption that certification of two transactions that access the same partition must be synchronized. This is a conservative assumption, in that two transactions that access the same partition might access the same data item in non-conflicting modes, or might access different data items in the partition, which implies the transactions do not conflict. Therefore, the synchronization overhead can be improved by finer-grained conflict testing. For example, in LastAssignedLSNMap, instead of storing one value for each partition that identifies the LSN of the transaction assigned to the partition, it could store two values: the LSN of the last transaction that read the partition and was assigned to the partition and the LSN of the last transaction that wrote the partition and was assigned to the partition. A similar distinction could be made for the other variables. Then, S could generate a constraint that would avoid requiring that a multi-partition transaction that only read partition P_i be delayed by an earlier single-partition transaction that only read partition P_i, and vice versa. Of course, the constraint would still need to ensure that a transaction that wrote P_i is delayed by earlier transactions that read or wrote P_i, and vice versa.

This finer-grained conflict testing would not completely do away with synchronization between C_0 and C_i, even when a synchronization constraint is immediately satisfied. Synchronization would still be needed to ensure that only one of C_0 and C_i is active on a partition P_i at any given time, since conflict-testing within a partition is single-threaded. Aside from that synchronization, and the use of finer-grained constraints, the rest of the algorithm for parallelizing certification remains the same.

4 Partitioned Log

Partitioning the database also allows partitioning the log, provided ordering constraints between intentions in different logs are preserved. The log protocol is executed by each server that processes transactions. Alternatively, it could be embodied in a log server, which receives requests to append intentions from servers that run transactions.

4.1 Design

In our design, there is one log L_i dedicated to every partition $P_i (\forall i \in [1, n])$, which stores intentions for single-partition transactions accessing P_i. There is also a log L_0, which stores the intentions of multi-partition transactions. If a transaction T_i accesses only P_i, its intention is appended to L_i without communicating with any other log. If T_i accessed multiple partitions $\{P_i\}$, its intention is appended to L_0 followed by communication with all logs $\{L_i\}$ corresponding to $\{P_i\}$. The log protocol must ensure the following constraint for correctness:

> **Partitioned Log Constraint:** There is a total order between transactions accessing the same partitions, which is preserved in all logs where both transactions appear.

Figure 9 provides an overview of the log sequence numbers used in the partitioned log design. A technique similar to vector clocks is used for sequence-number generation [11, 17]. Each log L_i for $i \in [1, n]$ maintains the single-partition LSN of L_i, denoted SP-LSN(L_i), which is the LSN of the last single-partition log record appended to L_i. To order single-partition transactions with respect to multi-partition transactions, every log also maintains the multi-partition LSN of L_i, denoted MP-LSN(L_i), which is the LSN of the last multi-partition

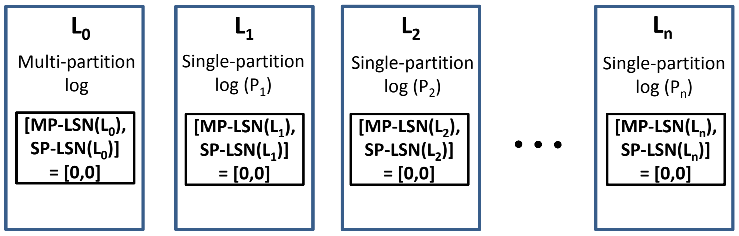

L0
Multi-partition log
[MP-LSN(L0), SP-LSN(L0)] = [0,0]
L1
Single-partition log (P1)
[MP-LSN(L1), SP-LSN(L1)] = [0,0]
L2
Single-partition log (P2)
[MP-LSN(L2), SP-LSN(L2)] = [0,0]
Ln
Single-partition log (Pn)
[MP-LSN(Ln), SP-LSN(Ln)] = [0,0]

Figure 9: Ordering of entries in the log. Each log L_i maintains a compound LSN ([MP-LSN(L_i), SP-LSN(L_i)]) to induce a partial order across conflicting entries in different logs.

transaction that accessed P_i and is known to L_i. The sequence number of each record R_k in log L_i for $i \in [1, n]$ is expressed as a pair of the form $[\text{MP-LSN}_k(L_i), \text{SP-LSN}_k(L_i)]$ which identifies the last multi-partition and single-partition log records that were appended to L_i, including R_k itself. The sequence number of each record R_k in log L_0 is of the form $[\text{MP-LSN}_k(L_0), 0]$, i.e., the second position is always zero. All logs start with sequence number $[0, 0]$.

The order of two sequence numbers is decided by first comparing MP-LSN(L_i) and then SP-LSN(L_i). That is, $[\text{MP-LSN}_m(L_i), \text{SP-LSN}_m(L_i)]$ precedes $[\text{MP- LSN}_n(L_j), \text{SP- LSN}_n(L_j)]$ iff either $\text{MP-LSN}_m(L_i) < \text{MP- LSN}_n(L_j)$, or $(\text{MP-LSN}_m(L_i) = \text{MP-LSN}_n(L_j) \bigwedge \text{SP-LSN}_m(L_i) < \text{SP-LSN}_n(L_j))$. This technique totally orders intentions in the same log (i.e., if $i = j$), while partially ordering intentions of two different logs (i.e., if $i \neq j$). If the ordering between two intentions is not defined, then they are treated as concurrent. Notice that LSNs in different logs are incomparable, because their SP-LSN's are independently assigned. The assignment of sequence numbers is explained in the description of the log protocol.

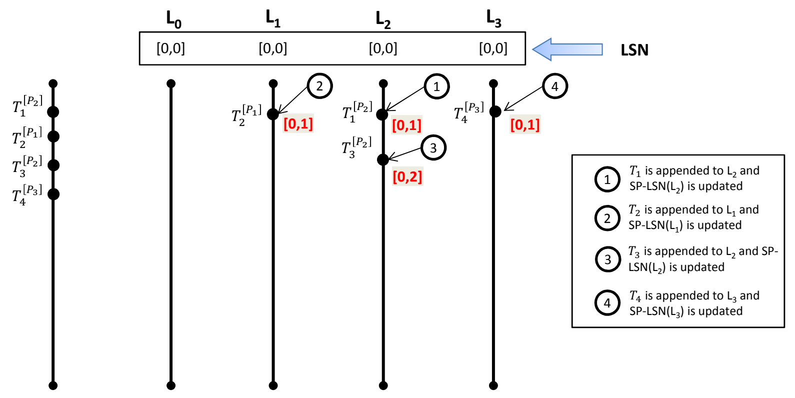
4.2 Log Protocol

Single-partition transactions: Given transaction T_i, if T_i accessed a single partition P_i, then T_i's intention is appended only to L_i. SP-LSN(L_i) is incremented and the LSN of T_i's intention is set to $[\text{mp-lsn}, \text{SP-LSN}(L_i)]$, where mp-lsn is the latest value of MP- LSN(L_0) that L_i has received from L_0.
Multi-partition transactions: If T_i accessed multiple partitions $\{P_{i1}, P_{i2}, \ldots\}$, then T_i's intention is appended to log L_0 and the multi-partition LSN of L_0, MP-LSN(L_0), is incremented. After these actions finish, MP-LSN(L_0) is sent to all logs $\{L_{i1}, L_{i2}, \ldots\}$ corresponding to $\{P_{i1}, P_{i2}, \ldots\}$, which completes T_i's append.

This approach of log-sequencing enforces a causal order between the log entries. That is, two log entries have a defined order only if they accessed the same partition.

Each log $L_i (\forall i \in [1, n])$ maintains MP-LSN(L_i) as the largest value of MP-LSN(L_0) it has received from L_0 so far. However, each L_i does not need to store its MP-LSN(L_i) persistently. If L_i fails and then recovers, it can obtain the latest value of MP-LSN(L_0) by examining L_0's tail. It is tempting to think that this examination of L_0's tail can be avoided by having L_i log each value of MP-LSN(L_0) that it receives. While this does potentially enable L_i to recover further without accessing L_0's tail, it does not avoid that examination entirely. To see why, suppose the last transaction that accessed P_i before L_i failed was a multi-partition transaction that succeeded in appending its intention to L_0, but L_i did not receive the MP-LSN(L_0) for that transaction before L_i failed. In that case, after L_i recovers, it still needs to receive that value of MP-LSN(L_0), which it can do only by examining L_0's tail. If L_0 has also failed, then after recovery, L_i can continue with its highest known value of MP-LSN(L_0) without waiting for L_0 to recover. As a result, a multi-partition transaction might be ordered in L_i at a later position than where it would have been ordered if the failure did not happen.

Alternatively, for each multi-partition transaction, L_0 could run two-phase commit with the logs corresponding to the partitions that the transaction accessed. That is, it could send MP-LSN(L_0) to those logs and wait for



L0
L1
L2
L3
[0,0]
[0,0]
[0,0]
[0,0]
LSN
$T_1^{[P_2]}$
$T_2^{[P_1]}$
$T_3^{[P_2]}$
$T_4^{[P_3]}$
[0,1]
[0,1]
[0,2]
[0,1]
1 T_1 is appended to L2 and SP-LSN(L2) is updated
2 T_2 is appended to L1 and SP-LSN(L1) is updated
3 T_3 is appended to L2 and SP-LSN(L2) is updated
4 T_4 is appended to L3 and SP-LSN(L3) is updated

Figure 10: Single-partition transactions are appended to the single-partition logs L_1, L_2, and L_3.

acknowledgments from all of them before logging the transaction at L_0. However, like any use of two-phase commit, this protocol has the possibility of blocking if a failure occurs between phases one and two.

To avoid this blocking, in our design, when L_0 recovers, it communicates with every L_i to pass the latest value of MP-LSN(L_0). When one of the L_i's recovers, it reads the tail of L_0. This recovery protocol ensures that MP- LSN(L_0) propagates to all single-partition logs.

4.3 An Example

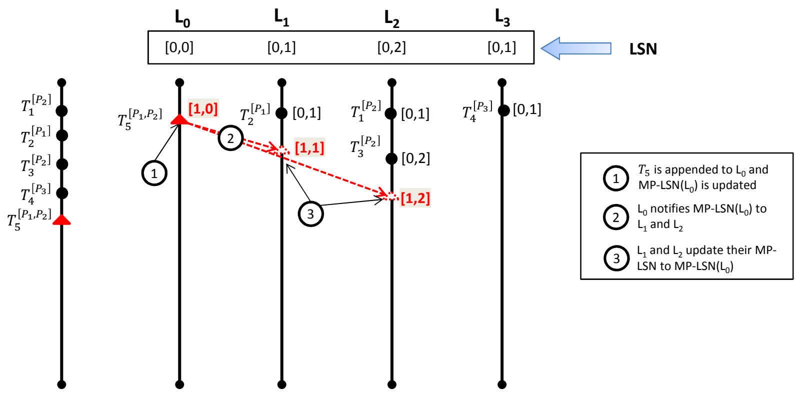
Let us assume that a database has three partitions P_1, P_2, P_3. Let L_1, L_2, L_3 be the logs assigned to P_1, P_2, P_3 respectively, and L_0 be the log for multi-partition transactions. Consider the following sequence of transactions:

$$T_1^{[P_2]}, T_2^{[P_1]}, T_3^{[P_2]}, T_4^{[P_3]}, T_5^{[P_1,P_2]}, T_6^{[P_2]}, T_7^{[P_3]}, T_8^{[P_1,P_3]}, T_9^{[P_2]}$$

As earlier, a transaction is represented in the form $T_i^{[P_j\ldots]}$ where i is a unique transaction identifier; note that this identifier does not induce an ordering between the transactions. The superscript on T_i identifies the partitions that T_i accesses. We use T_i to refer to both a transaction and its intention. In figures 10–14, the vertical line at the extreme left shows the order in which the append requests arrive; time progresses from top to bottom. The LSN at the top of each figure shows each log's LSN before it has appended the recently-arrived transactions, i.e., the values after processing the transactions from the previous figure in the sequence. The black circles on each vertical line for a log shows the append of the transaction and the updated values of the LSN. A multi-partition transaction is shown using a triangle and receipt of a new multi-partition LSN at the single partition logs is shown with the dashed triangle. The values updated as a result of processing an intention are highlighted in red.

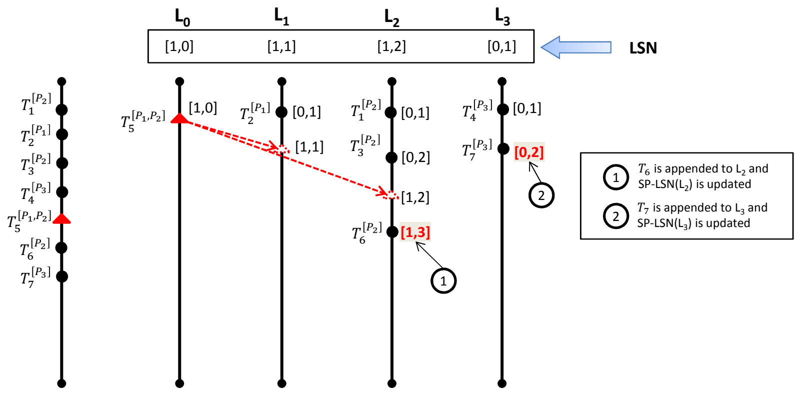
Figure 10 shows four single-partition transactions T_1, T_2, T_3, T_4 that are appended to the logs corresponding to the partitions that the transactions accessed; the numbers ①-④ identify points in the execution. When appending a transaction, the log's SP-LSN is incremented. For instance, in Figure 10, T_1 is appended to L_2 at ① which changes L_2's LSN from $[0, 0]$ to $[0, 1]$. Similarly at ②-④, the intentions for $T_2 - T_4$ are appended and the SP-LSN of the appropriate log is incremented. Appends of single-partition transactions do not need synchronization between the logs and can proceed in parallel; an order is induced only between transactions appended to the same log. For instance, T_1 and T_3 both access partition P_2 and hence are appended to L_2 with T_1 (at ①) preceding T_3 (at ③); however, the relative order of T_1, T_2, and T_4 is undefined.

Multi-partition transactions result in loose synchronization between the logs to induce an ordering among transactions appended to different logs. Figure 11 shows an example of a multi-partition transaction T_5 that accessed P_1 and P_2. When T_5's intention is appended to L_0 (at ①), MP-LSN(L_0) is incremented to 1. In step ②, the new value MP-LSN(L_0) $= 1$ is sent to L_1 and L_2. On receipt of this new LSN (step ③), L_1 and L_2



L0
L1
L2
L3
[0,0]
[0,1]
[0,2]
[0,1]
LSN
[1,0]
[1,1]
[1,2]
[0,1]
[0,2]
1 T5 is appended to L0 and MP-LSN(L0) is updated
2 L0 notifies MP-LSN(L0) to L1 and L2
3 L1 and L2 update their MP-LSN to MP-LSN(L0)

Figure 11: A multi-partition transaction is appended to L_0 and MP-LSN(L_0) is passed to the logs of the partitions accessed by the transaction.



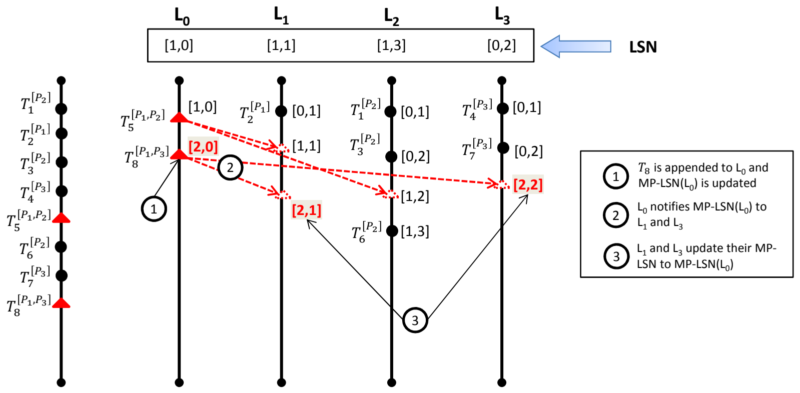
L0
L1
L2
L3
[1,0]
[1,1]
[1,2]
[0,1]
LSN
[0,2]
[1,3]
1 T6 is appended to L2 and SP-LSN(L2) is updated
2 T7 is appended to L3 and SP-LSN(L3) is updated

Figure 12: Single-partition transactions that follow a multi-partition transaction persistently store the new value of MP-LSN(L_i) in L_i.

update their corresponding MP-LSN, i.e., L_1's LSN is updated to $[1, 1]$ and L_2's LSN is updated to $[1, 2]$. As an optimization, this updated LSN is not persistently stored in L_1 or L_2. If either log fails, this latest value can be obtained from L_0 that stores it persistently.

Any subsequent single-partition transaction appended to either L_1 or L_2 will be ordered after T_5, thus establishing a partial order with transactions appended to L_0. As shown in Figure 12, T_6 is a single-partition transaction accessing P_2 which when appended to L_2 (at ①) establishes the order $T_3 < T_5 < T_6$. As a side-effect of appending T_6's intention, MP-LSN(L_2) is persistently stored as well. T_7, another single-partition transaction accessing P_3, is appended to L_3 at ②. It is concurrent with all transactions except T_4, which was appended to L_3 before T_7.

