University of Minho
School of Engeneering
Informatics Department

Nuno Filipe Pinto Faria

High Performance Data Processing

November 2020

University of Minho
School of Engeneering
Informatics Department

Nuno Filipe Pinto Faria

High Performance Data Processing

Master dissertation
Master Degree in Informatics Engeneering

Dissertation supervised by
José Orlando Pereira

November 2020

COPYRIGHT AND TERMS OF USAGE BY THIRD PARTIES

STATEMENT OF INTEGRITY

RESUMO

À medida que as aplicações atingem uma maior quantidade de utilizadores, precisam de processar uma crescente quantidade de pedidos. Para além disso, precisam de muitas vezes satisfazer pedidos de utilizadores de diferentes partes do globo, onde as latências de rede têm um impacto significativo no desempenho em instalações monolíticas. Portanto, distribuição é uma solução muito procurada para melhorar a performance das camadas aplicacional e de dados. Contudo, distribuir dados não é uma tarefa simples se pretendemos assegurar uma forte consistência. Isto leva a que muitos sistemas de base de dados dependam de protocolos de sincronização pesados, como *two-phase commit*, consenso distribuído, bloqueamento distribuído, entre outros, enquanto que outros sistemas dependem em consistência fraca, não viável para alguns casos de uso.

Esta tese apresenta o design, implementação e avaliação de duas soluções que têm como objetivo reduzir o impacto de assegurar garantias de forte consistência em sistemas de base de dados, especialmente aqueles distribuídos pelo globo. A primeira é o *Primary Semi-Primary*, uma arquitetura de base de dados distribuída com total replicação que permite que as réplicas evoluam independentemente, para evitar que os clientes precisem de esperar que escritas precedentes que não geram conflitos sejam propagadas. Apesar das réplicas poderem processar tanto leituras como escritas, melhorando a escalabilidade, o sistema continua a oferecer garantias de consistência forte, através do envio da certificação de transações para um nó central. O seu design é independente de modelos de dados, mas a sua implementação pode tirar partido do controlo de concorrência nativo oferecido por algumas base de dados, como é mostrado na implementação usando PostgreSQL e o seu Snapshot Isolation. Os resultados apresentam várias vantagens tanto em ambientes locais como globais. A segunda solução são os *Multi-Record Values*, uma técnica que particiona dinâmicamente valores numéricos em múltiplos registros, permitindo que escritas concorrentes possam executar com uma baixa probabilidade de colisão, reduzindo a taxa de abortos e/ou contenção na adquirição de *locks*. Garantias de limites inferiores, exigido por objetos como saldos bancários ou inventários, são assegurados por esta estratégia, ao contrário de muitas outras alternativas. O seu design é também indiferente do modelo de dados, sendo que as suas vantagens podem ser encontradas em sistemas SQL e NoSQL, bem como distribuídos ou centralizados, tal como apresentado na secção de avaliação.

Palavras Chave – Base de dados, Distribuído, Concorrência, Consistência, Transação.

ABSTRACT

As applications reach an wider audience that ever before, they must process larger and larger amounts of requests. In addition, they often must be able to serve users all over the globe, where network latencies have a significant negative impact on monolithic deployments. Therefore, distribution is a well sought-after solution to improve performance of both applicational and database layers. However, distributing data is not an easy task if we want to ensure strong consistency guarantees. This leads many databases systems to rely on expensive synchronization controls protocols such as *two-phase commit*, distributed consensus, distributed locking, among others, while other systems rely on weak consistency, unfeasible for some use cases.

This thesis presents the design, implementation and evaluation of two solutions aimed at reducing the impact of ensuring strong consistency guarantees on database systems, especially geo-distributed ones. The first is the *Primary Semi-Primary*, a full-replication distributed database architecture that allows different replicas to evolve independently, to avoid that clients wait for preceding non-conflicting updates. Although replicas can process both reads and writes, improving scalability, the system still ensures strong consistency guarantees, by relaying transactions' certifications to a central node. Its design is independent of the underlying data model, but its implementation can take advantage of the native concurrency control offered by some systems, as is exemplified by an implementation using PostgreSQL and its Snapshot Isolation. The results present several advantages in both throughput and response time, when comparing to other alternative architectures, in both local and geo-distributed environments. The second solution is the *Multi-Record Values*, a technique that dynamically partitions numeric values into multiple records, allowing concurrent writes to execute with low conflict probability, reducing abort rate and/or locking contention. Lower limit guarantees, required by objects such as balances or stocks, are ensure by this strategy, unlike many other similar alternatives. Its design is also data model agnostic, given its advantages can be found in both SQL and NoSQL systems, as well as both centralized and distributed database, as presented in the evaluation section.

Keywords – Database, Distributed, Concurrency, Consistency, Transaction.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACID Availability Consistency Isolation Durability.
AJITTS Adaptive Just-In-Time Transaction Scheduling.
AOCC Adaptive Optimistic Concurrency Control.
API Application Programming Interface.
AWS Amazon Web Services.

CGSI Collaborative Global Snapshot Isolation.
CPU Central Processing Unit.
CRDT Conflict-free Replicated Data Type.
CSI Causally-coordinated Snapshot Isolation.

DHT Distributed Hash Table.

GCE Google Cloud Engine.
GPS Global Positioning System.
GSI Generalized Snapshot Isolation.

IP Internet Protocol.

JSON JavaScript Object Notation.

LTS Long Term Support.

MRV Multi-Record Value.
MV3C Multi-Version Concurrency Control with Closures.
MVCC Multi-Version Concurrency Control.

NMSI Non-Monotonic Snapshot Isolation.
NTP Network Time Protocol.

OCC Optimistic Concurrency Control.
OLPT Online Transaction Processing.

PCSI Prefix-Consistent Snapshot Isolation.
PSI Parallel Snapshot Isolation.

RAM Random Access Memory.

SQL Standard Query Language.

SSD Solid State Drive.
STAMP Stanford Transactional Applications for Multi-Processing.
STM Software Transactional Memory.

TCP Transmission Control Protocol.
TPC-C Transaction Processing Performance Council - Type C.

YAML YAML Ain't Markup Language.

1

INTRODUCTION

As more and more people are in contact with technology, applications developed reach a larger user base than ever before and users' expectations of how quickly a system should respond have increased. The margin of allowed service failures is also slim, and an application should always be available, independently of hardware failures. Furthermore, larger amounts of data need to be processed in a shorter amount of time than ever before, since long response times can end up in the loss of both clients and money [57]. Finally, the increasing globalization also means that applications must process requests from all over the globe.

To respond to the increasing demands, application developers moved over from monolithic to distributed systems. This is relatively easy to achieve for the processing logic of an application, in view of the fact that it does not need store information. However, when it comes to data, it is not simple to distribute a traditional SQL database, since it is not possible to achieve high availability/scalability without compromising some consistency [13, 21], characteristic of SQL systems. This in turn led to a shift of database system design towards ones that favor availability over consistency, with the emergence of NoSQL stores.

The problem with most NoSQL stores is that, for the most part, if developers want to have stronger consistency or transactional guarantees, they need to explicitly program them directly in the application layer, which leads to complex and error prone code. In addition, because the data models offered by NoSQL stores are usually designed with distribution and scalability in mind, it might be difficult to implement complex use cases with the existing data structures and query syntax (e.g. *join* data). It is no coincidence that analytical workloads, despite often using NoSQL databases to store their large amounts of data, often rely on SQL engines for query processing [12, 15, 25, 35, 44, 61, 105, 115, 117, 120].

The mentioned limitations are responsible for a new shift in database design. NoSQL stores are starting to implement transactional guarantees and supporting stronger consistency models. A similar case happens with the relational model, where both open source and commercial solutions try to bring a higher scalability to SQL stores.

This is an area known as NewSQL[112], that seeks to combine the best aspects of both SQL and NoSQL systems.

1.1 PROBLEM STATEMENT

As both CPU, memory and persistent storage become faster and more affordable [59, 74–76], the main limitation to the performance of strongly consistent distributed databases becomes network latencies, where it can reach upwards of 400ms [119]. Common certification protocols such as *two-phase-commit*, distributed consensus or distributed locking require the exchange of multiple messages and/or the participation of multiple, possibly distant nodes. This in turn causes the mean wait time for an operation to complete to increase, especially in geo-distributed systems. In addition, when serving multiple clients in multiple geographical locations – which is becoming the norm for many companies – one has to make the decision on how a distributed database system is deployed: we can either have the data nodes close to each other, which make synchronization faster at the cost of a higher average latency between clients and servers; or have the data nodes close to the clients but distance from themselves, which reduces client-server latencies but increases synchronization overhead.

Another limitation to consistent databases are concurrent accesses to the same object. Although we can parallelize transactional reads to the same object, using multi-version concurrency control protocols, we cannot do that easily with writes. Concurrent updates will have to either abort, leading to wasted work, or acquire a lock, which leads to contention. Either way, they limit transactional performance. There are already some mechanisms that can help alleviate this problem, as presented in Chapter 2. However, their usage mainly targets avoiding conflicts in eventual consistent systems and not objects that require strong synchronization such as an account's balance or an item's stock.

1.2 OBJECTIVES

This thesis aims at advancing the state of the art in the NewSQL paradigm, by creating new architectures and algorithms to allow database managing systems to support strong consistent models with high horizontal scalability and to improve transactional performance. Namely:

- Reduce the impact of network latency in geo-distributed database systems – design of a distributed database architecture that can scale to many clients in different geographical locations, all while ensuring strong consistency guarantees; furthermore, implementation of the designed architecture using, when possible, already implemented database engines, as a means to reduce overhead and ensure the system's foundations are built on already proven concepts and software; finally, evaluation of the implemented solution and comparison to other alternative architectures, in order to prove its viability to solve the problem in question;

- Reduce the impact of hotspots on objects that require strong consistency guarantees – design of a data structure that allows concurrent writes to the same object to execute, when possible, without aborts or lock contention; in addition, implementation and evaluation of said structure, to demonstrate its trade-offs in different situations.