Figure 13 shows the processing of another multi-partition transaction T_8 which accesses partitions P_1 and P_3. Similar to the steps shown in Figure 11, T_8 is appended to L_0 (at ①) and MP-LSN(L_0) is updated. The new value of MP-LSN(L_0) is passed to L_1 and L_3 (at ②) after which the logs update their corresponding MP-LSN (at ③). T_8 induces an order between multi-partition transactions appended to L_0 and subsequent transactions accessing P_1 and P_3. The partitioned log design continues processing transactions as described, establishing a partial order between transactions as and when needed. Figure 14 shows the append of the next single-partition transaction T_9 appended to L_2 (at ①).



L0
L1
L2
L3
[1,0]
[1,1]
[1,3]
[0,2]
LSN
1 T8 is appended to L0 and MP-LSN(L0) is updated
2 L0 notifies MP-LSN(L0) to L1 and L3
3 L1 and L3 update their MP-LSN to MP-LSN(L0)
[2,0]
[2,1]
[1,2]
[2,2]

Figure 13: Different logs advance their LSNs at different rates. A partial order is established by the multi-partition transactions.

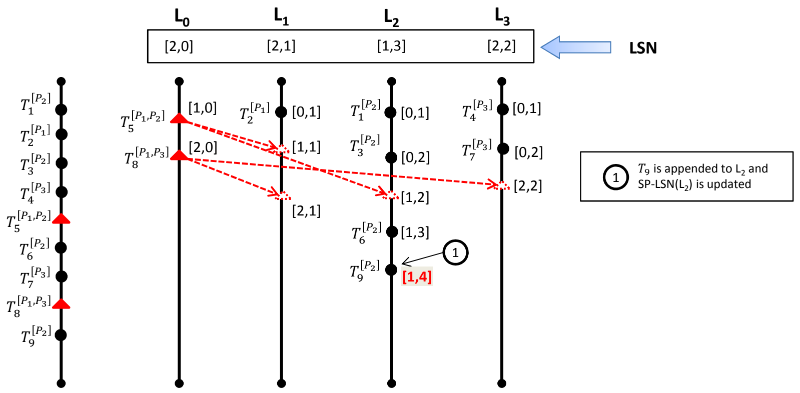


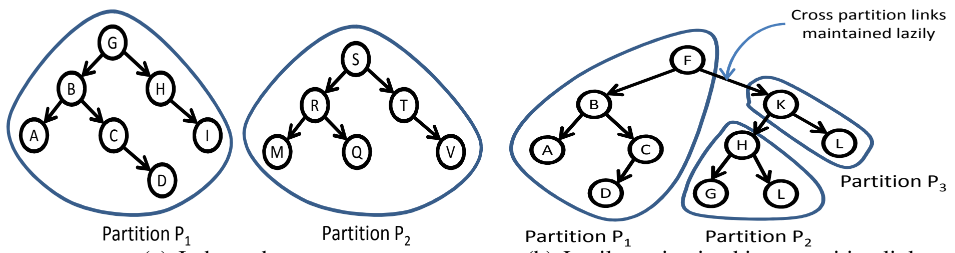
Figure 14: The partitioned log design continues appending single-partition transactions without the need to synchronize with other logs.

4.4 Concurrent Appends to L_0

To ensure that multi-partition transactions have a consistent order across all logs, a new intention is appended to L_0 only after the previous append to L_0 has completed, i.e., the new value of MP-LSN(L_0) has propagated to all single-partition logs corresponding to the partitions accessed by the transaction. This sequential appending of transactions to L_0 might increase the latency of multi-partition transactions. A simple extension can allow parallel appends to L_0 simply by requiring that each log partition retains only the largest MP- LSN(L_0) that it has received so far. If a log L_i receives values of MP-LSN(L_0) out of order, it simply ignores the stale value that arrives late. For example, suppose a multi-partition transaction T_i is appended to L_0 followed by another multi-partition transaction T_j, which have MP-LSN(L_0) $= 1$ and MP-LSN(L_0) $= 2$, respectively. Suppose log L_i receives MP-LSN(L_0) $= 2$ and later receives MP-LSN(L_0) $= 1$. In this case, L_i ignores the assignment MP-LSN(L_0) $= 1$, since it is a late-arriving stale value.

4.5 Discussion

With a sequential certification algorithm, the logs can be merged by each compute server. A multi-partition transaction T_i is sequenced immediately before the first single-partition transaction T_j that accessed a partition that T_i accessed and was appended with T_i's MP-LSN(L_0). To ensure all intentions are ordered, each LSN is augmented with a third component, which is its partition ID, so that two LSNs with the same multi-partition and single-partition LSN are ordered by their partition ID.



(a) Independent trees (b) Lazily-maintained inter-partition links

Figure 15: Partitioning a database in Hyder. Subfigure (a) shows partitions as independent trees. Subfigure (b) shows a single database tree divided into partitions with inter-partition links maintained lazily.

With the parallel certifier, the scheduler S adds constraints when assigning intentions to the certifiers. Certifiers C_i ($i \in [1, \ldots, n]$) will process single-partition transactions appended to L_i and C_0 will process multi-partition transactions appended to L_0. For C_i ($i \in [1, \ldots, n]$) processing a single-partition transaction with LSN [MP-LSN$_k(L_i)$, SP-LSN$_k(L_i)$], the certification constraint for C_i is LastProcessedLSN(C_0) $\geq$ [MP-LSN$_k(L_i)$, 0]. This constraint ensures that the single-partition transaction is certified only after C_0 has certified the multi-partition transaction with MP-LSN$_k(L_i)$. For C_0 processing multi-partition transaction T that accessed partitions $\{P_i\}$ and has LSN [MP-LSN$_k(L_0)$, 0], the scheduling constraint is $\bigwedge_{(\forall j: P_j \in \{P_i\})}$ LastProcessedLSN$(C_j) \geq X_j$, where X_j is the LSN of the last single-partition transaction accessing P_j that appeared in L_j before T. This constraint ensures that the multi-partition transaction T is certified only after all single-partition transactions that are ordered before T have been certified. These constraints can be deduced from the data structures that the scheduling thread S maintains, as described in Section 3.1.

Consider for example the sequence of transactions in Section 4.3 and the LSNs assigned as shown in Figure 14. T_6 is a single partition transaction with LSN $[1, 3]$ is ordered after multi-partition transaction T_5 with LSN $[1, 0]$. T_5's position in L_2 is between T_3 and T_6. The constraint passed to C_2 which certifies T_6 is LastProcessedLSN(C_0) $\geq$ [1, 0]. This constraint ensures that C_2 certifies T_6 only after C_0 has certified T_5. Now consider the certification of multi-partition transaction T_8 which accessed partitions P_1 and P_3. C_0's constraint is LastProcessedLSN$(C_1) \geq [0, 1] \bigwedge$ LastProcessedLSN$(C_3) \geq [0, 2]$. This ensures that C_0 certifies T_8 only after C_1 has certified T_2 and C_3 has certified T_7.

To argue about correctness, we need to show that the partitioned log behaves the same as a non-partitioned log. For sequential certification, the partitioned log is merged into a single non-partitioned log, so the result follows immediately. For parallel certification, for each log L_i ($i \neq 0$), the constraints ensure that each multi-partition transaction is synchronized between L_0 and L_i in exactly the same way as in the single-log case.

If most of the transactions access only a single partition and there is enough network capacity, this partitioned log design provides a nearly linear increase in log throughput as a function of the number of partitions. The performance impact of multi-partition transactions is not expected to be very high.

5 Partitioning in Hyder – An application scenario

As we explained in Section 1, Hyder is a system that uses OCC and a log-structured database that is shared by all servers. Given an approximate partitioning of the database, the parallel certification and partitioned log algorithms described in this paper can be directly applied to Hyder. Each parallel certifier would run Hyder's OCC algorithm, called meld, and each log partition would be an ordinary Hyder log storing updates to that partition. Each log stores the after-image of the binary search tree created by transactions updating the corresponding partition. Multi-partition transactions result in a single intention record that stores the after-image of all partitions, though this multi-partition intention can be split so that a separate intention is created for every partition.

The application of approximate partitioning to Hyder assumes that the partitions are independent trees as shown in Figure 15(a). Directory information is maintained that describes which data is stored in each partition. During transaction execution, the executer tracks the partitions accessed by the transaction. This information is included in the transaction's intention, which is used by the scheduler to parallelize certification and by the log partitioning algorithm.

In addition to the standard Hyder design where all compute nodes run transactions (on all partitions), it is possible for a given compute node to serve only a subset of the partitions. However, this increases the cost of multi-partition transaction execution and meld.

A design with a partitioned tree, as shown in Figure 15(b), is also possible, though at the cost of increased complexity. Cross-partition links are maintained as logical links, to allow single-partition transactions to proceed without synchronization and to minimize the synchronization required to maintain the database tree. For instance, in Figure 15(b), the link between partitions P_1 and P_3 is specified as a link from node F to the root K of P_3. Since single-partition transactions on P_3 modify P_3's root, traversing this link from F requires a lookup of the root of partition P_3. This link is updated during meld of a multi-partition transaction accessing P_1 and P_3 and results in adding an ephemeral node replacing F if F's left subtree was updated concurrently with the multi-partition transaction. The generation of ephemeral nodes is explained in [9].

6 Related Work

Optimistic concurrency control (OCC) was introduced by Kung and Robinson in [14]. Its benefits and trade-offs have been extensively explored in [1, 2, 12, 16, 18, 20]. Many variations and applications of OCC have been published. For example, Tashkent uses a centralized OCC validator over distributed data [10]. An OCC algorithm for an in-memory database is described in [15]. None of these works discuss ways to partition the algorithm.

An early timestamp-based concurrency control algorithm that uses partitioning of data and transactions is described in [5]. More recent examples of systems that partition data to improve scalability are in [3, 13, 19, 21].

The only other partitioned OCC algorithm we know of is for the Tango system[4]. In Tango, after a server runs a multi-partition transaction T and appends T's log record, it rolls forward the log to determine T's commit/abort decision and then writes that decision to the log. The certifier of each partition uses that logged decision to decide how to act on log records from multi-partition transactions. This enables the certifier to update its version state of data, so it can perform OCC validation of single-partition transactions. That is, each certifier C_i reads the sequence of single-partition and multi-partition log records that read or updated P_i. When C_i encounters a multi-partition log record, it waits until it sees a decision record for that transaction in the log. This synchronization point is essentially the same as that of C_i waiting for C_0 in our approach. However, the mechanism is different in two ways: the synchronization information is passed through the log, rather than through shared variables; and every server that runs a multi-partition transaction also performs the log roll-forward to determine the transaction's decision (although this could be done by a centralized server, like C_0). The experiments in[4] show good scalability with a moderate fraction of cross-partition transactions. It remains as future work to implement the algorithm proposed here and compare it to Tango's.

In Tango, all partitions append log records to a single sequential log. Therefore, the partitioned log constraint is trivially enforced. By contrast, our design offers explicit synchronization between log records that access the same partition. This enables them to be written to different logs, which in aggregate can have higher bandwidth than a single log, like Tango's.

Another approach to parallelizing meld is described in[6]. It uses a pipelined design that parallelizes meld onto multiple threads. One stage of the pipeline preprocesses each intention I by testing for conflicts with committed transactions before the final meld step. It also "refreshes" I by replacing stale data in I by committed updates. The other stage combines adjacent intentions in the log, also before the final meld step. Each of these stages reduces the work required by the final meld step.

7 Concluding Remarks

In this paper, we explained a design to leverage approximate partitioning of a database to parallelize the certifier of an optimistic concurrency control algorithm and its accompanying log. The key idea is to dedicate a certifier and a log to each partition so that independent non-conflicting transactions accessing only a single partition can be processed in parallel while ensuring transactions accessing the same partition are processed in a sequence. Since partitioning of the database, and hence the transactions, need not be perfect, i.e., a transaction can access multiple partitions, our design processes these multi-partition transactions using a dedicated multi-partition certifier and log. The efficiency of the design stems from using lightweight synchronization mechanisms—the parallel certifiers synchronize using constraints while the partitioned log synchronizes using asynchronous causal messaging. The design abstracts out the details of the certifier and the logging protocol, making it applicable to a wide variety of systems. We also discussed the application of the design in Hyder, a scale-out log-structured transactional record manager. Our design allows Hyder to leverage approximate partitioning to further improve the system's throughput.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34, 1995.

[2] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed multi-version optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1):45 – 59, 1987.

[3] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Hushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. 5th Biennial Conf. on Innovative Data Systems Research*, 2011.

[4] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. Davis, S. Rao, T. Zou, and A. Zuck. Tango: distributed data structures over a shared log. In *Proc. 24th ACM Symp. on Operating System Principles*, pages 325–340, 2013.

[5] P. Bernstein, D. Shipman, and J. R. Jr. Concurrency control in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 5(1):1 – 17, 1980.

[6] P. A. Bernstein, S. Das, B. Ding, and M. Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2015.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[8] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *Proc. 5th Biennial Conf. on Innovative Data Systems Research*, pages 9–20, 2011.

[9] P. A. Bernstein, C. W. Reid, M. Wu, and X. Yuan. Optimistic concurrency control by melding trees. *Proc. VLDB Endowment*, 4(11):944–955, 2011.

[10] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. In *Proc. 1st ACM SIGOPS/EuroSys European Conf. on Computer Systems*, pages 117 – 130, 2006.

[11] M. J. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proc. 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 70–75, 1982.

[12] R. Gruber. Optimistic concurrency control for nested distributed transactions. Technical Report MIT/LCS/TR-453, MIT, June 1989.

[13] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endowment*, 1(2):1496 – 1499, 2008.

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

[15] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endowment*, 5(4):298–309, 2011.

[16] G. Lausen. Concurrency control in database systems: A step towards the integration of optimistic methods and locking. In *Proc. ACM Annual Conf.*, pages 64 – 68, 1982.

[17] D. S. Parker Jr., G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. A. Edwards, S. Kiser, and C. S. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Software Eng.*, 9(3):240–247, 1983.

[18] S. Phatak and B. R. Badrinath. Bounded locking for optimistic concurrency control. Technical Report DCS-TR-380, Rutgers University, 1999.

[19] J. Rao, E. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endowment*, 4(4):243 – 254, 2011.

[20] A. Thomasian and E. Rahm. A new distributed optimistic concurrency control method and a comparison of its performance with two-phase locking. In *Proc. 10th Int. Conf. on Distributed Computing Systems*, pages 294 – 301, 1990.

[21] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1 – 12, 2012.

Transaction Processing Techniques for Modern Hardware and the Cloud

Justin Levandoski Sudipta Sengupta Ryan Stutsman Rui Wang

Microsoft Research

Abstract

The Deuteronomy architecture provides a clean separation of transaction functionality (performed in a transaction component, or TC) from data storage functionality (performed in a data component, or DC). For the past couple of years, we have been rethinking the implementation of both the TC and DC in order for them to perform as well, or better than, current high-performance database systems. The result of this work so far has been a high-performance DC (the Bw-tree key value store) that currently ships in several Microsoft systems (the Hekaton main-memory database, Azure DocumentDB, and Bing). More recently, we have been completing the performance story by redesigning the TC. We are driven by two main design goals: (1) ensure high performance in various deployment scenarios, regardless of whether the TC and DC are co-located or distributed and (2) take advantage of modern hardware (multi-core and multi-socket machines, large main memories, flash storage). This paper summarizes the design of our new TC that addresses these two goals. We begin by describing the Deuteronomy architecture, its design principles, and several deployment scenarios (both local and distributed) enabled by our design. We then describe the techniques used by our TC to address transaction processing on modern hardware. We conclude by describing ongoing work in the Deuteronomy project.

1 Deuteronomy

The Deuteronomy architecture decomposes database kernel functionality into two interacting components such that each one provides useful capability by itself. The idea is to enforce a clean, layered separation of duties where a transaction component (TC) provides concurrency control and recovery that interacts with one or more data components (DC) providing data storage and management duties (access methods, cache, stability). The TC knows nothing about data storage details. Likewise, the DC knows nothing about transactional functionality — it is essentially a key-value store. The TC and DC may be co-located on the same machine or may be separated on different machines by a network. Regardless, the TC and DC interact through two main channels: (1) record operations, supporting the so-called CRUD (create, read, update, delete) interfaces, and (2) control operations that enforce the write-ahead log protocol and enable clean decomposition of components. Details of the Deuteronomy architecture have been discussed in prior work [10].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

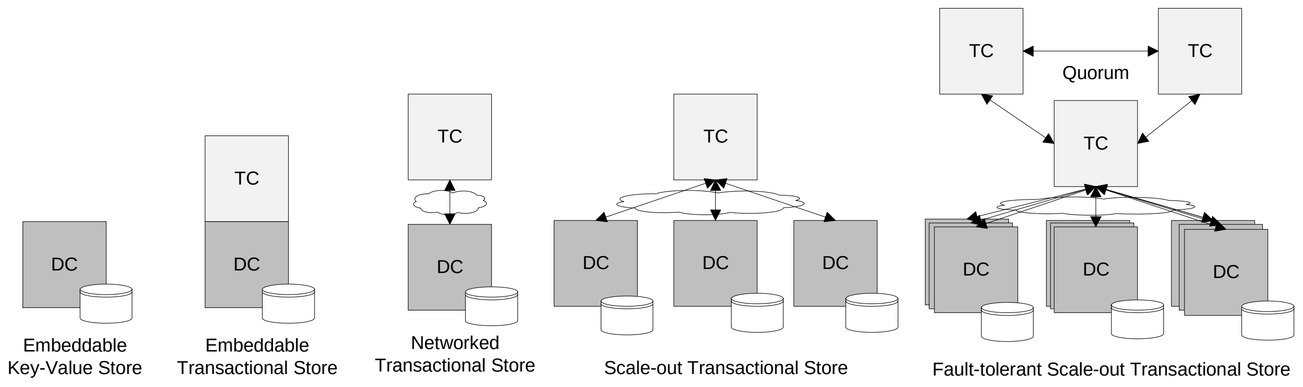

DC
Embeddable Key-Value Store
TC
DC
Embeddable Transactional Store
TC
DC
Networked Transactional Store
TC
DC
DC
DC
Scale-out Transactional Store
TC
TC
Quorum
TC
DC
DC
DC
Fault-tolerant Scale-out Transactional Store

Figure 1: Different deployment scenarios (local and distributed) for Deuteronomy.