1.3 CONTRIBUTIONS

The main contributions of this thesis are:

- Reducing the impact of network latency in geo-distributed database systems:
  - The *Primary Semi-Primary* architecture for geo-distributed fully replicated databases that reduces the impact of update propagation latency on transaction response time as observed by clients;
  - An implementation strategy that reuses the query engine and transactional isolation from an existing database engine, demonstrated with PostgreSQL;
  - An experimental evaluation that compares the novel architecture to a single PostgreSQL instance and to other multi-database architectures such as *Primary-Standby*, *Multi-Primary Replica* and *Multi-Primary Shards*.
- Reducing the impact of hotspots that require strong consistency guarantees:
  - The proposal of the *Multi-Record Values* (MRVs) mechanism that improves parallelism on numeric values, including access and maintenance algorithms;
  - A generic implementation for PostgreSQL that can be used to convert a numeric value into an MRV;
  - An experimental evaluation of the MRV architecture with three different benchmarks and with three different database architectures (single-instance, cluster with single-writer instance and cluster with multi-writer instances).

1.4 STRUCTURE OF THE DOCUMENT

This thesis is structured in four chapters: Chapter 2 presents the state of the art related to consistency, scalability and availability in modern database systems; Chapter 3 presents the design, implementation and evaluation of the *Primary Semi-Primary* architecture, developed to reduce the impact of network latency in geo-distributed database systems; Chapter 4 presents the design, implementation and evaluation of the *Multi-Record Values* technique, created to allow concurrent accesses to the same numerical object to execute with low conflict probability; finally, Chapter 5 concludes this thesis, in addition to providing suggestions for future work that builds on the concepts presented in this document.

2

STATE OF THE ART

Database systems developers had the option to either choose a consistent system or a scalable/available one. Research has long aimed to reduce that gap to provide applications with databases without compromises. This section presents an overview of that research, studying isolation/consistency, concurrency control protocols, different methods of distributed database synchronization, reducing conflict probability and various classical and modern database system architectures.

2.1 ISOLATION AND CONSISTENCY

Isolation and consistency are database systems concepts related to how concurrent executing operations are handled and how the data is presented to the clients that access it, respectively. Since they are both related to the correctness of data and operations over it and are often times mixed together by researchers [2], this section does not make an explicit division between the two. Rather, it addresses solutions that combine both concepts in order to provide some degree of correctness guarantees.

The strictest concurrency levels – e.g. strict serializability – ensure that a transaction is not affected by concurrent ones, simulating serial execution. However, these levels have significant impact of both response time and throughput, as they reduce the degree of allowed parallelism. Given that some workloads do not require the strictest guarantees to work correctly, many databases offer varying levels that sacrifice some degree of correctness in favor of higher performance. This includes, but not limited to, Read Uncommitted, Read Committed, Repeatable Read and Snapshot Isolation.

Perhaps one of the most well-known and studied isolation level is the Snapshot Isolation [16], where a transaction reads from a snapshot of all previously committed transactions from when it started and only completes if there are no other concurrent writes. It is implemented using a multi-version concurrency control method (MVCC) [99], where different versions of the same record are provided, together with timestamps that display its validity, allowing non-blocking reads. Snapshot Isolation usage is so widespread that it has been adapted to run under Serializable guarantees [41, 55] and even implemented using physical clocks [36].

A problem of Snapshot Isolation with strict consistency guarantees is that since it requires for a snapshot to contain the results of all previously committed transactions, it forces synchronous replication with transactions executed on different replicas. This, in turn, forces a client to wait before beginning a new transaction. To prevent this, *Generalized Snapshot Isolation* (GSI) [39] extends Snapshot Isolation to allow clients to be able to read an older snapshot. In a distributed environment, there is an instance of GSI that states that the possibly older snapshot will always contain all previous transactions already applied in that respective site, designated *Prefix-Consistent Snapshot Isolation* (PCSI), presented in the same paper. This means that different sites can display snapshots with different delays, but total order is still guaranteed.

Although GSI and PCSI prevent clients from waiting before starting a transaction, the total order guarantees still impact the freshness of a snapshot, since a transaction can only be applied after all the previous ones are applied, leading to higher response time and increased abort probability. Instead of providing total order guarantees, numerous research and systems rely on causal order [64], a more relaxed ordering model. In the context of database systems, a transaction T_1 causally precedes T_2 if T_2 executes over the changes made by T_1. T_1 and T_2 are concurrent if neither T_1 causally precedes T_2 or vice-versa, and no assumption can be made about who committed first. Since causal order should be enough to guarantee the correctness of a large number of use cases, while at the same time proving better performance by removing strict synchronization, Snapshot Isolation has been adapted to run under causal guarantees. Such example is the *Parallel Snapshot Isolation* (PSI). PSI extends Snapshot Isolation by allowing different nodes to apply transactions in different orders, as long as they are conflict-free. This causes for the different replicas' snapshots to evolve independently, which should only be a concern for use cases that require the strictest consistency guarantees.

An alternative to PSI is the *Causally-coordinated Snapshot Isolation* [95] (CSI), with the main difference being in how the causality is ensured. In PSI, this is done based on the timestamp specific to the node the transactions were executed on, and transactions from the same node are applied in the same order they were committed, waiting if necessary. In CSI however, causality is ensured based on a transaction's read and write-sets. This means that in CSI a transaction T_2 causally precedes T_1 if and only if T_2 read or modified the data modified by T_1, while in PSI T_2 can be considered to causally precede T_1 just by starting in a snapshot with the effects of T_1. CSI thus allows for faster transaction replication, since the mean wait time is theoretically reduced, at the cost of increased implementation complexity.

Another alternative to PSI is the *Non-Monotonic Snapshot Isolation* [11] (NMSI), whose main difference is the fact that a transaction may read versions of data committed after it started, in order to reduce aborts due to stale data.

Ensuring global Snapshot Isolation for partitioned systems has also been a topic of research, as is the case with *Collaborative Global Snapshot Isolation* [24] (CGSI). CGSI guarantees that if a transaction T_2 reads the effects of T_1 in some partition, it can read the changes of T_1 on all the other partitions. In other words, the snapshots must be consistent across partitions. To ensure this, it predicts the usage of commit time certification for both read and write-sets of a transaction. CGSI loosens this property by only forcing it to transactions that depend on each other, i.e. intersection between reads and write-sets, or between write-sets, is not empty, since it is irrelevant if some transaction has the changes of another in one partition but not the other if it does not rely on its work. CGSI takes it one step further than PSI and also guarantees that transactions are committed by total order guarantees in all sites, providing a higher degree of consistency. This is to avoid two different snapshots from seeing partial commits that are incompatible due to the changes of two or more non-causal transactions. Although this guarantee could be necessary for some edge cases, it requires transactions to be applied by their total order in all sites, which can lead to higher response time and/or higher abort rate. To achieve total ordering, CGSI predicts the usage of logical clocks belonging to each site and a server id (e.g. IP address) to resolve ties.

All the previous alternatives must rely on some form of pre-commit synchronization between the replicas in order to ensure the desired consistency. If we want the guarantee that a distributed database system not only performs as fast as possible, but also makes it possible to guarantee availability in case of network partitions [21], we need a solution that offers a lower consistency level, e.g. eventual consistency [118]. In this level, database replicas receive and process both reads and writes, committing without any coordination. Eventually, states are merged and causal conflicts are often dealt with using vector clocks and rules such as *last-writer-wins* [113], while concurrent conflicts can use rules defined by the application developers. This provides full availability, since no node depends on any other to write data, at the cost that it might not be enough to ensure the correctness of some applications, without at least increasing applicational code complexity.

For distributed database systems, a similar consistency level to eventual consistency is the *Causal Consistency with Convergent Conflict Handling* (causal+ consistency), ensured by systems such as COPS [71], Eiger [72] and Cure [6]. In addition to resolving conflicts using rules, causal+ also guarantees that if some transaction T_2 depends on T_1, then T_2 is applied after T_1 in every site.

A middle ground between strong and weak consistency is the RedBlue consistency [69]. It proposes tagging operations as blue, if they commute, or red, if not. Blue operations execute locally and are asynchronously replicated to other sites, under eventual consistency. Since they commute, the order they are applied does not affect the outcome. Red operations are serialized across sites, as they do not commute and have to be applied in the same order in all nodes, to ensure strong consistency. Furthermore, some operations that don't commute and would have been tagged red can be modeled in order to commute – e.g. updating a value *by* some delta instead of *to* some constant value. The RedBlue consistency thus allows serializing operations only when explicitly necessary, which can lead to lower response times in distributed systems, especially geo-distributed ones.

2.2 CONCURRENCY CONTROL

Concurrency controls refers to methods that allow the correct execution of concurrent transactions, implemented to ensure some desired isolation level. The simplest concurrency control is to serialize transaction execution, i.e. execute one transaction at a time. This alternative yields the minimum possible throughput, as it does not make use of multiple cores nor use a single core to its full potential (e.g. idle during I/O calls). Some alternatives thus exploit the fact that some transactions update disjoint data and use it to maximize parallelism.

Instead of limiting one transaction to execute in the system at the same time, we can limit one transaction per resource (e.g. row, table, document, ...). This allows transactions that read/write different records to execute at the same time. This widely used concurrency control is known as locking, with *two-phase locking* [18] being a common technique. To further improve parallelism, locking can be divided into shared and exclusive. This way, multiple transactions can read the same value concurrently by acquiring a shared lock. Transactions that need to modify a value must acquire an exclusive lock.

Although locking is a step up over the sequential execution, it still limits parallelism. For example, a transaction cannot read a value that has been exclusive locked by another. In addition, acquiring an exclusive lock could take a significantly long time (i.e. starvation), depending on the number of transactions that acquired/will acquire the shared one. Furthermore, this solution has the potential to lead to deadlocks. These problems motivated the emergence of multi-version concurrency control (MVCC) protocols [17, 99]. In these protocols, multiple versions of the same record are used, labelled with timestamps that display its validity. They thus allow non-blocking reads, even if the record was updated after it started (e.g. Snapshot Isolation). Depending on the implementation, the main problem with MVCC protocols is the storage overhead of keeping track of multiple versions. To alleviate this, tasks that periodically delete obsoleted versions are employed (e.g. VACUUM in PostgreSQL [54]). However, they are computational heavy and can resort to locking entire data structures to operate. In distributed database systems, timestamps are often implemented using logical [64] or vector clocks [70]. Research has also proposed the usage of physical clocks to accomplish this task [4, 27, 36, 37, 40]. However, they must deal with the inevitable problems inherent of skews between clocks of different sites. Apart from frequent synchronization, solutions to alleviate this include making transactions wait [27, 40], providing older snapshots to guarantee consistent reads in partitioned databases [36, 37], providing commit time certification for read-only transactions [3] or guaranteeing a weaker consistency level [4].