1.1 Flexible Deployment Scenarios

A salient aspect of Deuteronomy is that it enables a number of interesting and flexible deployment scenarios, as depicted in Figure 1. These scenarios span both distributed configurations and as well as cases where components coexist on the same machine. For instance, applications needing a high-performance key-value store might only embed the data component (e.g., the Bw-tree). A transaction component can be added on top to make it a high-performance embedded transactional key-value store. The TC and DC can also interact over a network when located on different machines. A TC can provide transaction support for several DCs (serving disjoint data records) in order to scale out. Finally, both the TC and DC can be replicated to form a fault-tolerant, scale-out transactional store. Today, such flexibility is especially important as applications span a wide array of deployments, from mobile devices, to single servers, all the way to on-premise clusters or cloud environments; this is certainly the case at Microsoft. Thus, systems can mix, match, and configure Deuteronomy's reusable components in a number of different ways to meet their data management needs.

1.2 The Current Deuteronomy Stack

After finishing an initial implementation of Deuteronomy that demonstrated its feasibility [10], we began working on a redesign of both the TC and DC to achieve performance competitive with the latest high performance systems. Our first effort resulted in a DC consisting of the Bw-tree latch-free access method [8] and LLAMA [7], a latch-free, log structured cache and storage manager. The result was a key-value store (DC) that executes several million operations per second that is now used as the range index method in SQL Server Hekaton [3] and the storage and indexing layer in other Microsoft products, e.g., Azure DocumentDB [4] and Bing [2]. This effort also showed that in addition to a TC/DC split, the DC could be further decomposed to maintain a hard separation between access methods and the LLAMA latch-free, log structured cache and storage engine, as depicted in Figure 2. With a DC capable of millions of operations per second the original TC became the system bottleneck. The rest of this article summarizes our effort to build a high-performance TC. We begin by discussing the high-level architecture and design principles in Section 2. We then provide a summary of the techniques that allow the TC to achieve high performance (Sections 3 to 5). We end by providing a summary of ongoing work in the Deuteronomy project in Section 6.

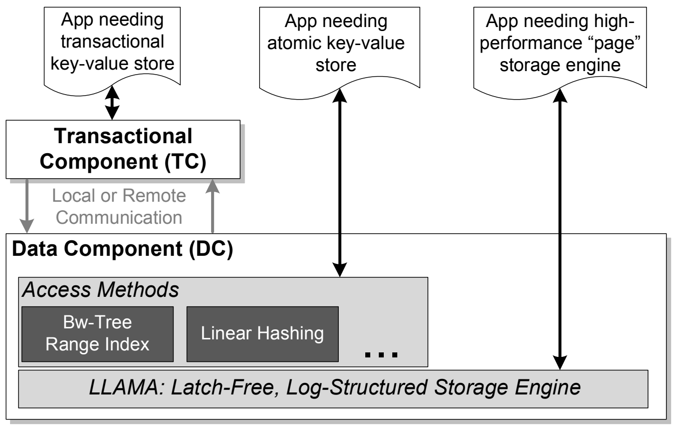

App needing transactional key-value store
App needing atomic key-value store
App needing high-performance "page" storage engine
Transactional Component (TC)
Local or Remote Communication
Data Component (DC)
Access Methods
Bw-Tree Range Index
Linear Hashing
...
LLAMA: Latch-Free, Log-Structured Storage Engine

Figure 2: The Deuteronomy layered architecture. In addition to a TC/DC split, we further support a hard separation between access methods and a high-performance storage engine targeting modern hardware.

2 Transaction Component Design

2.1 Design Principles

As we outlined in the previous section, the Deuteronomy architecture enables a number of different deployment scenarios. Whether the TC and DC are co-located, or remote (or replicated), the design of our transaction component aims to achieve excellent performance in any scenario. We are driven by the following principles.

1. *Exploit modern hardware*. Modern machines, whether located on premise or in a data center, currently contain multiple CPU sockets, flash SSD, and large main memories. We aim to exploit the performance advantages of this modern hardware by using latch-free data structures, log structuring, and copy-on-write delta updates. These techniques avoid update-in-place and are well-suited to modern multi-core machines with deep cache hierarchies and low-latency flash storage. All components in the Deuteronomy stack (both TC and DC) take advantage of these techniques.

2. *Eliminate high latency from the critical paths*. DC access latency can limit performance, especially when the TC and DC are located on separate machines (or even sockets). This is particularly bad for hotspot data where the maximum update rate of 1/latency (independent of concurrency control approach) can severely limit performance. TC caching of hot records is essential to minimizing latency.

3. *Minimize transaction conflicts*. Modern multi-version concurrency techniques demonstrate the ability to enormously reduce conflicts. Deployed systems like Hekaton [3] have proven that MVCC performs well in practice. We also exploit MVCC in the TC in the form of multi-version timestamp ordering.

4. *Minimize data traffic between TC and DC*. Data transfers are very costly. Our "distributed" database kernel requires some data to be transferred between TC and DC. Our design aims to limit the TC/DC traffic as much as possible.

5. *Exploit batching*. Effective batching can often reduce the per "item" cost of an activity. We exploit batching when shipping data updates from the TC to the DC.

6. *Minimize data movement and space consumption*. Obviously, one wants only "necessary" data movement. We use pure redo recovery and place data in its final resting place immediately on the redo log within the TC, avoiding what is very frequently a major performance cost, while reducing memory footprint as well.

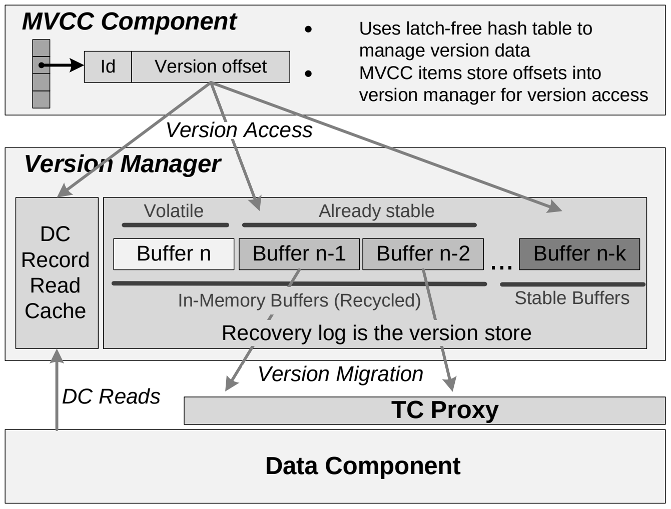


Figure 3: The Deuteronomy TC architecture and data flow. The TC consists of an multi-version concurrency control component and a version manager to manage the redo log and version cache. A TC Proxy is co-located with the DC to enable fast parallel replay of committed records to the database.

2.2 Transaction Component Architecture

Figure 3 presents the TC architecture, illustrating the flow of data within it and between TC and DC. The TC consists of three main components: (a) an MVCC component to manage the concurrency control logic; (b) a version manager to manage our redo log (also our version store) as well as cached records read from the DC; and (c) a TC Proxy that lives beside the DC and whose job is to submit committed operations to the DC, which maintains database state.

Caching Since we use MVCC, versions must be cached somewhere for concurrency control purposes. Versions resulting from updates are written into the redo recovery log. These recovery log versions are accessed via the MVCC component, which stores *version offsets* as part of its version entry and requests them through the version manager interface. The version manager uses the redo log as part of the TC record version cache. In-memory log buffers are written to stable storage and retained in memory to serve as a version cache until they are eventually recycled and reused.

Versions of data not recently updated need to be acquired from the DC. To make them accessible to MVCC, the version manager retains these versions in the read cache. Both read cache and recovery log buffers are subject to different forms of log structured cleaning (garbage collection). Thus, an MVCC request to the version manager could hit (1) the read cache; or (2) a log buffer (in-memory or stable).

To minimize data movement, we immediately post updates in their final resting place on the recovery log. Immediately logging updates means that uncommitted updates are on the log without any means of undoing them if their transaction is aborted. So we cannot post updates to the DC until we know that the containing transaction has committed. Updates from aborted transactions are simply never applied at the DC.

TC Proxy The TC proxy is a module that resides on the same machine as the DC. The TC Proxy receives log buffers from the version manager after the buffer is made stable and posts updates to the DC as appropriate. Since we use pure redo logging, these updates are blind, in that they do not require first reading a pre-image of the record for undo. Posting can only be done after the transaction responsible for the update has committed. This posting is part of log buffer garbage collection when the TC Proxy and DC are collocated with the TC. Otherwise cleaning occurs once the buffer has been received by a remote TC Proxy. Posting updates to the DC is not part of the latency path of any operation and is done in the background. However, it is important for it to be somewhat timely, because it constrains the rate at which MVCC entries can be garbage collected.

3 Concurrency Control

The TC uses timestamp ordering (TO) as its concurrency control method. TO is a very old method [13] (including its multi-version variant); the idea is to assign a timestamp to a transaction such that all of its writes are associated with its timestamp, and all of its reads only "see" versions that are visible as of its timestamp. A correct timestamp order schedule of operations using the timestamps is then enforced. Transactions are aborted when the timestamp ordering cannot be maintained. While TO never took off in practice, recent work from us [9] and others [6, 12] has shown that TO performs well when all concurrency control metadata is kept in memory. In addition, a real advantage of TO is that it enables serializable isolation without needing a validation step at the end of the transaction [5].

We use a multi-version form of TO [1]. The TC tracks transactions via a transaction table. Each entry in the table denotes a transaction status, its transaction id, and a timestamp issued when the transaction starts. Version information for a record is managed by a latch-free MVCC hash table (see [9] for details). The entry for each version in the table is marked with the transaction id that created the version. This permits an easy check of the transaction information for each version, including its timestamp. The status in the transaction table entry indicates whether the transaction is active, committed, or aborted.

In order to keep the MVCC table from growing unbounded to the size of the entire database, we periodically perform garbage collection to remove entries. We compute the oldest active transaction (the one with the oldest timestamp), or OAT, which is used to determine which version information is safe to garbage collect from the table. We conservatively identify versions that are not needed by transactions or that are available from the DC. These items are (1) any updated version older than the version visible to the OAT; these versions are never needed again. (2) Any version visible to the OAT but with no later updates can also be discarded once it is known to have been applied at the DC. We are guaranteed to be able to retrieve such a record version from the DC.

4 Version Management

Providing fast access to versions is critical for high performance in Deuteronomy. The TC serves requests for its cached versions from two locations. The first is directly from in-memory recovery log buffers. The second is from a "read cache" used to hold hot versions that may not have been written recently enough to remain in the recovery log buffers.

4.1 Recovery Log Caching

The TC's MVCC approves and mediates all updates, which allows it to cache and index updated versions. The TC makes dual-use of the MVCC hash table to gain performance: the act of performing concurrency control naturally locates the correct version contents to be returned. When a transaction attempts an update, it is first approved by MVCC. If the update is permitted, it results in the new version being stored in a recovery log buffer within the version manager. Then, an entry for the version is created in the MVCC hash table that associates it with the updating transaction and contains an offset to the version in the recovery log. Later reads for that version that are approved by the MVCC directly find the data in memory using the version offset. Thus, in addition to concurrency control, the MVCC hash table serves as a version index, and the in-memory recovery buffers play the role of a cache.

Each updated version stored at the TC serves both as an MVCC version and as a redo log record for the transaction. The TC uses pure redo logging and does not include "before" images in these log records; this ensures that the buffers are dense with version records that make them efficient as a cache. Versions are written immediately to the recovery log buffer to avoid later data movement. Consequently, an update (redo log record) cannot be applied at the DC until its transaction is known to be committed, since there is no way to undo the update. The TC Proxy (§5) ensures this.

Recovery log buffers are written to stable storage to ensure transaction durability. The use of buffers as a main memory version cache means they are retained in memory even after they have been flushed to disk. Once a buffer is stable, it is given to the TC Proxy so that the records (that are from committed transactions) that it contains can be applied to the DC.

4.2 Read Cache

The recovery log acts as a cache for recently written versions, but some read-heavy, slow-changing versions are eventually evicted from the recovery log buffers when they are recycled. Similarly, hot read-only versions may pre-exist in the DC and are never cached in the recovery log. If reads for these hot versions were always served from the DC, TC performance would be limited by the round-trip latency to the DC. To prevent this, the TC's version manager keeps an in-memory read cache to house versions fetched from the DC. Each version that is fetched from the DC is placed in the read cache, and an entry is added to the MVCC table for it.

The read cache is latch-free and log-structured, similar to recovery log buffers. One key difference is that the read cache includes its own latch-free hash index structure. The MVCC table refers to versions by their log-assigned offsets, and this hash index allows a version to be relocated into the cache without having to update references to it. The TC uses this so that it can relocate hot versions into the cache from recovery log buffers when it must recycle the buffers to accommodate new versions.

Being log-structured, the read cache naturally provides a FIFO eviction policy. The cache is a large ring, and newly appended versions overwrite the oldest versions in the cache. So far, this policy has been sufficient for us; it works symbiotically with the TC's MVCC to naturally preserve hot items. This is because whenever a record is updated it is "promoted" to the tail of the record log buffers, naturally extending the in-memory lifetime of the record (though, via a new version). So far, our metrics indicate there is little incentive for more complex eviction polices.

Finally, the read cache aggressively exploits its cache semantics to eliminate synchronization to a degree that would otherwise be unsafe. For example, its hash index is lossy and updaters don't synchronize on update collisions. Also, readers may observe versions that are concurrently being overwritten. In these cases, the read cache detects (rather than prevents) the problem and treats the access as a cache miss. The concurrently developed MICA key-value store [11] uses similar techniques, though we were unaware of its design until its recent publication.

5 TC Proxy: Fast Log Replay at the DC

The TC Proxy's main job is to receive stable recovery log buffers from the TC and efficiently apply the versions within them to the DC using parallel log replay. The TC Proxy runs co-located with the DC. It enforces a well-defined contract that separates the TC and the DC sufficiently to allow the DC to run locally with the TC or on a remote machine. The TC Proxy receives recovery log buffers from the TC as they become stable, so the buffers act as a natural batch and make communication efficient even when over a network. Finally, it unbundles operations and applies them to the DC as appropriate.

The TC proxy must provide two things to achieve good overall performance. First, log records must be processed as few times as possible. The pure redo design requirement complicates this. Not all of the records

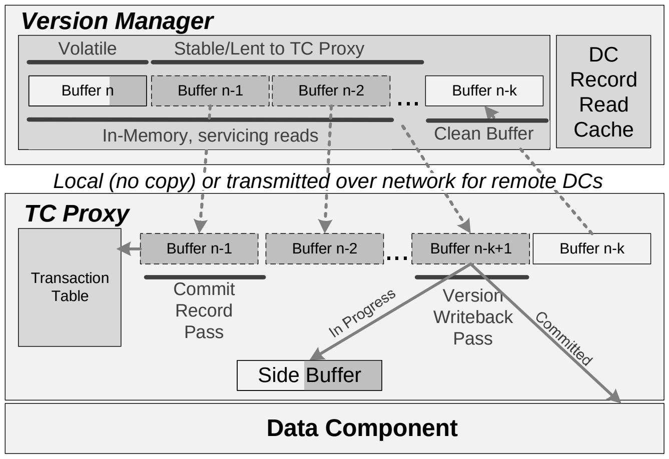


Figure 4: The TC Proxy receives full, stable log buffers from the TC. An eager pass updates transaction statuses; a lazy pass applies committed operations to the DC.

in the received buffers are from committed transactions, so they may have to be applied after a delay. Second, log replay must be parallelized across multiple cores to prevent it from becoming a bottleneck in write-heavy workloads.

Ideally, the TC Proxy would only encounter records of committed transactions when it processes a recovery log buffer. Then, all operations could be applied as they were encountered. To make that ideal scenario "almost true", buffer processing is delayed. Figure 4 depicts this. The TC Proxy immediately scans arriving buffers for commit records and updates its own version of the transaction table to track which transactions are committed (as indicated by encountering their commit records). A second scan applies all of the operations of transactions known to be committed. Operations of aborted transactions are discarded, and operations whose transaction outcome is not yet known are relocated into a side buffer.