MVCC and locking are not mutually exclusive and are often used together to ensure a target isolation level. For example, allowing concurrent reads and a single write to execute but locking concurrent writes (e.g. PostgreSQL [49]). However, a database can also use MVCC without locking. A transaction can execute without synchronizing with others, as if it were only one active. Synchronization can later happen at commit time, where write sets are certified in order to abort transactions that generate conflicts. This method is known as optimistic concurrency control (OCC) [62], since a transaction executes as if is guaranteed to commit. By removing synchronization throughout its execution, a database can ensure lower response times. However, since conflicts are only detected at commit time, a transaction can conflict with another right at its inception and keep executing without knowing, leading to wasted work. In addition, a long running transaction under OCC is subjected to starvation, as abort probability increases as the number of other transactions writes' it as to compare against increases.

OCC has also been an area of research. For example, in order to reduce aborts, there is the *Adaptive Just-In-Time Transaction Scheduling* (AJITTS) [97], that looks to improve it at scheduling level, by controlling the amount of transactions executing at the same time based on the system load. There is also the Multi-Version Concurrency Control with Closures (MV3C) [29], which instead of completely aborting a transaction on conflict, partially re-executes it, rerunning the blocks of code affected by the conflict(s). This is ideal especially for long running transactions. There is also research to improve long lived OCC transactions (in the context of mobile transactions), by trying to abort them before the certification phase, if conflicts were found, with the intent to reduce wasted resources [122]. The concept of physical clocks has also been applied to OCC, as is the case with *Adaptive Optimistic Concurrency Control* (AOCC) [3]. It uses client caching and loosely synchronized clocks that define transaction order to achieve high performance. In addition, AOCC doesn't rely on MVCC, instead it just keeps a single copy each object. This, together with the fact that clocks can skew from each other, makes it so it needs to rely on commit-time certification to detect data inconsistencies, even for read-only transactions. The TicToc [123] optimistic concurrency control algorithm has been proposed in order to enable higher degrees of concurrency. Instead of assigning a timestamp to a transaction at its beginning, TicToc lets the transaction execute without timestamp and, at commit time, provides it one based on the timestamps of its read and write-sets. Then, it checks whether the transaction's reads are still correct based on the chosen timestamp. This allows to commit a larger number of concurrent transactions that could have otherwise been aborted by other concurrent control algorithms. In addition, it also reduces contention of monotonically sequential timestamps in systems that solve conflicts based on *timestamp ordering*, a technique where serialization order is decided before the transaction executes [18]. Finally, since transaction certification can itself be resource heavy, depending on the size of the dataset that has to be compared, Meld takes advantage of a tree structured database in order to model the transaction changes as a tree, and quickly identify if it can commit without needing to process the entire structure [100].

More research has also been invested into improving MVCC in general, with a subset of examples presented in [5, 18, 23, 66].

2.3 DISTRIBUTED DATABASE SYNCHRONIZATION

Any multi-writer database that employs a high consistency must rely on some form of synchronization protocol between the replicas. As they must be applied to some, if not all, executing transactions, they must be designed to complete as fast as possible. This led to the research of various number of methods of synchronization in distributed systems.

The most simple solution is a centralized certifier, where one component decides which transactions to commit and which ones to abort. The main disadvantages are the fact that it creates a single point of failure and could become the bottleneck of the entire system. Therefore, some research such as Meld [100] aim to improve certification's performance. The main advantage is the fact that centralized coordination requires the minimum number of messages exchanged (just a request and reply), which makes it less susceptible to network latencies, especially when geo-distributed environments.

Another method is the *two-phase commit* [17], used mainly to ensure atomic commitment when different nodes hold different subsets of data. In this method, some previously designated node, or one chosen at certification time or even the client itself [124], acts as the coordinator, telling the participant nodes to prepare for commit (*prepare* phase). Each participant then replies to the coordinator with their votes, which can be either commit or abort. Having received all the votes, the coordinator then instructs all participant nodes to either commit – if all votes are commit – or abort – if at least one of the votes is abort (*commit* phase). One disadvantage of this protocol is the fact that it relies on the exchange of multiple messages between coordinator and participants, together with the fact that the coordinator becomes a single point of failure of the certification. In addition, it must also lock objects currently waiting to be committed in order to avoid conflicts of concurrent transactions, which can significantly impact performance both in regular execution and especially if the coordinator or one of the participants crashes. The *two-phase commit* protocol might not recover if both coordinator and a participant fail in the *commit* phase, as the coordinator could have told the participant that crashed to commit or to abort, but not the others, leaving no way for them to know the actual decision. They can neither commit or abort because the participant that crashed could have aborted or committed. To prevent this, the *three-phase commit* [108] protocol has been proposed, that adds a additional phase between *prepare* and *commit*, named *prepare to commit*. If the coordinator and participant(s) fail before the *prepare to commit* phase is completed (i.e. not all nodes received the *prepare to commit* message), the participants know that an irreversible decision has not been made by the node(s) that failed (since no node actually commits at that phase), and the process restarts or the transaction is safely aborted. If the coordinator and participant(s) fail after the *prepare to commit*, all nodes agree that the decision to commit has been made (since they all received the message), and can all safely commit. Despite solving a recovery aspect of the *two-phase commit*, the fact that it requires an extra phase means that a transaction's response time is even further increased.

Distributed consensus, in which nodes vote to agree or disagree on some proposal, is also used to ensure that a transaction has the same effects in multiple nodes in a distributed database system. For example, it can be used to determine the order in which transactions are executed or applied by different nodes, as is the case with active-active replication [56], Granola [28], Spanner [27] and others. In addition, different replicas can execute different transactions and rely on consensus at commit time, to ensure that concurrent conflicts are dealt equally in all replicas, as is the case with MySQL Group Replication [92]. Some well-known examples of consensus protocols used in the context of distributed database systems include Paxos [67] (or Multi-Paxos, in which the leader election phase is done one time instead of once every time the algorithm is executed) and Raft [88]. Having a decentralized protocol ensures that the system provides high availability guarantees. However, just like *two-phase commit*, they require the exchange of multiple messages and the participation of several, possibly distant nodes, which has a significant impact in a transaction's response time. There are several proposals to improve Paxos, namely Fast Paxos [65], where the client request bypasses the leader and goes straight to the entire group, only requiring one round trip if they immediately reach a super-majority. If not, it resorts to the slow execution of Paxos.

To control updates on concurrent records, the locking of resources is also used, i.e. distributed locking. The acquirement of locks can be made in both centralized – e.g. using a central coordinator – and distributed fashions – e.g. using distributed consensus to agree on the order the locks are acquired. The inherent problem with locking resources makes this alternative potentially heavy on geo-distributed systems, given the time a transaction may need to wait.

The passing of a token to, for example, guarantee the total order of concurrent transactions [69, 94, 96], can be seen as a form of locking.

All previous methods are used to ensure pre-commit synchronization, which always end up having some negative impact in a transaction's response time. Alternatively, there is also the post-commit synchronization, relying on replica merges to resolve conflicts. This sacrifices strong consistency (e.g. resorting to eventual consistency [118] or causal+ [71]), since a client can have its committed changes annulled, to allow for a faster execution and optimal availability.

2.4 CONFLICT AVOIDANCE

Write conflicts, especially in distributed database systems, greatly limit performance of strongly consistent systems, by making transactions abort or wait to acquire some exclusive lock. To reduce conflict probability, there has been a large body of research that exploits the properties of some data structures and/or operations in order to allow concurrent update to execute simultaneously.

Such example is a classical solution in centralized systems titled escrow locking [87], that targets numeric values. For subtractions, instead of locking the entire record, a transaction first checks if the current value has the desired quantity and, if true, proceeds to only lock the necessary sub-amount of the total value, thus allowing multiple transactions that update the same field to execute concurrently, improving throughput. There are several disadvantages of using this approach. Namely, it requires synchronizations at two points, one when testing if the value has the required amount and another at commit time. In addition, the synchronizations make this solution only viable in centralized systems, since in a distributed one the cost to acquire the lock now incurs network latency, given that a request to a central monitor that handles that synchronization is necessary once every time a numeric value is updated.

To reduce hotspots on the same record, one can split write-heavy fields with 1-to-1 relations, preventing unnecessary collisions. In addition, there is also the *timestamp splitting* technique [58], that divides a record's columns into multiple subsets and assigns to each a different timestamp, reducing collision probability.

A way to improve transactions executed on different nodes is by making them operate on structures that allow for replica merges in any order without triggering conflicts. Examples of these structures are *Conflict-free replicated data types* (CRDTs) [68, 106], being mainly used in NoSQL systems as result of their non-relational data models (e.g. sets). There are two types of CRDTs: operation based CRDTs, that are based on the replication of commutative operations – for example, instead of simply replicating the final result of a counter, the operations could be replicated instead (increment and decrement). Independently of the order they are merged, the final result will always be the same, since addition and subtraction are commutative; and state based CRDTs, based on the commutative merging of states – for example, instead of replicating all add and subtract operations, the counter could be modeled as a map of replica id to pair with the total added and total decremented values. This way, two counters can be merged by executing the pointwise maximum for each node, that will always be the same independently of the merging order, since both adds and subtracts can only increase. CRDTs cannot avoid, for example, that two clients concurrently purchase the last product in stock. They are instead designed with the objective that all data replicas converge to the same state independently of the order they were merged. Walter also uses a similar type of data structure, called *Conflict-free counting set objects* [109]. This structure models a row as a counting set, where it supports inserting some element – increment the respective counter – and deleting some element - decrementing the respective counter. Records from different nodes are later merged and, since addition and subtraction are commutative, the insert and delete operations never conflict with each other.