This works well when very few operations have undecided outcomes. Delaying the replay scan can minimize the number of operations that must be relocated into a side buffer. In practice, the side buffer space requirement for storing the undecided transaction operations has been tiny, but will vary by workload. A side buffer operation is applied to the DC (or discarded) whenever its transaction's commit (or abort) record is encountered.

The TC Proxy parallelizes log replay by dispatching each received log buffer to a separate hardware thread. Log records are applied to the DC out of log order. The LSN of the commit record for each version is used for idempotence checking, and versions that have already been superseded by newer versions at the DC are discarded. That is, the LSN is used to ensure that the version associated with a particular key progresses monotonically with respect to log order. In the current experiments, this parallel replay naturally expands to use all the cores of a socket under write-heavy loads.

6 Ongoing Work

Our future plans for Deuteronomy involve a number of interesting directions:

- **Range concurrency control**. The TC currently supports serializable isolation for single-record operations, but not phantom avoidance for range scans. We plan to integrate range support into our multi-version timestamp order concurrency control scheme while still maintaining high performance.

- **Scale out**. A number of performance experiments have shown that the TC is capable of handling millions of transactions per second [9]. A current major thrust of our work is to scale out the number of DCs beneath a TC. In this setting, each DC manages a disjoint set of records while transactional management for all records (across all DCs) is managed by a single TC (see the "scale out transactional store" deployment in Figure 1).

- **Fault tolerance**. Fault tolerance is extremely important, especially if the TC and DC are embedded in a cloud-based system running on commodity machines. We are actively working on replication methods such that a TC or DC can actively fail over efficiently with little downtime.

7 Acknowledgements

This is joint work with David Lomet. We would like to also thank the various product groups at Microsoft (SQL Server, Azure DocumentDB, and Bing) for helping to improve the Deuteronomy stack.

References

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

[2] Bing. **`https://www.bing.com`**.

[3] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*, 2013.

[4] Azure DocumentDB. **`https://www.documentdb.com`**.

[5] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB*, 5(4):286–297, 2012.

[6] Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting Hardware Transactional Memory in Main-Memory Databases. In *ICDE*, pages 580–591, 2014.

[7] Justin Levandoski, David Lomet, and Sudipta Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *PVLDB*, 6(10):877–888, 2013.

[8] Justin Levandoski, David Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*, 2013.

[9] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. High Performance Transactions in Deuteronomy. In *CIDR*, 2015.

[10] Justin Levandoski, David B. Lomet, Mohamed F. Mokbel, and Kevin Zhao. Deuteronomy: Transaction Support for Cloud Data. In *CIDR*, pages 123–133, 2011.

[11] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *NSDI*, pages 429–444, 2014.

[12] Henrik Mühe, Stephan Wolf, Alfons Kemper, and Thomas Neumann. An Evaluation of Strict Timestamp Ordering Concurrency Control for Main-Memory Database Systems. In *IMDM*, pages 74–85, 2013.

[13] David P. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, M.I.T., Cambridge, MA, 1978.

Scaling Off-the-Shelf Databases with Vela: An approach based on Virtualization and Replication

Tudor-Ioan Salomie [*]
Google Inc.
tsalomie@google.com

Gustavo Alonso
Systems Group,
Computer Science Department,
ETH Zürich, Switzerland
alonso@inf.ethz.ch

Abstract

Off-the-Shelf (OTS), relational databases can be challenging when deployed in the cloud. Given their architectures, limitations arise regarding performance (because of virtualization), scalability (because of multi-tenancy), and elasticity (because existing engines cannot easily take advantage of the application migration possibilities of virtualized environments). As a result, many database engines tailored to cloud computing differ from conventional engines in functionality, consistency levels, support for queries or transactions, and even interfaces.

Efficiently supporting Off-the-Shelf databases in the cloud would allow to port entire application stacks without rewriting them. In this paper we present a system that combines snapshot isolation (SI) replication with virtualization to provide a flexible solution for running unmodified databases in the cloud while taking advantage of the opportunities cloud architectures provide. Unlike replication-only solutions, our system works well both within larger servers and across clusters. Unlike virtualization only solutions, our system provides better performance and more flexibility in the deployment.

1 Introduction

When deploying applications on the cloud there is a trade-off between using Off-the-Shelf (OTS) databases or cloud-ready solutions. While using OTS databases is convenient as entire application stacks can be migrated to the cloud, there is no efficient way of doing this yet [1]. Ad-hoc solutions that try to virtualize OTS databases often hit performance bottlenecks due to virtualization [6] or a miss-match between what virtualization offers and what OTS databases support [34]. Recent trends in increasing number of cores in cluster nodes also add to the complexity of the problem. OTS databases and replication-based solutions built on top of them have difficulties in harnessing these computational resources [19, 29, 7].

Cloud-ready solutions offer scalability and elasticity at the cost of reduced functionality, varied levels of consistency, or limited support in queries and transactions. For instance, Dynamo [13] sacrifices consistency

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*]The work presented in this paper was done during the author's PhD studies in the Systems Group at ETH Zürich, Switzerland and is in no way affiliated, nor does it represent the views of Google Inc.

under certain scenarios and has a simple put/get query API. Bigtable [10] also has a simple query API that does not support a full relational model.

Relational systems like SQL Azure [9] use replication for fault tolerance and load balancing, but the replication factor is fixed with no way of tuning it in favor of fault tolerance or performance. Similarly, Amazon RDS [2] relies on replication but imposes a variety of restrictions based on the chosen underlying database engine.

In this paper we present **Vela**, a system for running OTS databases on the cloud. The system relies on *virtualization* (to take advantage of the system architectural possibilities available in the cloud) and snapshot isolation (SI) *replication* (for fault tolerance, performance, and scalability, without sacrificing consistency). Synchronous replication has become a de facto standard for achieving scalability and availability with industry solutions like SQL Azure [8], Megastore [3], or Oracle RAC [25] relying on it.

Novel virtualization related techniques such as memory ballooning [4], hot plugging/removing cores or live-migration [11] add flexibility and support for online reconfiguration to Vela.

Although based on known techniques like replication and virtualization, the design of Vela faced several non-trivial challenges. The biggest ones were in designing our system such that it can seamlessly take advantage of both the size and number of cluster machines over which it is deployed. First, handling the two dimensions (of size and number) separately makes reasoning and taking decisions (e.g., deployment, online reconfiguration) about the system more difficult. Virtualizing the hardware reduces the complexity with a minimal cost in resources and performance. Further, having a homogeneous view of the machines allowed us to define an API for the online reconfiguration of the system. Second, managing a dynamic set of replicated OTS database instances over a virtualized cluster required a coordinator component that implements the replication logic, monitors the deployment, and automatically takes decisions in order to react to load changes. Third, addressing performance and functional miss-matches between databases and virtual machine monitors required engineering effort in order to achieve good scalability and support the reconfiguration API.

The performance evaluation of the system, relying on standard workloads (e.g., TPC-W and TPC-E), shows that it scales both on large servers and on clusters, it supports multi-tenancy and can be dynamically reconfigured at run-time. In scalability tests, spanning two dozen nodes, the system handles more than 15k transactions per second servicing transactions for 400 clients, while maintaining response times less than 50ms. The multi-tenancy experiments show that the system can easily accommodate 20 collocated tenants while maintaining response times below 400ms under an aggregated load generated by 800 clients.

The main contributions of the paper are the following:

- To the best of our knowledge, Vela is the first solution that treats clusters and multicore machines as a uniform pool of resources for deploying and managing a replicated database system, an architectural step necessary to operate in modern cloud systems.
- Vela demonstrates how to combine virtualization and replication such that the benefits of both can be simultaneously achieved: a scalable system that does not have a static deployment and can be reconfigured at runtime.
- The evaluation of the system illustrates many important technical design choices that lead to a system that is generic (handling different read- intensive workloads), scalable, online reconfigurable and supports multitenancy.

2 Motivation

The main argument in favor of using Off-the-Shelf (OTS) databases in cloud setups comes from the shortcomings of existing cloud-ready solutions. Existing solutions like Dynamo [13], Dynamo DB or Bigtable [10] require the applications running on top to implement logic that traditionally resides in the database. This includes consistency guarantees, support for transactions over the whole data-sets, or support for complex queries and operators (e.g., join).

There are also counter arguments for running OTS databases in the cloud. Virtualization, elasticity, and online reconfigurability are common traits of cloud systems which do not match the design of OTS databases as these are usually deployed on dedicated servers and statically provisioned with resources (CPU, memory) for peak loads.

In virtualized environments, hot plugging / removing of resources is a common mechanism for online reconfiguration, supported by most hypervisors like Xen or VMWare. While OTS databases, both commercial (SystemX) and open source (PostgreSQL, MySQL) will react to hot plugging or removing of CPUs, they face larger challenges with memory. This comes from the fact that databases take memory management under their own control (to different degrees). For example, commercial SystemX is statically configured with a fixed memory size that serves as both data-cache and as a memory pool used for sorting or aggregation operations. At the other extreme we find PostgreSQL which liberally only takes a small amount of memory (i.e., the "shared_buffer", recommended $\frac{RAM}{4}$) under its control, relying on the OS disk cache for data-caching.

For OTS systems like PostgreSQL, dynamic memory plugging or removing can be used out of the box, while for systems like InnoDB or SystemX, Application Level Ballooning [22, 31] mechanisms are needed.

While OTS databases provide functionality lacking in readily available cloud solutions, a naïve solution does not work out of the box. We show however that with the design of Vela, elasticity, collocation and online reconfiguration can be achieved.

2.1 Replication is not enough

Snapshot Isolation based replication is commonly used in both commercial products (SQL Azure [8] built on top of Cloud SQL Server [5] or Teradata [40]) and research prototypes ([21], Tashkent [14], Ganymed [28]) as a means of scaling read intensive workloads over compute clusters and to increase availability.

Where these systems come short is user controlled dynamic reconfiguration or flexibility. For instance SQL Azure [8] does not emphasize virtualization or flexibility in the replication configuration.

Its Intra-Stamp replication offers fault tolerance and load balancing by having a primary master and two secondary replicas, synchronously replicated. The Inter-Stamp asynchronous replication is used for geo-replication and disaster recovery, as well as a mechanism for data migration.

The authors of [16] investigate the performance of SQL Azure as a black box. The points of interest are the network interconnect inside the cluster (peaking 95% of the time at 90MB/sec) and the cost of a TCP RTT, which is below 30ms for 90% of the samples. Also, under a TPC-E [36] like workload, they observe that the system scales up to 64 clients. In our evaluation we show that Vela scales well beyond this in an environment with a similar network configuration.

While these systems scale their replicas over clusters, they rely on the database engine's parallelization to scale up with the individual resources of each cluster node. As servers have increasingly more cores, many new factors impede traditional relational databases from scaling up. Some of them include contention on synchronization primitives in a database [18, 19], workload interaction in the presence of many real hardware contexts [32] as well as the effects of hardware islands in database deployments on NUMA systems [29].

Scaling up refers to the ability of the system to usefully utilize the resources of a single machine and to yield performance gains as more resources (in our case CPU and memory) are added. Scaling out refers to the ability to utilize and gain performance benefits from adding more machines (cluster nodes) to the system. In most replication based systems, scaling out and up are seen as two orthogonal problems as there is no uniform view of resources within a machine and in the cluster. Using virtualization over clusters of multicore machines, it is possible to define a uniform resource pool, with replication used for scaling both up and out.

2.2 Virtualization is not enough

Favorable arguments for running databases in virtualized environments emphasize consolidation opportunities and dynamic reconfiguration of allocated resources.

Recent work shows a large adoption of virtualized databases for multi-tenancy. Tuning database performance by controlling the encapsulating virtual machine [34] or optimal data placement in the storage layer based on the tenant workload characteristics [26] are commonly investigated.

Virtualizing OTS databases may lead to a degradation in performance. A few studies have investigated this [6], corroborating our experience in building Vela. In most cases, main memory workloads behave very similarly in Bare Metal vs. Virtualized setups, mostly due to Intel's VT-x and AMD's AMD-V support for virtulization. For I/O intensive workloads, virtualized performance degrades and CPU utilization increases, as compared to Bare Metal.

Counter arguments for virtualizing databases hint at resource overheads. The Relational Cloud [12] system makes the case for a database-as-a-service that efficiently handles multi-tenancy, has elastic scalability and supports privacy. The authors argue that achieving multi-tenancy by having a database in a VM architecture is inefficient due to additional instances of the OS and database binaries. In our evaluation of Vela, we have not seen the memory consumption overhead of the OS and database binaries to be an issue, being far lower than that of the data being processed.

We consider virtualization to be a suitable solution being a proved technology that is readily available and that requires no modifications to the existing database engines. It allows fast deployments, easy manageability, implicit resource separation, and offers means for online reconfiguration of the system through techniques like live migration, memory ballooning or hot plugging of CPU cores.

2.3 Dynamic reconfiguration

Dynamic system reconfiguration addresses the system's ability to change (at runtime) at different levels of granularity. Solutions like SCADS [37] or that of Lim et al. [20] do automatic scaling only through coarse grained operations by adding / removing physical servers. DeepDive [24] reconfigures the collocation of VMs in case of performance interference. These solutions do not cover fine grained reconfiguration like hot plugging / removing of memory or cores in VMs. Other approaches study finer grained online reconfiguration. For instance Microsoft SQL Server's dynamic memory configured on top of Hyper-V [30] and "Application Level Ballooing" as a generic memory ballooning mechanism implemented for MySQL [31] demonstrate that memory can be dynamically added or removed from databases running in VMs. However, neither takes into account coarse grained operations.

Complementing mechanisms for online reconfiguration, research looks at policies that drive them. For example, dynamic resource allocation for database servers for proportioning caches and storage bandwidth from a virtual storage server is described in [35].

Besides virtualization, there are also custom resource management and task scheduling solutions for large data-centers. Projects like Omega [33] or Mesos [17] are generic frameworks for running varied jobs in data-centers. They are not designed for relational data processing and might require re-engineering the database engines in order to be suitable for running on top of them. Both projects avoid virtualization for the purpose of managing the global set of resources in a data center, arguing that seemingly small resource overheads can be huge at scale.

In contrast to most of the existing cloud database systems for OTS databases, Vela emphasizes scaling both up and out and has an API for online reconfiguration. Instead of relying on data replication for durability, it uses replication for improving latency and scalability, decoupling the update workload from the read-only workload. Vela expands on using virtualization for supporting online reconfiguration with minimal overhead in resource utilization and performance.

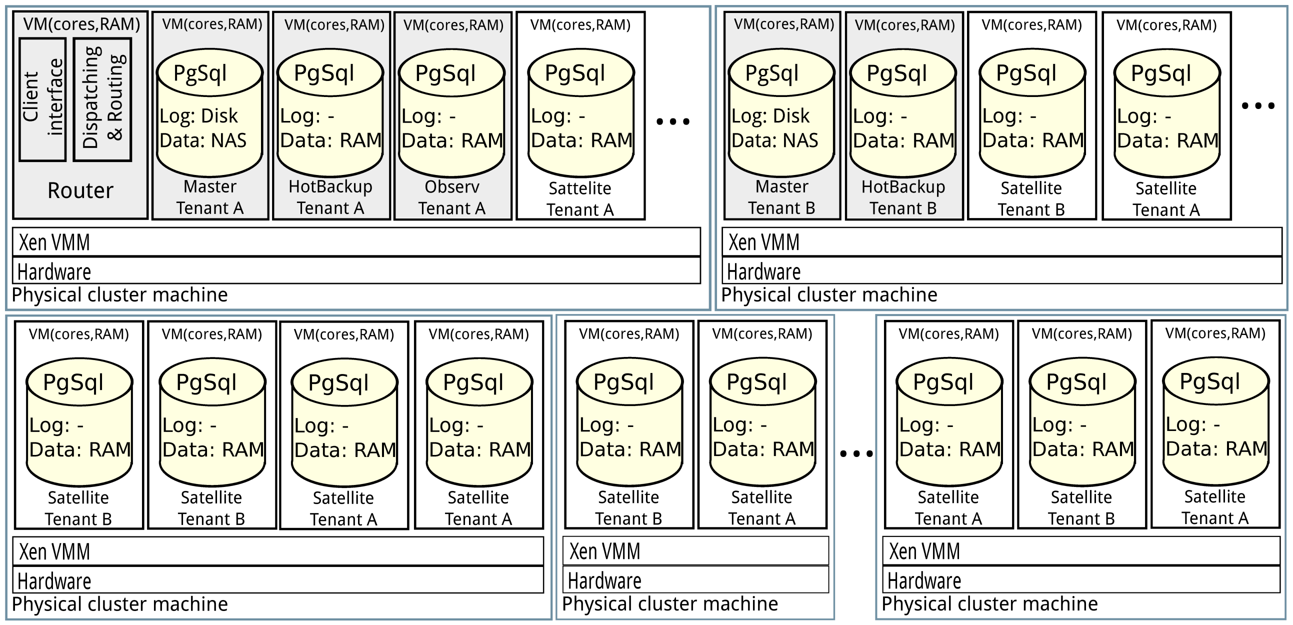


Figure 1: Architecture overview: an example deployment over a cluster of multicores

3 System architecture

Vela achieves scalability both within the boundaries of the same multicore machine and in a cluster of multicores by relying on single master data replication, similar to the Intra-Stamp replication of WAS [8] or Ganymed [27].