Another approach to avoid conflicts of concurrent operations is to transform the operations themselves [38]. An update from a replica that generates a conflict in another can be transformed, based on the state of the latter, in way that ensures both replicas end up towards the same state after the operation is executed. For example, concurrent insertions of characters in the same line in a real-time collaborating text editor. The position of an inserted character can change if another character is inserted to its left. Therefore, a local operation that inserts a character at position x can be applied in other replicas at position y. There is also the proposal of *delta transactions* [110], that aim to convert two transactions that would otherwise conflict into multiple that commute. They are intended to reduce conflicts of updates on numeric values. For example, the author suggests updating an average not *to* some value but *by* some value, for those updates to commute.

The RedBlue [69] consistency also helps reduce conflicts in the context of distributed database systems, since it tags commutative operations that can later be applied in different orders by different nodes.

Finally, for replicated systems, the best optimization one can implement is to build the transactions in such a way that there is no need for distributed certification. One way to achieve this is to partition data access (but not data itself) in a manner that each transaction only need to write on one node, so that each one can commit independently, later replicating the changes. This is the case of Walter with its *preferred sites* technique [109] and SLOG with its concept of object's *home* [101].

2.5 DISTRIBUTED DATABASE SYSTEMS ARCHITECTURES

This section presents how different architectures use the concepts presented in the previous sections in order to implement consistent and scalable distributed database systems.

2.5.1 *Single-writer*

Single-writer database systems contain only a single node that handles writes and commits. Distributing a single-writer database consists in adding more standby nodes, that hold a copy of the data, in what is known as passive replication [56]. In addition to providing higher availability through failover, standby nodes also handle reads, allowing them to horizontally scale. However, since writes are constricted to the primary node, it may become the bottleneck of the system. Furthermore, this means that writes in a geo-distributed system must all contact the same, possibly distance server, leading to high average network latencies. This common architecture is natively supported by the most popular SQL database managing systems, including PostgreSQL [51], MySQL [90], SQL Server [77] and Oracle [93]. It also has support among NoSQL systems, as is the case with MongoDB's replica set [82].

A caveat of this architecture is how the replication is handled. We can either have a synchronous replication, guaranteeing that the most recent data is stored in all replicas at the cost of higher transaction response time, or have an asynchronous one, improving response time but increasing the probability that clients can read older data. Depending on the use case, both options can be used, which is why database systems provide the choice to opt for one or the other [47, 52, 91].

The active replication [56] to ensure high availability can also be seen as a single-writer architecture. Even though technically multiple nodes can write, they all process the same transactions in the same way, which means writes cannot scale out. The main disadvantage when comparing to the above alternative is the fact that transaction execution must be deterministic, i.e. the outcome must only depend on the initial state of the replica (e.g. random functions should be avoided). The main advantage is that if one node fails, the others can still complete the transaction. In the passive replication, if the primary fails, not only the transaction is automatically aborted, but also the crash is visible to the client, which must reissue the request. Depending on the implementation, different nodes can process different read-only transactions, allowing reads to scale.

2.5.2 *Multi-writer*

The reduced scalability of the single-writer architecture lead to the design and implementation of systems that employ multi-writer nodes, to allow both reads and writes to scale and to reduce latencies in a geo-distributed setting. However, multi-writer architectures must deal with trade-offs between consistency and performance.

Some systems allow concurrent transactions to execute under optimistic concurrency control on different full-replicated nodes and employ commit time certification to ensure strong consistency. Examples include MySQL Group Replication [92], Galera Cluster [94] and Percona XtraDB Cluster [96]. They ensure strong degrees of consistency while sacrificing some performance due to synchronization overhead and the fact that all replicas must certify all transactions, limiting the aggregate available resources, given that work is repeated. MySQL Group Replication uses Mencius [14], a variation of Paxos, to order transaction certification, while Galera Cluster and Percona XtraDB use the Totem Single-ring Ordering protocol [9], a form of token passing synchronization.

Amazon's Aurora [116] also uses consensus to resolve write-write conflicts, however it detects them at run time [7]. An interesting property of Aurora is that it decouples the engine from the storage, allowing both components to scale independently. Aurora pushes the redo log to multiple storage nodes, while using an 4/6 quorum to ensure consistency. To improve performance, the database processing systems used (either PostgreSQL or MySQL) never flush the page blocks to the storage, and only write to the log, in order to avoid writing large amounts of data through the network. Periodically, a process in the storage nodes materializes the pages from the redo log, with the intent to avoid generating them by reading the entire log from the begin each time a page is needed. Because of this, the authors state that as far as the processing engine is concerned, the log is the database and the pages generated are merely a cache. Considering that the data is shared in a storage system, this allows to mount new processing nodes without having to worry about installing new persistent storage, which can be ideal for large databases. However, large databases probably won't fit entirely in memory, so every time there is a cache miss, the system must request the page from the storage, which depending on the network distance between the two components, can be an order of magnitude slower than reading directly from a mounted disk. This solution still relies on distributed consensus at storage level to ensure transactional consistency, which can harm the performance for distant storage nodes. In this architecture, to achieve global distribution, we can either have the clients distant from the processing/storage nodes, have the clients and processing nodes distant from the storage, or have the clients/processing close to a storage node but having the storage nodes distant from each other. Either way, performance penalties inherent from network latency are unavoidable. Aurora itself only allows for a maximum of two primary database nodes, with the additional requirement that they need to be in the same AWS region [8].