In this replication model, we define a "master" copy of the data and a series of replicas ("satellites"). All update transactions go to the master while read transactions go to the satellites, which are kept in sync with the master. Upon committing an update transaction, the master's server count number (SCN) is incremented and its change set is extracted and propagated to all satellites along with the SCN. Satellites increment their SCN after applying the changes from the master in the same order. Each transaction is tagged with the current SCN of the master when entering the system.

The system guarantees snapshot isolation as its consistency level. When using snapshot isolation the queries are guaranteed to see all changes that have been committed at the time the transaction they belong to started (a form of multiversion concurrency control that is present in many database engines such as Oracle, SQLServer, or PostgreSQL).

3.1 Replication

Vela extends the design of Multimed [32], a similar system designed for scaling transactional data processing on large multicores, though lacking support for scaling out over clusters, virtualization and dynamic reconfiguration. Figure 1 illustrates the components of the system, in a possible deployment over a cluster of multicores. At a high level Vela is composed of different Data Processing Instances (DPIs) and a Router.

The **Router** accepts requests and returns responses to and from the clients.

Worker threads dispatch requests to the databases in the Data Processing Instances (DPIs). The Router also extracts the changes to the data on the Master (called WriteSets) and propagates them to the other DPIs in the system. Each DPI is backed up by a database engine holding a copy of the data and is allocated a dynamic set of resources (CPU, memory). Each database within a DPI holds two data components: the actual data and the transaction logs. The options for storing the two can be specified for each replica as: RAM, local disk, or remote storage (NAS).

Like Amazon RDS or SQL Azure, our system uses existing relational database engines in its design. We chose PostgreSQL as the supporting database as it is open-source, with a well understood behavior and shows good standalone scalability on multicore systems. Also, PostgreSQL by its data cache design can react out of the box to memory ballooning operations in VMs. Replacing it with another database engine is straightforward as the only engine specific part is the WriteSet extraction mechanism. This is well described in Multimed [32].

The DPIs in Vela can have different types: Master, Satellite, HotBackup, or Observer. As shown in Figure 1, the Router and each DPI is encapsulated in a virtual machine. Vela supports multi-tenancy by allowing multiple Masters. Each Master DPI has its own set of Satellite, HotBackup and Observer DPIs. In the configuration shown in Figure 1, Vela handles two tenants (A and B), each having a Master, a HotBackup and multiple Satellites. Tenant A also has an Observer DPI.

The **Masters** are primary copies of the data and are configured to offer durability, with the transaction log on a local disk and the data on either local disk or NAS. Alternatively, in a non-durable setup, the Master can be entirely held in RAM. The Router dispatches each tenant's update transactions to its Master.

Satellites hold replicas of the data of a Master. The Router assigns the read-only transactions to Satellites that have the SCN matching the transaction's SCN. When more Satellites fulfill this condition, the one with least load is chosen. The load is defined as the number of current active transactions bound to a Satellite. Satellites do not need to ensure durability (but could be configured to do so).

The **HotBackup** and **Observer** are special and optional types of DPI. HotBackups do not process client requests, though they are kept up to date, receiving all the WriteSets. HotBackups are used for spawning new Satellites at runtime for each tenant. Observers are used to approximate the best response time for the read-only transactions of a tenant. Vela ensures that there is always one read-only transaction routed to each Observer.

The Satellites, HotBackups and Observers do not need to offer durability. As most database engines do not have an option for disabling the transaction log, we emulate this by storing it in RAM and disabling synchronous commits. The overall system durability is given by the Master. While the transaction logs are small enough to be always held in RAM, the actual data might be too large for this. In this case, the data is held on NAS and the available RAM is used for caching. The reconfiguration API supports adjusting the memory of each DPI for increasing or decreasing the memory cache. As PostgreSQL relies on the disk cache for most of its data caching it requires no modifications to be able to adapt to changes in the DPI's memory. For database engines that do their own memory management, a virtulization based solution like Application Level Ballooning [31] can be used.

3.2 Virtualization

In Vela, all the DPIs are spread out over the available resources in a cluster of multicores. We have looked at different options for managing the resources (CPU, RAM) of a cluster of multicores and formulate Vela's requirements for a resource management system:

- *Multiplexing* the hardware for running multiple Data Processing Instances on the same physical machine while allowing for NUMA-awareness. This enables Vela to scale up data processing.
- *Isolation* of CPU and memory allocated to each Data Processing Instance, such that it can make assumptions on the current available cores and memory.
- *Reconfiguration* at runtime of the resources allocated to Data Processing Instances.
- *Uniform* view of the whole cluster. Migrating a Data Processing Instance from one cluster node to another should be seamless and with no service interruption.

In Vela we opted for virtualization (Xen Hypervisor [4, 39]), as it fulfills our requirements. The guest domains (DomUs) run in paravirtualized mode. We have observed no impact on the performance of the Satellites that have the dataset in main-memory. For the Master and the Router that are I/O intensive, the DomUs in their default configuration have quickly become the bottleneck. Similar observations [23] indicate that network I/O can become a bottleneck very fast. *Latency* wise, we were able to tune the system through Linux Kernel network specific settings in both Dom0 and DomU. *CPU* wise, the problem is more involved. For network I/O, Xen allocates in the host domain (Dom0) one kernel thread for each virtual network interface (VIF) in the DomUs. DomUs hosting Routers that do high frequency network I/O were provisioned with up to 8 VIFs. In order to remove the CPU bottleneck of the VIF backing threads in Dom0 we allocated a number of cores equal to the total number of VIFs of all DomUs. Alas, the *interrupts* generated in Dom0 and DomUs need to be balanced over all available cores. Failing to do this leads to one core handling most IRQ requests, becoming a bottleneck.

3.3 Dynamic reconfiguration

Databases deployed on one server are traditionally given exclusive access to the machine's resources. They are generally also over-provisioned to handle peak load scenarios. Vela tries to avoid this by dynamically adjusting the resources of each DPI and the number of DPIs in the system to the load.

We differentiate between two types of reconfiguration operations: DPI level (fine grained) reconfiguration and System level (coarse grained) reconfiguration. At the DPI level, we can reconfigure the provisioned number of cores and the amount of memory. At the System level, we can reconfigure the number of DPI as well as their placement in the cluster. **addCoresToInstance** and **removeCoresFromInstance** control the number of cores allocated to each DPI. They rely on Xen's ability to hot plug cores in a VM. **increaseInstanceMemory** and **decreaseInstanceMemory** control the amount of RAM allocated to each DPI. Increasing / decreasing a DPI's memory uses Xen's memory ballooning. **addInstance** operation is used to spawn new DPIs. A new Master is added to the system for each new tenant, while a new Satellite can be added to mitigate an increase in the load of a tenant. Adding a new Master requires spawning a new VM and starting the Master database in it. Adding a Satellite is more involved. Besides spawning a new VM and starting a new database in this VM, Vela needs to ensure that the database is identical to the one in the Master. Instead of halting requests to the Master database and cloning it, Vela relies on the HotBackup described below. When removing a DPI (**removeInstance**), no more requests are accepted by the Master of the corresponding tenant. When all outstanding requests are served, the Master and all its Satellites are stopped (database then VM). The operation corresponds to removing a tenant from the system. In the case of stopping a Satellite, no more transactions are routed to it and once the outstanding ones complete, the database and the encapsulating VM are stopped and their resources (CPU and RAM) are freed. **moveInstance** operation handles moving a DPI in the cluster from one physical machine to another physical machine. Two types of "move" operations can be performed. A *cold* move will make the DPI unusable during the operation. This is implemented in Vela either by *removeInstance* followed by *addInstance* or through Xen's cold migration. With *hot* moving, the DPI is migrated while it is still serving request, relying on Xen's live migration. *Cold* moves are faster then *hot* moves. As Satellites in Vela do not have any state associated, except for the currently running transactions, they can always be *cold* moved. Masters however can never be taken offline and consequently are always *hot* moved.

The HotBackups enable adding new Satellites without taking the whole system offline. The overhead is minimal in terms of resources: the HotBackup requires only 1 core and main memory for the transaction log. The time it takes to copy the HotBackup into a new Satellite is bound by the network bandwidth. This time only influences the amount of main-memory needed by the queue that temporarily stores pending WriteSets for the HotBackup and new Satellite. The maximum size of the queue can be approximated: $\frac{Tx}{sec} \times \frac{avg(WriteSetSize)}{Tx} \times CopyTime(sec)$. For a tenant processing 10k transactions per second, with average WriteSets of 1KB per transaction (way more than the TPC-W average), deploying a 20GB dataset over a 1GBit network would grow the queue to a maximum of $\approx 1.5GB$ – which is easily manageable.

3.4 Automatic runtime reconfiguration

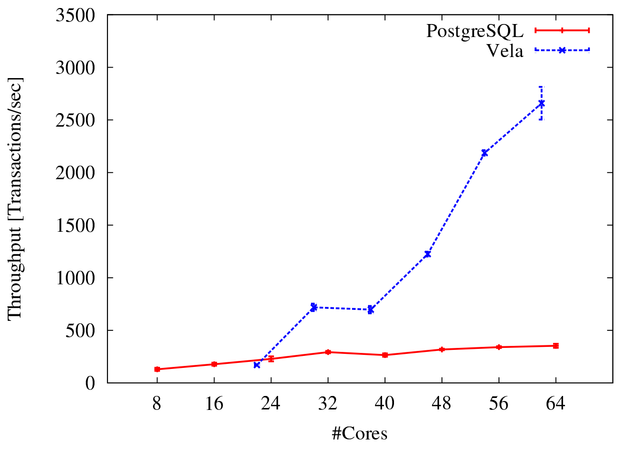
The dynamic reconfiguration of Vela can be done manually or automatically based on monitored metrics. The system reports low level metrics for each VM corresponding to a DPI (e.g., CPU utilizations, memory consumption and disk/network utilizations) as well as per-tenant statistics (e.g., overall throughput, read-only and update transaction latencies, etc.). The system maintains a set of user specified *target functions* that describe threshold values for monitored metrics and simple policies based on the reconfiguration API for achieving these thresholds. Basic tenant SLAs, like desired response time, can be expressed through target functions. The evaluation Section 4.2 exemplifies how the reconfiguration API and target functions work together.

For automatic reconfiguration support for target functions involving system response time, we rely on the Observer for determining the best response time of read-only workload. The Observer is an optional DPI, always deployed on 1 core. The Router ensures that if an Observer is present, it will always have one and only one transaction routed to it. As only one transaction is being executed at a time there is no contention in the Observer. Also as it runs on only 1 core it exhibits no scalability issues. The Observer approximates the best transaction latency for the current workload and reports it as a metric that can be used in target functions.

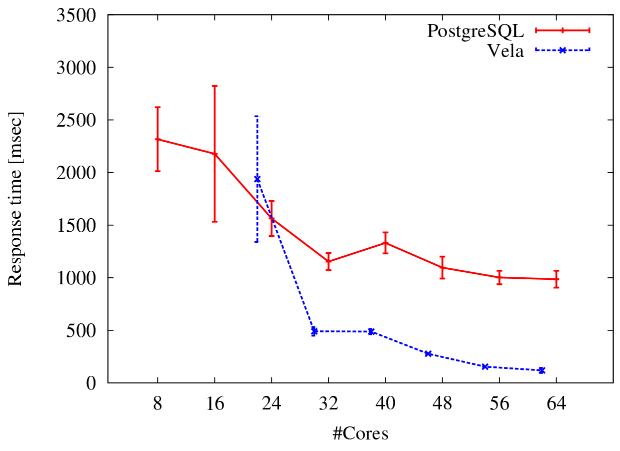
4 Experimental evaluation

This section presents the experimental evaluation of Vela, focusing on the three key aspects of the system: scalability, automatic online reconfiguration and support for multi-tenancy.

For the scalability study of Vela on large multicore machines, we used a 64 core (4 sockets$\times$16 cores) AMD Opteron 6276 (2.3GHz) with 256GB DDR3 RAM (1333MHz). For the scalability study over multiple machines, we used a cluster of 10 servers, each with 16 cores (Intel Xeon L5520 2.27GHz) with Hyperthreading enabled, with 2 NUMA Nodes, and a total of 24GB RAM. As the database engine we opted for PostgreSQL 9.2, used both as a baseline in standalone experiments (in a Native setup) and as the underlying database for Vela (in a Virtualized setup). We chose 2 read-intensive workloads from the TPC-W and TPC-E [36] benchmarks: TPC-WB $\approx$ 95% reads and TPC-E $\approx$ 85% reads. Focusing on the data-processing system, we have omitted the application stack from the benchmark and connect the clients straight to the database engine. Due to similarity in results and space constraints, we only present the results for TPC-WB here.



(a) Throughput



(b) Response Time

Figure 2: TPC-WB – Scaling up with number of cores: Vela vs. Standalone PostgreSQL

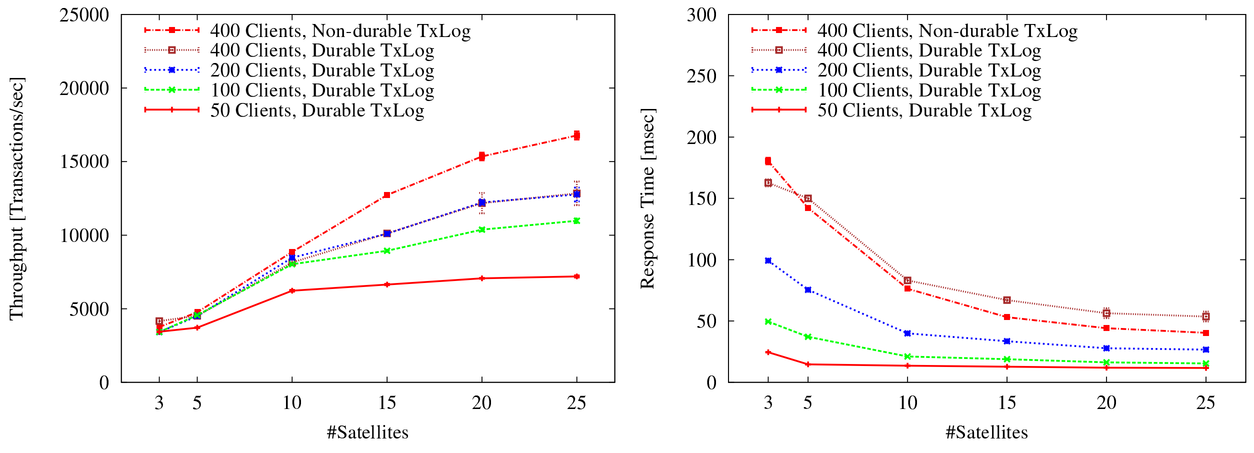
4.1 Scalability study

A main property of Vela is that it needs to scale with the resources it has been allocated. We present a series of experiments showing that Vela indeed seamlessly scales on multicores and on cluster of multicores, under different read-intensive workloads. Vela scales the read transactional workload until an I/O bottleneck is reached.

Scaling up

Figure 2 shows the scalability of Vela compared to that of Native PostgreSQL over a $\approx 23GB$ dataset. Both systems are under a load generated by 200 clients, each sending blocking requests, according to the TPC-WB benchmark. PostgreSQL has difficulties in scaling as number of cores it has at its disposal increases. Increasing by 8 cores at a time, the throughput grows slowly until 32 cores. Due to NUMA effects, after 32 cores, the performance drops, and slightly recovers as more cores are added. Vela's scalability line starts at 22 cores: 2 cores have been allocated to Xen's Dom0, 8 cores to Vela's Router and 12 cores to the Master. All subsequent datapoints correspond to adding additional Satellites, each assigned 8 cores. With the first Satellite, the throughput increases, as read load if offloaded to the Master (since the single Satellite has difficulties in keeping up with the incoming WriteSets). Once the second Satellite is added, performance does not increase as no more read transactions are being routed to the Master, and are handled by the two Satellites that can now keep up with the Master. With subsequent Satellites, throughput increases linearly up to the 5^{th} satellite, point at which the Master's CPU becomes the bottleneck.

It is clear that Vela has an added overhead of resources when deployed on a single machine. The cost of the Router, the Master and the Virtualization layer add up to the used cores and main-memory. Even so, Vela outperforms a traditional database by reducing contention on synchronization and minimizing workload interaction. Vela also exhibits a better scaling trend with the number of cores.



(a) Throughput (b) Response Time

Figure 3: TPC-WB – Scaling out over a cluster of multicores

Scaling out

Using the same architecture Vela also scales over clusters of multicores until either the Master's transaction log disk becomes the bottleneck or the network's bandwidth is fully utilized.

Figure 3 shows Vela's scalability as increasingly more Satellites are added, while blocking clients are issuing requests according to the TPC-WB benchmark. The dataset is $\approx 6GB$. All Satellites are configured to use 4 cores, and hold the data and the transaction logs in main memory. The Master is allocated 14 cores in a VM using up all the cores and memory of a cluster machine (16 total cores with 2 pinned to Dom0). The Master holds the data on NAS with the transaction logs on a local SSD drive, and has the shared buffers large enough to cache the data in main memory. The Router is running on a Native system (without virtualization) on an entire cluster node, in order to avoid the network issues described in Section 3.2.

As we increase the number of Satellites, we expect a linear increase in throughput. First we observe that for 50 and 100 clients, the system is underloaded. With 200 and 400 clients, the throughput is the same (lines completely overlap). At this point the transaction log disk of the Master is the bottleneck. Finally, giving up durability in order to further push the scalability of the system, we moved the Master's transaction log in main memory. With load offered by 400 clients, we see a further increase in throughput. The flattening of the throughput line from 20 to 25 satellites is this time due to reaching a CPU bottleneck on the Master, which can be solved by migrating it to a machine with more cores. At this point, the latency of the read workload cannot be improved and most of the 400 client connections end up being update transactions in the Master.

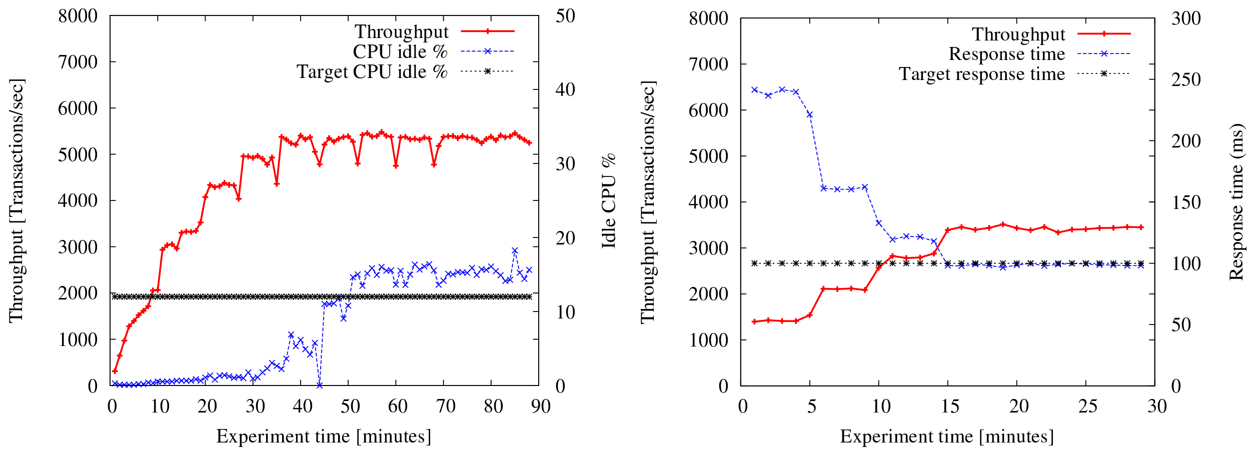
We point out that the flat throughput between 3 and 5 Satellites is caused by contention in the Satellites which cannot keep up with the Master. Consequently the Master also receives read-only transactions, improving the throughput. Scaling both up and out, Vela reaches high throughput rates over a commodity multicore cluster, processing more than 15'000 transactions/sec with latencies less than 50ms.

4.2 Elasticity study

All the experiments in the elasticity study were carried out over the same cluster used in the scale out experiments. This section presents both manual and automatic online reconfiguration of Vela, under a constant load of 200 clients issuing requests based on the TPC-WB benchmark. While the reconfiguration API offers support for shrinking the system, we present policies only for dynamically expanding the system.

Target functions

Manually monitoring and reconfiguring the system is an option. Alternatively, knowing what are the possible bottlenecks, they can be automatically addressed using target functions.



(a) Router core count (b) Satellite count

Figure 4: Automatic system reconfiguration

In Figure 4 (a) we show an experiment in which the system is under a heavy load based on TPC-WB. Knowing that the Router can become a bottleneck under heavy load and as it is CPU intensive, we set a target function on its CPU idle percentage.

Vela monitors the CPU idle % of the Router and uses the reconfiguration API to adjust the system. The target function monitors the CPU idle %. As long as it is less then 12% of the CPU cycles, the Router core count is increased. If no more cores are available on the current cluster node, the Router is live- migrated on a cluster node with more free cores. The reconfiguration stops when the target function is reached or when the live migration cannot be done. The plot shows the system throughput on the primary y-axis, CPU-idle percentage on the secondary y-axis and the experiment time on the x-axis. The throughput grows as more cores are added to the Router. Once the target function of 12% idle CPU is reached, the throughput is stable.

Target functions can monitor response variables that are specific to Vela, not only on generic ones like CPU or memory. Figure 4 (b) shows a target function set on the average response time of the system. The primary y-axis shows the throughput and the secondary y-axis the response time. The x-axis is the experiment time. A target value of 100ms for the average response time is set. The target function works by adding a new satellite (using the *addInstance* API) until the target response time is reached (the $check$ function).

The experiment starts with Vela configured with 1 Satellite of 4 cores. After 2 minutes a new Satellite is added to the system. Within less than 4 minutes the new Satellite receives load. This continues until the 4^{th} Satellite is added, point at which the response time falls under the target value (minute 15 onwards).

In this approach a cut-off value was supplied. This value is unfortunately workload dependent. Using the Observer described in Section 3.4, we monitor the best attainable response time for the current hardware, software & workload combination. The target for Vela can be set to a percentage of this value, making the system agnostic of the workload. We denote the optimal response time continuously measured in the Observer as $O(RT_t)$. Assuming that the Satellites and the workload are uniform, we probe the response time from a Satellite to get the runtime value $V(RT_t)$ at time t. The new $check$ function returns $V(RT_t) > \frac{100+tolerance}{100} \times O(RT_t)$, indicating if the overall system response time is matching the attainable response time, with the given $tolerance$.

```
TESTNEXTPACKAGE()
1  if TESTINCREASE($crtCores$)
2    then $crtCores++$
3    else return $FULL$
4  if PERFGAIN($crtCores$)
5    then return $OK$
6    else $crtCores--$
7  return !$OK$

ADDSATELLITES()
1  while CHECK($O, V$) and HAVEHW()
2  do ADDSATELLITEINSTANCE()
3  if ! CHECK($O, V$)
4    then return $DONE$
5  return WARN($ProvisionHardware$)

FILLPACKAGE()
1  while $crtCores < coresInCrtPkg$
2  do $crtCores++$
3  if ! CHECK($O, V$)
4    then return $DONE$
5  return !$DONE$

ADAPT()
1  if ! CHECK($O, V$)
2    then return $DONE$
3  repeat
4        if ! CHECK($O, V$)
5          then return $DONE$
6    until $OK ==$ TESTNEXTPACKAGE()
7  return ADDSATELLITES()
```

Figure 5: Automatic satellite scaling pseudo-code

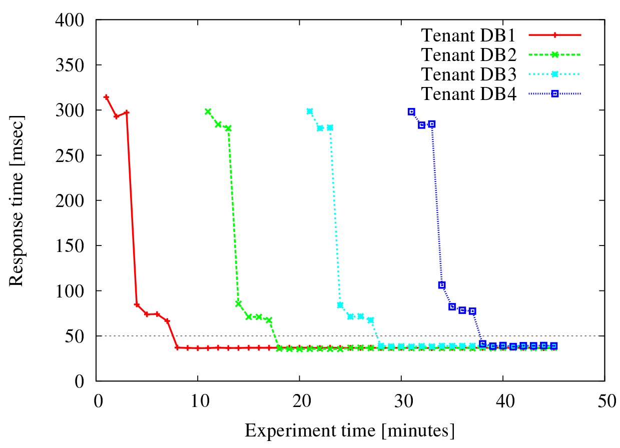
Automatic satellite scaling

Given a cluster of multicores we have shown that Vela can scale both up and out. One tradeoff has not be discussed so far: should Vela opt for few replicas with many cores (fat replicas) or for many replicas with few cores (thin replicas). Few fat replicas are easier to manage and require overall less RAM but will hit the scalability issues of individual databases. Many thin replicas increase the overall RAM consumption and transaction routing time. Each Satellite should be run at the optimal size with respect to the number of cores. Unfortunately the sweet spot depends on the hardware topology, database software and current workload. Based on empirical experience, we present an algorithm that adjusts the number of cores in a Satellite. It assumes that all Satellites are uniform and that the clients of each tenant have the same workload distribution. On the other hand it does tolerate changes in the workload, as long as all clients exhibit the same change.

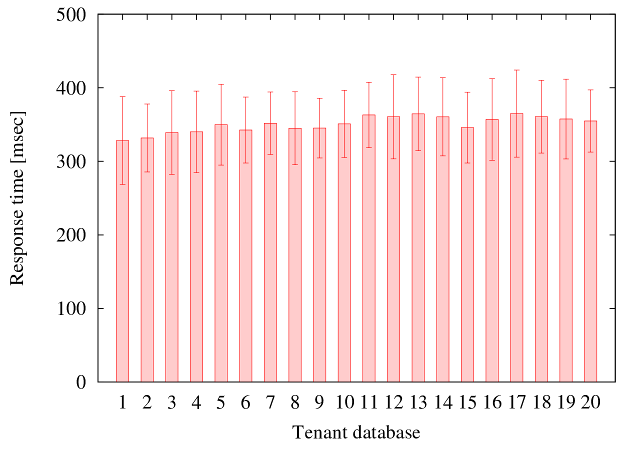
From the experiments that we conducted, we observed that: (1) cache locality matters; and (2) cores that share the same cache behave similarly. The *adapt* function described in Figure 5 controls the process. Starting with each Satellite having 1 core, we expand it to all the cores sharing the same L2 cache (first package). If the required latency is not met, we expand to the cores sharing the LLC cache and then to different NUMA nodes. When moving from one level to another, we "probe" to see if a speedup is gained or not (*testNextPackage*). If no performance gain is obtained, then we stop increasing the cores. Otherwise we go ahead and allocate all the cores of that level to the Satellite (*fillPackage*). Once the size of a Satellite is determined, more Satellites of the same size are spawned (*addSatellites*) in an attempt to satisfy the $check$ function.

4.3 Multitenancy study

Scaling and dynamic reconfiguration in Vela work on a per-tenant base. This section shows that multiple Masters (each with its set of Satellites) can be configured under the same deployment of Vela. The dynamic reconfiguration plays an even larger role in this case. Satellites can be scaled for each tenant.



(a) Adapting to load



(b) High tenant concurrency

Figure 6: Multitenancy supporting auto-reconfiguration

Figure 6 (a) shows 4 tenants deployed in Vela. Each has its own Master and 1 Satellite with 1 core. Each tenant specified its target response time of 50ms. The plot shows the response time of each database tenant during the experiment time. The load is generated by 50 clients for each tenant and is based on the TPC-WB workload mix. The fast drop in response time captures the effect of increasing the Satellite cores from 1 to 4. This is insufficient to meet the latency target and Vela adds a new 4 core Satellite for the tenant (which takes about 5 minutes). At this point the target latency is met and the system stabilizes. Under similar load conditions the tenants behave similarly and in isolation. Moreover, Vela can use the set of cluster nodes that it manages to dynamically add Satellites for any of the tenants.

Figure 6 (b) shows Vela managing 20 tenant databases, each under a constant load from 40 clients (again, TPC-WB). All Masters are collocated on the same cluster node and each tenant has 1 Satellite. Within the standard deviation, all tenants perform similarly. The variation in response time is caused by contention on the transaction log storage of the cluster node holding the Masters.

5 Conclusions

We have presented Vela, a replication based system that scales Off-the-Shelf database engines both on large multicores as well as on clusters, using a primary master model. The system relies on virtualization for achieving a uniform view of all the available resources and also for supporting both fine and coarse grained online system reconfiguration. We also showed that multi-tenancy support is easily achieved by collocating database in the system.

Future directions for Vela include support for diverse Satellites optimized for certain transactions (e.g., heterogenous storage engines, partial replication, different indexes among replicas) or adding functionality to tenants (e.g., time-travel, sky-line or full text search operators) without affecting their performance. The applicability of specialized Satellites is not limited to performance and functional aspects. Vela could also be used for Byzantine fault tolerance, as proposed for heterogenous database replication systems [38, 15].

As shown in our evaluation, Vela handles diverse read-intensive workloads and exhibits little overhead from virtualization. Paying attention to engineering aspects, both in implementing and deploying the system, we were able to achieve good scalability and high throughput rates while supporting transactional workloads with snapshot isolation guarantees.

References

[1] Ashraf Aboulnaga, Kenneth Salem, Ahmed A. Soror, Umar Farooq Minhas, Peter Kokosielis, and Sunil Kamath. Deploying Database Appliances in the Cloud. *IEEE Database Engineering Bulletin*, 32:13–20, 2009.

[2] Amazon RDS Multi-AZ Deployments `http://aws.amazon.com/rds/multi-az/`.

[3] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proc. of the 2011 Conference on Innovative Data system Research*, CIDR'11, pages 223–234, 2011.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. of the 19th ACM symposium on Operating systems principles*, SOSP'03, pages 164–177, New York, NY, USA, 2003. ACM.

[5] Philip A. Bernstein, Istvan Cseri, Nishant Dani, Nigel Ellis, Ajay Kalhan, Gopal Kakivaya, David B. Lomet, Ramesh Manne, Lev Novik, and Tomas Talius. Adapting Microsoft SQL Server for Cloud Computing. In *Proc. of the 27th IEEE International Conference on Data Engineering*, ICDE'11, pages 1255–1263, Washington, DC, USA, 2011. IEEE Computer Society.

[6] Sharada Bose, Priti Mishra, Priya Sethuraman, and Reza Taheri. Benchmarking Database Performance in a Virtual Environment. In *Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 167–182. Springer Berlin Heidelberg, 2009.

[7] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proc. of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[8] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, SOSP'11, pages 143–157, New York, NY, USA, 2011. ACM.

[9] David G. Campbell, Gopal Kakivaya, and Nigel Ellis. Extreme Scale with Full SQL Language Support in Microsoft SQL Azure. In *Proc. of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD'10, pages 1021–1024, New York, NY, USA, 2010. ACM.

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

[11] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proc. of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[12] Carlo Curino, Evan Jones, Raluca Ada Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. Relational Cloud: A Database Service for the Cloud. In *5th Biennial Conference on Innovative Data Systems Research*, CIDR'11, Asilomar, CA, January 2011.

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP'07, pages 205–220, New York, NY, USA, 2007. ACM.

[14] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. Tashkent: Uniting Durability with Transaction Ordering for High-performance Scalable Database Replication. In *Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys'06, pages 117–130, New York, NY, USA, 2006. ACM.

[15] Rui Garcia, Rodrigo Rodrigues, and Nuno M. Preguiça. Efficient middleware for byzantine fault tolerant database replication. In *Proc. of the 6th ACM European Conference on Computer Systems*, EuroSys'11, pages 107–122, New York, NY, USA, 2011. ACM.

[16] Zach Hill, Jie Li, Ming Mao, Arkaitz Ruiz-Alvarez, and Marty Humphrey. Early Observations on the Performance of Windows Azure. In *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC'10, pages 367–376, New York, NY, USA, 2010. ACM.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[18] Ryan Johnson, Manos Athanassoulis, Radu Stoica, and Anastasia Ailamaki. A New Look at the Roles of Spinning and Blocking. In *Proc. of the 5th International Workshop on Data Management on New Hardware*, DaMoN'09, pages 21–26, New York, NY, USA, 2009. ACM.

[19] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proc. of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT'09, pages 24–35, New York, NY, USA, 2009. ACM.

[20] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated Control for Elastic Storage. In *Proc. of the 7th International Conference on Autonomic Computing*, ICAC'10, pages 1–10, New York, NY, USA, 2010. ACM.

[21] Yi Lin, Bettina Kemme, Marta Patiño Martínez, and Ricardo Jiménez-Peris. Middleware Based Data Replication Providing Snapshot Isolation. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD'05, pages 419–430, New York, NY, USA, 2005. ACM.

[22] Richard Mcougall, Wei Huang, and Ben Corrie. Cooperative memory resource mamagent via application-level balloon. Patent Application, 12 2011.

[23] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *Proc. of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE'05, pages 13–23, New York, NY, USA, 2005. ACM.

[24] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. Technical report, 2013.

[25] Oracle Database Advanced Replication. `http://docs.oracle.com/cd/B28359_01/server.111/b28326/repmaster.htm`.

[26] Oguzhan Ozmen, Kenneth Salem, Jiri Schindler, and Steve Daniel. Workload-aware storage layout for database systems. In *Proc. of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD'10, pages 939–950, New York, NY, USA, 2010. ACM.

[27] Christian Plattner and Gustavo Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proc. of the 5th ACM / IFIP / USENIX International Conference on Middleware*, Middleware'04, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[28] Christian Plattner, Gustavo Alonso, and M. Tamer Özsu. Extending DBMSs with Satellite Databases. *The VLDB Journal*, 17(4):657–682, July 2008.

[29] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pinar Tözün, and Anastasia Ailamaki. OLTP on hardware islands. *Proc. VLDB Endow.*, 5(11):1447–1458, July 2012.

[30] Running SQL Server with Hyper-V Dynamic Memory. `http://msdn.microsoft.com/en-us/library/hh372970.aspx`.

[31] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. Application Level Ballooning for Efficient Server Consolidation. In *Proc. of the 8th ACM European Conference on Computer Systems*, EuroSys'13, pages 337–350, New York, NY, USA, 2013. ACM.

[32] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database Engines on Multicores, Why Parallelize when You Can Distribute? In *Proc. of the 6th Conference on Computer Systems*, EuroSys'11, pages 17–30, New York, NY, USA, 2011. ACM.

[33] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proc. of the 8th ACM European Conference on Computer Systems*, EuroSys'13, pages 351–364, New York, NY, USA, 2013. ACM.

[34] Ahmed A. Soror, Umar Farooq Minhas, Ashraf Aboulnaga, Kenneth Salem, Peter Kokosielis, and Sunil Kamath. Automatic virtual machine configuration for database workloads. *ACM Trans. Database Syst.*, 35(1):7:1–7:47, February 2008.

[35] Gokul Soundararajan, Daniel Lupei, Saeed Ghanbari, Adrian Daniel Popescu, Jin Chen, and Cristiana Amza. Dynamic Resource Allocation for Database Servers Running on Virtual Storage. In *Proc. of the 7th Conference on File and Storage Technologies*, FAST'09, pages 71–84, Berkeley, CA, USA, 2009. USENIX Association.

[36] TPC, `http://www.tpc.org`.

[37] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements. In *Proc. of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[38] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *Proc. of 21st ACM SIGOPS symposium on Operating systems principles*, SOSP'07, pages 59–72, New York, NY, USA, 2007. ACM.

[39] Xen Hypervisor 4.1, `http://xen.org/`.

[40] Xkoto. `http://www.terradata.com/`.

CumuloNimbo: A Cloud Scalable Multi-tier SQL Database

Ricardo Jimenez-Peris[*] Univ. Politécnica de Madrid
Marta Patiño-Martinez[*] Univ. Politécnica de Madrid
Bettina Kemme McGill Univ.
Ivan Brondino Univ. Politécnica de Madrid
José Pereira Univ. de Minho
Ricardo Vilaça Univ. de Minho
Francisco Cruz Univ. de Minho
Rui Oliveira Univ. de Minho
Yousuf Ahmad McGill Univ

Abstract

This article presents an overview of the CumuloNimbo platform. CumuloNimbo is a framework for multi-tier applications that provides scalable and fault-tolerant processing of OLTP workloads. The main novelty of CumuloNimbo is that it provides a standard SQL interface and full transactional support without resorting to sharding and no need to know the workload in advance. Scalability is achieved by distributing request execution and transaction control across many compute nodes while data is persisted in a distributed data store. In this paper we present an overview of the platform.

1 Introduction

With cloud infrastructure becoming increasingly popular, scalable data management platforms that are able to handle extremely high request rates and/or data sets have started to appear on the market. One development line achieves scalability through scale-out – using clusters of commodity computers and dynamically assigning as many of these servers to the application as are needed for the current load. While this has proven to be tremendously successful for batch analytical processing (through the map-reduce paradigm) and for applications that only require a simple key/value API, scaling out transactional applications has shown to be much more challenging. Business applications are often built on widely used software stacks (e.g. Java EE, Ruby on Rails, etc.), and scaling them out transparently, offering a full-fledged scalable Platform as a Service (PaaS) is much more complex due to the data and transaction management that must be spread across and between different layers in the software stack. For this reason, the most common current industrial solutions fall back to sharding where the data is split into many different partitions and transactions are expected to access only one partition. Some solutions support distributed transactions, but then often rely on standard distributed commit protocols to handle them, making these distributed transactions expensive. Thus, application developers are encouraged to avoid distributed transactions which might require rewriting existing applications. At the same time, finding proper

[*]This work has been partially funded by the Spanish Research Council (MICCIN) under project BigDataPaaS (TIN2013-46883), and by the Regional Government of Madrid (CM) under project Cloud4BigData (S2013/ICE-2894) cofunded by FSE & FEDER and CoherentPaaS and LeanBigData FP7 projects funded by European Commission under contracts FP7-611068, FP7-619606.

partitions that lead to few distributed transactions and an equal load on all servers is just the same challenging. All this requires expertise that many developers and system administrators do not have. Therefore, there is a huge need for a PaaS that allows developers to continue writing applications as they do today, with standard software stacks such as Java EE, relying on relational database technology providing standard SQL and full transactional properties without concerns for data storage scalability needs.

In the last years, several new proposals have appeared as alternatives to sharding [35, 9, 31]. All of them, however, have certain restrictions. For instance, [35] assumes transactions are written as stored procedures and all data items accessed by a transaction are known in advance, [9] has been designed for very specialized data structures, and [31] is optimized for high throughput at the cost of response time.

In this paper, we present CumuloNimbo, a new scalable fault-tolerant transactional platform specifically designed for OLTP workloads. CumuloNimbo provides SQL and transactional properties without resorting to sharding. This means that an existing application can be migrated to our platform without requiring any single change, and all transactional properties will be provided over the entire data set. Our current PaaS prototype follows the traditional multi-tier architecture that allows for the necessary separation of concerns between presentation, business logic and data management in business applications. At the highest level, our prototype supports one of the most common application development environments, Java EE, as development layer. Beneath it, a SQL layer takes care of executing SQL queries. Data management is scaled by leveraging current state of the art NoSQL data stores, more concretely, a key-value data store, HBase. The key-value data store scales by partitioning, but this partitioning is totally transparent to applications that do not see it neither get affected by it. Additionally, the NoSQL data store takes care of replicating persistent data for data availability at the storage level (file system) outside the transaction response time path. Transaction management is located between the SQL and the data store layer, providing holistic transaction management.

A challenge for a highly distributed system in which each subsystem resides on multiple nodes as ours is the cost of indirection. The more subsystems a request has to pass through, the longer its response time. We avoid long response times by collocating components of different layers when appropriate. Furthermore, we make calls to lower levels or remote components (e.g., for persisting changes to the storage layer) asynchronously whenever possible so that they are not included in the response time of the transactions.

In this paper, we give an overview of the overall architecture of CumuloNimbo, the individual components and how they work together. We outline how transaction processing is performed across all software stacks.

2 CumuloNimbo Architecture Overview

The architecture of the CumuloNimbo PaaS is depicted in Figure 1. The system consists of multiple tiers, or layers, each of them having a specific functionality. In order to achieve the desired throughput, each tier can be scaled independently by adding more servers. Nevertheless, instances of different tiers might be collocated to improve response time performance. The whole stack provides the same functionality as the traditional tiers found in multi-tier architectures. However, we have architected the multi-tier system in a very different way.

Application Server Layer The highest layer depicted in the figure, the application layer, consists of application servers (such as Java EE or Ruby on Rails) or any application programs connecting to the SQL query layer via standard interfaces (such as JDBC or ODBC). Transaction calls are simply redirected to the SQL query layer. For our current implementation, we have taken the open-source JBoss application server that we left unchanged. Our architecture allows for the dynamic instantiation of application server instances depending on the incoming load.

SQL query layer The query engine layer provides the standard relational interface (SQL) including the transaction commands (commit, abort, and begin). The query layer itself is responsible for planning, optimizing and executing queries. However, it does not perform transaction management nor is it responsible for data storage.

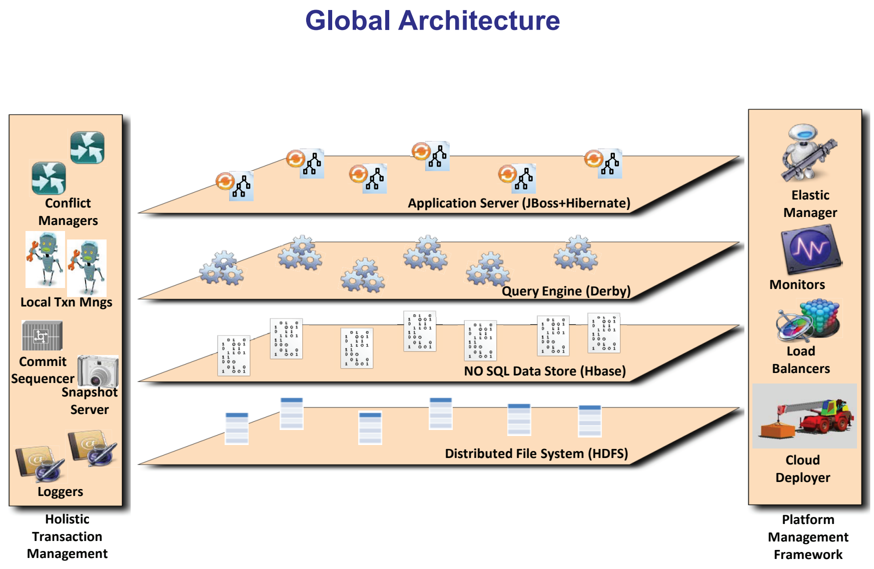


Figure 1: CumuloNimbo Architecture

Our implementation uses the Apache Derby query engine (i.e., compiler and query plan runtime). Substantial parts of Derby have been rewritten to replace the original storage system based on local files with the NoSQL data storage system. This includes the mapping of the relational schema into an appropriate schema of the NoSQL data store, replacing the standard scan operators with scan operators that interact with the NoSQL data store, re-implementing indices, and distributing cost estimation over the SQL query layer and the NoSQL data store. Furthermore, the transactional management in Derby has been replaced by our transaction management.

NoSQL data store layer This layer uses an advanced data store that is designed for inherent scalability in terms of data size and processing requirements. As the currently available NoSQL data stores differ greatly in terms of functionality, the choice of NoSQL data store has a considerable influence of the task distribution between SQL query layer and data store layer. Our current system is based on HBase, an elastic key-value store. The HBase API is provided in form of a library to the application, which in our case is an instance of the SQL query layer, i.e., a Derby instance. In the following, we refer to this library as HBase client as it resides on the client.

HBase offers a tuple interface, i.e., it has the concept of tables (HTable) with a primary key and a set of column families. Having column families allows separate storage and tuple maintenance of each family. HBase already provides advanced query facilities such as simple scans that return all tuples of a table on an iteration basis, and predicate filters on individual columns of a table. Our Derby extension takes advantage of the HBase features to push parts of the query execution to the NoSQL data store layer in order to reduce the amount of tuples that have to be transferred over the network, and to implement secondary indices as tables in HBase.

HBase splits tables horizontally into regions, each region consisting of rows with continuous primary key values. Regions are then distributed onto region servers. HBase also maintains internally an in-memory cache for fast execution. It sits on top of a parallel-distributed file system, HDFS [4], that provides persistent and fault-tolerant data storage. Our CumuloNimbo system can take the default HDFS implementation as all additional semantics needed is embedded in HBase.

To provide durability, HBase persists each incoming update to its write-ahead log (also stored in HDFS). When an HBase server fails, its regions are re-assigned to other servers. These servers run a recovery procedure that replays lost updates from the log, bringing these regions back to a consistent state. For performance reasons, HBase allows the deactivation of a synchronous flush of the write-ahead log to HDFS, so that the server may return from an update operation before the update is persisted to HDFS. In this case, the recovery of HBase cannot guarantee that all updates executed before a server failure will be recovered.

We have extended HBase to help providing transactional guarantees both in terms of isolation and durability while at the same time offering low response times for demanding OLTP workloads.

Generally, it is possible to use other scalable data stores at this layer. The choice might affect the binding to the SQL query layer but it should not have any influence on the other layers.

Configuration In principle, each layer can be scaled individually, that is, one can instantiate as many application server instances, query engine instances and HBase region servers as needed. However, depending on the configuration and execution costs of the components some components can be collocated on the same machine. In particular, it might make sense to put an instance of the application layer and an instance of the SQL query engine (together with its HBase client) on the same node. Then, communication between these instances remains local. What is more, the SQL query engine in this case is run in embedded mode avoiding all remote invocations to it. If large data sets have to be transferred, this collocation avoids one indirection that can be costly in terms of communication overhead.

In contrast, HBase servers are always independent of these two upper layers, as this is where the data resides. Given that OLTP workloads have stringent response time requirements, one possible configuration is to have as many region servers as necessary to hold all data, or at least all hot data, in main memory.

Transaction management Transactions are handled in a holistic manner providing full ACID properties across the entire stack. In particular, we made the design decision to invoke our advanced transaction management from the NoSQL data store, therefore enabling transactions for both SQL and NoSQL applications of the same data in case parts of the application do not require the upper layers of the stack. Holistic transaction management relies on a set of specifically designed transaction subsystems each of them providing a specific transactional property. The fundamental idea is that scaling out each of the transactional properties individually is not that hard. The difficulty lies in ensuring that they work together in the correct way.

Platform management There is an additional component taking care of platform management tasks such as deployment, monitoring, dynamic load balancing and elasticity. Each instance of each tier has a monitor collecting data about resource usage and performance metrics that are reported to a central monitor. This central monitor provides aggregated statistics to the elasticity manager. The elasticity manager examines imbalances across instances of each tier and reconfigures the tier to dynamically balance the load. If the load is balanced, but the upper resource usage threshold is reached, then a new instance is provisioned and allocated to that tier to diminish the average usage across instances of the tier. If the load is low enough to be satisfied with a smaller number of instances in a tier, some instances of the tier transfer their load to the other instances and are then decommissioned. The deployer enables the configuration of an application and its initial deployment. After the initial deployment the elasticity management takes care of dynamically reconfiguring the system to adjust the resources to the incoming load. Figures 2 and 3 show such an adjustment. Figure 2 shows CPU usage and response time of one query engine. As response time goes beyond the agreed level for a predefined time, the system deploys an additional query engine. Figure 3 now shows that the CPU load on both query engines is significant lower and the latency well beyond the service level agreement.

3 Transactional Processing

Efficient and scalable transaction management is the corner stone of our architecture. This section provides a high-level overview of transaction management across the entire stack. Our transactional protocol is based on snapshot isolation. With snapshot isolation, each transaction reads the data from a commit snapshot that represents all committed values at the start time of a transaction. Conflicts are only detected on updates. Whenever

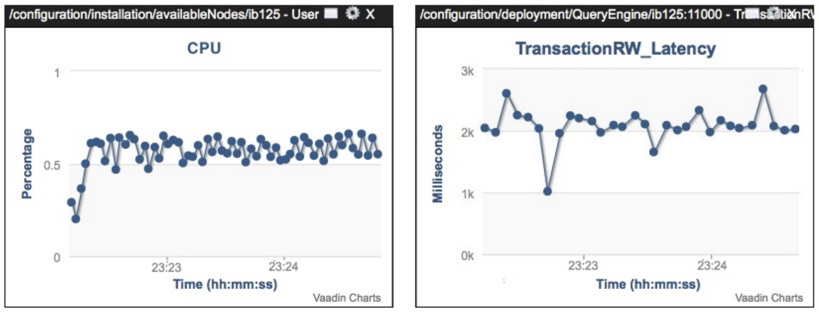


Figure 2: Query engine and latency monitoring

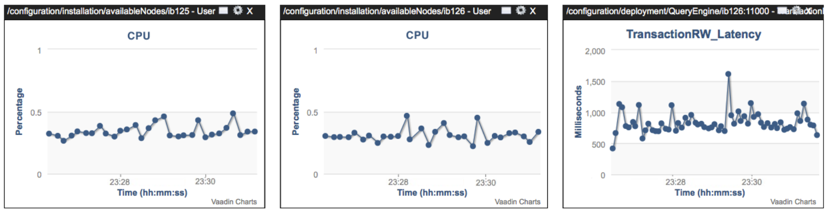


Figure 3: Query engine provisioning

two transactions are concurrent (neither commits before the other starts), and they want to write a common data item, one of them has to abort. Snapshot isolation has been very popular for distributed transaction management as it avoids conflicts between predicate reads and writes.

Transaction Execution We control transaction execution within the HBase layer. With this, transactions can be used directly on HBase if an application does not need the services of the application server and query layers. For that, we extended the HBase client to provide a transaction interface (start, commit/abort). Each read and write request is associated with a transaction. The actual transaction control is embedded in a local transaction manager module that is collocated with the HBase client. Upon transaction start the local transaction manager provides the start timestamp, and the transaction enters the *executing* stage. Write operations create new versions that are only visible to the local transaction that creates them. In fact, we use a deferred-update approach. Upon a write request the HBase Client caches the write locally as private version in a transaction specific cache. When the query engine submits a commit request, the local transaction manager coordinates a conflict check and aborts the transaction in case of conflict with a concurrent transaction that already committed. Abort simply requires discarding the transaction specific cache. If there are no conflicts the transaction obtains its commit timestamp, the updates are persisted to a log and a confirmation is returned to the query engine. At this time point the transaction is considered *committed*. The HBase client then asynchronously sends the versions created by the transaction to the various HBase region servers. The HBase region servers *install* the updates in their local memory store but do not immediately *persist* them to HDFS. This only happens asynchronously in a background process. This is ok, as the updates are already persisted in a log before commit. In summary, an update transaction is in one of the following states: *executing*, *aborted*, *committed*, *installed*, *persisted*.

In order to read the proper versions according to snapshot isolation, we tag versions with commit timestamps and whenever a transaction T requests a data item, we return the version with the largest commit timestamp that is smaller or equal to T's start timestamp, or, if T has already updated the data item, with the version in T's transaction specific cache. As we write versions to the HBase region servers only after commit, start timestamps

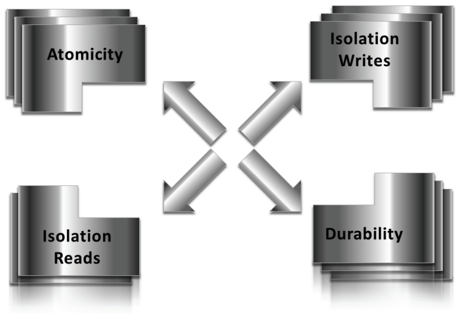


Figure 4: Scaling and distributing transactional tasks

are chosen so that we can ensure that all data that needs to be read is actually installed.

A crucial aspect of our approach is that we take the expensive flushing of changes to the data store out of the critical path of the transaction while still guaranteeing that transactions see a consistent snapshot of the data.

Distributing transaction tasks There are as many local transaction managers as there are HBase clients (i.e., as there are query engine instances), each managing a subset of all transactions in the system. Thus, a local transaction manager cannot independently create timestamps or perform conflict checks. Instead, these tasks are performed by highly specialized components each of them being scaled and distributed independently (see Figure 4): loggers, timestamp managers, conflict managers. These components interact with the local transaction managers in a highly optimized way to keep message volume, message delay and the tasks to be performed by any individual instance as small as possible.

Failures Being a highly distributed system, many different kinds of failures can occur: a query engine together with the HBase client and the local transaction manager can fail, an HBase region server can fail, or any of the transaction components can fail. We have implemented failure management in a recovery manager which is described in detail in [2]. When an HBase client fails, the recovery manager's recovery procedure replays from the transaction manager's log any of the client's transactions that were committed but not yet completely installed in all participating servers. This is necessary because our client holds the updates until commit time, so we may lose the committed updates if a client failure occurs before or during the install phase. Note that updates that were not yet committed are permanently lost when the client fails. These transactions are considered aborted. This is not a problem because only the durability of committed transactions must be guaranteed.

When an HBase server fails, first HBase assigns the server's region to another server and executes its own recovery procedure. Then CumuloNimbo's recovery procedure replays all committed transactions that the failed server participated in that were installed by the client but not persisted to HDFS before the server failure occurred. This is necessary because updates are persisted asynchronously to HDFS, so we may lose updates if a server failure occurs before or during the persist phase. Note that for transactions that were fully persisted before the crash, we can rely on HBase's own recovery procedure to guarantee the durability of the transaction.

During normal processing, HBase clients and HBase servers inform the recovery manager on a regular basis of what they have installed and persisted, respectively. This information is sent periodically to reduce the overhead and avoid that the recovery manager becomes a bottleneck. It serves as a lower bound on which transactions have to be replayed in case of recovery.

4 Query engine

While the query engine layer has no tasks in regard to transaction management, it is the component that enables the execution of arbitrary SQL queries on top of a key-value data with limited query capabilities. Now, we shortly outline how we couple Derby with HBase.

We have taken some of the components of Derby and adjusted and extended them to our needs: SQL compiler and optimizer, and generic relational operator implementations. The major modifications that we performed were (1) a reimplementation of major data definition constructs in order to provide an automatic mapping from the relational data model to the one of HBase; (2) the implementation of sequential and index scan operators to use HBase; (3) implementation of secondary indexes within HBase. Finally, we deactivated Derby's transaction management and integrated it with the our transaction management. This last part was fairly easy as the only task of the query engine is to forward any start and commit requests to the HBase client. In the following we briefly describe these modifications. More details can be found in [37].

Data Model Mapping The relational data model is much richer than the key-value data model and a non-trivial mapping was required. HBase offers a table interface (HTable), where a table row consists of a primary key and a set of column families. Data is maintained in lexicographic order by primary key. A table is split into regions of contiguous primary keys which are assigned to region servers.

We have adopted a simple mapping. Each relational table is mapped to an HTable (base table). The primary key of the relational table becomes the (only) key of the HTable. The rest of the relational columns are mapped as a single column family. We believe that this approach is plausible for OLTP workloads[1]. Each secondary index is mapped as an additional HTable that translates the secondary key into the primary key. For unique indexes the row has a single column with its value being the key of the matching indexed row in the primary key table. For non-unique indexes there is one column per matching indexed row with the name of the column being the matching rows key. Ideally, the data in the secondary index is sorted by the key value as we can then restrict the rows for a given scan by specifying start and stop keys. The problem is that HBase stores row keys as plain byte arrays which do not preserve the natural order of the data type of the key (e.g., char, date, integer, etc.). Therefore, we developed a specific encoding from key values into plain bytes that preserves the lexicographic byte order of the original data type. For composite indexes, we encode multiple indexed columns in the same byte array through concatenation using pre-defined separators.

Query Execution Since the interaction between the query engine and the key-value data store is through the network it is important to keep network traffic and network latency low. Simple database scans of the entire table use the standard HBase scan. Transfer of tuples is done in blocks by the HBase client to avoid individual fetches. When a query contains a range or equality predicate the query optimizer checks whether a secondary index exists on that column. If this is the case, we first access the secondary index table to retrieve all primary keys of all matching tuples (specifying the start and stop keys for the range). Then, we retrieve all the rows of the table with a single query specifying the matching primary keys. When there is no index available that would allow retrieving exactly the matching tuples, we could use an index that provides us with a superset. However, this would mean that we have to retrieve tuples from the base table and check additional conditions in Derby, resulting in unnecessary network traffic. Therefore, in this case, we restrain from using the secondary indexes and perform a scan on the base table using the filter capacity of HBase (i.e., concatenating several single column filters into a conjunctive normal form that is equivalent to the original SQL predicate query). The join operators are completely executed in the query engine as HBase does not offer join operators.

[1]For OLAP workloads that we have not targeted yet, it might be interesting to split the relational columns in different column families to reduce the overhead.

5 Related Work

Several approaches have been proposed to implement ACID transactions in the cloud. Most of the approaches are based on sharding (i.e., partitioning) [8, 35, 7, 40, 17, 14, 16, 33, 13] and some of them even limit transactions to access a single data partition.

CloudTPS [40] supports scalable transactions by distributing the transaction management among a set of local transaction managers and partitioning the data. The list of keys being accessed by a transaction must be provided at the beginning of the transaction. Microsoft SQL Azure [8] supports SQL queries but requires manual data partitioning and transactions are not allowed to access more than one partition. ElasTraS [17] provides an elastic key-value data store where transactions execute within a single data partition (static partitions). Dynamic partitioning is also supported using *minitransactions* [1] which piggyback the whole transaction execution in the first phase of the two-phase commit protocol executed among data partitions. Google Megastore [7] allows programmers to build static entity groups of items (sharding). Megastore provides an SQL-like interface and is built on top of Bigtable [11] providing transactional properties on an entity group. Two-phase commit is used, although not recommended, for cross entity group transactions.

Calvin [35] is a fault-tolerant transactional system that provides ACID transactions across multiple data partitions. Serializability is achieved by resorting to a deterministic locking mechanism that orders transactions before they are executed based on the data they are going to access (read and write sets). Relational Cloud [14] provides automatic workload partitioning [15] and distributed SQL transactions. G-Store [16] provides transactions over partitioned groups of keys on top of a key-value store. Groups are dynamically created. Granola [13] uses a novel timestamp-based coordination mechanism to order distributed transaction but is limited to one-round transactions which do not allow for interaction with the client.

After the limitations presented in [10] several approaches were suggested for providing transaction isolation on top of a scalable storage layer [18]. For instance, Deuteronomy supports ACID transactions over arbitrary data [23]. It decouples transaction processing from data storage [26, 25]. Data stores can be anywhere. The system caches data residing on secondary storage and avoids data transfer between the transaction management component and the data store whenever possible. In fact, similar to our approach described in [2], the data store is only updated after transaction commit. With this it exhibits a performance similar to the one of main memory systems. Deuteronomy implements a multi-version concurrency control mechanism that avoids blocking. The design exploits modern multicore hardware. Google Percolator [31] provides transactions with snapshot isolation on top of Bigtable. Percolator provides high throughput and is intended to be used for batch processing. Omid follows a similar approach providing transactions on top of HBase [19]. In [30, 29], serializable executions are guaranteed in a partial replicated multi-version system. Update transactions, however, require a coordination protocol similar to 2-Phase Commit. In [36], the authors use a novel form of dynamic total order multicast and rely on some form of determinism to guarantee that transactions on replicated data are executed in the proper order. cStore [38] implements snapshot isolation on top of a distributed data store. At commit time write locks are requested for all the data items updated by a transaction. If all the locks are granted, the transaction will commit. If some locks are not granted, the granted locks are kept, the read phase of the transaction is re-executed and those locks that were not granted are requested again. More recently, Google has proposed F1 [33], a fault-tolerant SQL database on top of MySQL. As reported by the authors the latency of writes is higher than a traditional SQL database (e.g., MySQL) but they are able to scale the throughput of batch workloads. F1 requires sharding and the business applications need some knowledge about it. Spanner [12] is a distributed and replicated database that shards data across datacenters. It provides ACID transactions based on snapshot isolation and a SQL-based interface. Synchronized timestamps are used to implement snapshot isolation. Two-phase commit is used when data from different shards is accessed from a transaction. MDCC [21] also replicates data across multiple data centers providing read committed transactions by default. Replicated commit [27] provides ACID transactional guarantees for multi-datacenter databases using two-phase commit multiple times in different datacenters and Paxos [22] to reach consensus among datacenters. Walter [34] implements transactions for geo-replicated systems for key-value data store. Transactions use 2-phase commit and data is propagated across sites asynchronously. MaaT [28] proposes a solution for coordination of distributed transactions in the cloud eliminating the need for locking even for atomic commitment, showing that their approach provides many practical advantages over lock-based methods (2PL) in the cloud context.

While strong consistency is sufficient to maintain correctness it is not necessary for all applications and may sacrifice potential scalability. This has lead to a whole range of solutions that offer weaker consistency levels, without requiring coordination [6, 24, 3, 5, 20]. Moreover, in [32] the authors developed a formal framework, invariant confluence, that determines whether an application requires coordination for correct execution.

In [39] the authors studied the scalability bottlenecks in seven popular concurrency control algorithms for many-core CPUs. The results shows that all existing algorithms fail to scale in a virtual environment of 1000 cores. However the analysis is restricted to a single many-core CPU machine.

In contrast, CumuloNimbo provides a transparent fault tolerant multi-tier platform with high throughput and low latencies without partitioning data or requiring a coordination protocol. CumuloNimbo is highly scalable by separation of concern and scaling each component individually.

6 Conclusions

This paper presents a new transactional cloud platform, CumuloNimbo. This system provides full ACID properties, a standard SQL interface, smooth integration with standard application server technology, and coherence across all tiers. The system provides full transparency, syntactically and semantically. CumuloNimbo is highly distributed and each component can be scaled independently.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *ACM SIGOPS Symp. on Operating Systems Principles (SOSP)*, pages 159–174, 2007.

[2] M. Y. Ahmad, B. Kemme, I. Brondino, M. Patiño-Martínez, and R. Jiménez-Peris. Transactional failure recovery for a distributed key-value store. In *ACM/IFIP/USENIX Int. Middleware Conference*, pages 267–286, 2013.

[3] S. Almeida, J. a. Leitão, and L. Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *EuroSys*, 2013.

[4] Apache. HDFS. http://hadoop.apache.org.

[5] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32, 2013.

[6] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *ACM Int. Conference on Management of Data (SIGMOD)*, pages 761–772, 2013.

[7] J. Baker, C. Bond, J. C. 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 *Conference on Innovative Data System Research (CIDR)*, pages 223–234, 2011.

[8] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kallan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting Microsoft SQL Server for cloud computing. In *IEEE Int. Conference on Data Engineering (ICDE)*, pages 1255–1263, 2011.

[9] P. A. Bernstein, C. W. Reid, M. Wu, and X. Yuan. Optimistic concurrency control by melding trees. *PVLDB*, 4(11):944–955, 2011.

[10] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *ACM Int. Conference on Management of Data (SIGMOD)*, pages 251–264, 2008.

[11] 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 *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 15–15, 2006.

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 251–264, 2012.

[13] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 223–235, 2012.

[14] C. Curino, E. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational Cloud: A Database Service for the Cloud. In *Conference on Innovative Data Systems Research (CIDR)*, pages 235–240, 2011.

[15] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *PVLDB*, 3(1-2):48–57, 2010.

[16] S. Das, D. Agrawal, and A. El Abbadi. G-store: A scalable data store for transactional multi key access in the cloud. In *ACM Symposium on Cloud Computing (SoCC)*, pages 163–174, 2010.

[17] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems (TODS)*, pages 5:1–5:45, 2013.

[18] A. Dey, A. Fekete, and U. Röhm. Scalable transactions across heterogeneous NoSQL key-value data stores. *PVLDB*, 6(12):1434–1439, 2013.

[19] D. G. Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *IEEE Int. Conference on Data Engineering (ICDE)*, pages 676–687, 2014.

[20] W. Golab, M. R. Rahman, A. A. Young, K. Keeton, J. J. Wylie, and I. Gupta. Client-centric benchmarking of eventual consistency for cloud storage systems. In *ACM Symposium on Cloud Computing (SoCC)*, pages 28:1–28:2, 2013.

[21] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: multi-data center consistency. In *EuroSys*, pages 113–126, 2013.

[22] L. Lamport. The part-time parliament. *ACM Trans. on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[23] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. In *Conference on Innovative Data Systems Research (CIDR 2015)*, 2015.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *ACM Symp. on Operating Systems Principles (SOSP)*, pages 401–416, 2011.

[25] D. Lomet and M. F. Mokbel. Locking key ranges with unbundled transaction services. *PVLDB*, 2(1):265–276, 2009.

[26] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *Conference on Innovative Data System Research (CIDR)*, 2009.

[27] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *PVLDB*, 6(9):661–672, 2013.

[28] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. *PVLDB*, 7(5):329–340, 2014.

[29] S. Peluso, P. Romano, and F. Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *ACM/IFIP/USENIX Int. Middleware Conference*, pages 456–475, 2012.

[30] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *2012 IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 455–465, 2012.

[31] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–15, 2010.

[32] A. F. Peter Bailis, M. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *PVLDB*, 8(3), 2015.

[33] J. Shute, M. Oancea, S. Ellner, B. Handy, E. Rollins, B. Samwel, R. Vingralek, C. Whipkey, X. Chen, B. Jegerlehner, K. Littlefield, and P. Tong. F1: The fault-tolerant distributed rdbms supporting google's ad business. In *ACM Int. Conference on Management of Data (SIGMOD)*, pages 777–778, 2012.

[34] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 385–400, 2011.

[35] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[36] P. Unterbrunner, G. Alonso, and D. Kossmann. High availability, elasticity, and strong consistency for massively parallel scans over relational data. *VLDB J.*, 23(4):627–652, 2014.

[37] R. Vilaça, F. Cruz, J. Pereira, and R. Oliveira. An effective scalable SQL engine for nosql databases. In *Distributed Applications and Interoperable Systems (DAIS)*, pages 155–168, 2013.

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

[39] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8:209–220, 2014.

[40] W. Zhou, G. Pierre, and C.-H. Chi. Cloudtps: Scalable transactions for web applications in the cloud. *IEEE Transactions on Services Computing (TSC)*, 5(4):525–539, Jan. 2012.

ICDE
SEOUL KOREA, 2015
Research issues in designing, building, managing, and evaluating advanced data-intensive systems, and applications

31st IEEE International Conference on DATA ENGINEERING

SEOUL, KOREA | APRIL 13-17, 2015

HTTP://ICDE2015.KR

A leading forum for researchers, practitioners, developers, and users to explore cutting-edge ideas and to exchange techniques, tools, and experiences

HONORARY GENERAL CHAIR
Kyu-Young Whang (KAIST)

GENERAL CO-CHAIRS
Sang Kyun Cha (Seoul National University & SAP Labs Korea)
Guy Lohman (IBM Almaden Research)

PROGRAM COMMITTEE CO-CHAIRS
Johannes Gehrke (Cornell University)
Wolfgang Lehner (TU Dresden)
Kyuseok Shim (Seoul National University)

APPLICATION PROGRAM CO-CHAIRS
Masaru Kitsuregawa (University of Tokyo)
Bongki Moon (Seoul National University)
Jignesh M. Patel (University of Wisconsin)

INDUSTRIAL PROGRAM CO-CHAIRS
Wook-Shin Han (POSTECH)
Divesh Srivastava (AT&T Research)

Important Dates for Research, Industry, and Application Papers
Abstract due: July 29, 2014
Full paper submissions due: August 5, 2014
Notification to authors: October 17, 2014
Final versions due: November 28, 2014

IEEE computer society   Data Engineering

Data Engineering

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name ____________________

IEEE Member # ____________________

Mailing Address ____________________

____

Country ____________________

Email ____________________

Phone ____________________

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

TCDE Chair

Kyu-Young Whang
KAIST
371-1 Koo-Sung Dong, Yoo-Sung Ku
Daejeon 305-701, Korea
kywhang@cs.kaist.ac.kr

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398