Separating the storage from the engine is also the basis of Hyder [19]. In Hyder, all servers process both reads and writes over the entire dataset (to avoid distributed transactions, *two-phase commit*, . . .), which is stored in the form of a log and accessed through the network. To reduce latencies, each server caches a partial copy of the database with the most recent data accessed. Missing data will have to be acquired from the shared log. Hyder employs a MVCC protocol, meaning that a transaction executes over a snapshot from when it started. In addition, it also executes them under OCC, relying on a certification step before commit. At commit time, the transaction's write-set is broadcasted to all servers, which append it to their log. It is also sent to the central shared log, which is responsible for providing the servers the order by which it must be certified (in the form of an offset), to guarantee that same outcome in all servers. After receiving both the write-set and the offset, a server executes the certification (in the intended order), updating its local cache in case it commits. By relying on central synchronization, Hyder avoids the performance penalties of distributed consensus protocols. A disadvantage of this architecture is the fact that cache misses might incur great response time delays, since data must be fetched through network, just like Aurora. Just like MySQL Group Replication and other architectures, all servers must process the certification step, limiting the total available resources as work is repeated (Hyder mitigates this by relying on an efficient tree-based certification process designated Meld, however certification overhead is unavoidable).

Apart from consensus, there is also the option to use a central transactional manager to handle certification. Although this solution contains a single point of failure, requiring solutions such as failover that causes the delay of some operations, it avoids the delays caused by the resolution time of distributed synchronization algorithms, especially visible in geo-distributed environments. Such example is the central certification algorithm that ensures PCSI [39], that employs a transactional manager which stores transactions timestamp and write-sets in the form of a log. When a transaction T is certified, in addition to sending the result to the replica that requested it, the transaction manager also sends the data of transactions committed after T started, so they can too be applied. There are several problems with this algorithm: First, since PCSI requires total order, a transaction can only be committed in a replica only after all previous committed transactions are applied, leading to increased response time. Second, how is duplicate data in the transaction response handled? For example, if transactions T_1 and T_2 start at the same time t in the same site and both commit, does the transaction manager send the data of transactions that committed after t twice, increasing packet size and therefore transmission time? (the paper does not seem to mention how this is handled); Third, by the algorithm's definition, in order to achieve total order, the transactions will have to be seemingly sequentially certified, leading to increased response times and scalability issues; Forth, by replicating transaction data only after a commit response, we can have a situation where a site presents a significantly old snapshot due to executing few update transactions; Finally, since the data in the transaction manager is kept in the form of a log, it can grow indefinitely. The authors solve this by having a garbage collector that periodically cleans old entries, according to minimum snapshot timestamp among all replicas (in order to still allow a site to recover from the transaction manager, after a crash). However, if a transaction starts in some snapshot at time t that later evolves to $t + \delta$ and causes the transaction manager to remove the entries previous to $t + \delta$, the transaction will not be able to be properly certified, since some entries are missing. To guarantee safety in this situation, the algorithm has to rely on aborting, even if the transaction was conflict-free. There is also a distributed certification algorithm to ensure PCSI, presented in the same paper, that relies on atomic broadcast.

In addition to systems that rely on replication for scalability, there are also systems that rely on sharding/partitioning. Examples include Citus [30] for PostgreSQL, MongoDB [83], Google's Cloud Spanner [27], Calvin [114], ConfluxDB [24], Granola [28], TAPIR [124], Carousel [121], Ocean Vista [40], Jessy [11], Blotter [86], among others. Sharding the data allows for the load to be distributed across a set of nodes. At the same time, depending on the type of transactions executed, network latency can be reduced, since this architecture allows for different subsets of data to be close to different users. However, when it comes to distributed transactions, there needs to be a mechanism in place to ensure multi-instance atomicity, such as *two-phase commit* [18], which introduces a negative impact caused by the network latency of multiple exchanged messages and the locking of records. Also, the partitions themselves also require replication in order to guarantee fault tolerance. Unlike systems that rely on replication for scalability, the replicas in sharding might not be able to independently handle reads or writes, depending on the consistency level adopted.

Citus and MongoDB use *two-phase commit* for multi-node transactions, having a separate node designated as coordinator. Citus also uses the coordinator to route the statements to the respective data nodes. Spanner, on the other hand, shards the data across sets of Paxos state machines, where it is replicated inside each one to provide high availability and to scale read-only transactions, while read-write ones have to be directed to the Paxos leader. Transactions execute under a conservative execution method implemented with *two-phase locking*, together with what the authors call a *lock-table*, in order to provide serializability. The reason behind the conservative model is due to the fact that the system was designed for long lived transactions, which have a lower starvation in this mode than in an optimistic one. Each Paxos group leader implements the *lock-table* that manages concurrent accesses. For transactions affecting multiple Paxos groups, *two-phase commit* is used, managed by the transaction managers implemented by all group leaders. Spanner uses synchronized wall clocks in order to present consistent snapshots throughout its shards (True Time). To guarantee that different clocks present the minimum drift, it uses expensive hardware, including atomic clocks and GPS antennas. As clock skews are unavoidable, transactions pick an upper bound (twice the uncertainty) as its timestamp that is guaranteed to be consistent across all shards, waiting to it after acquiring all the needed locks (on average, it has to wait around 10 ms). An interesting optimization done by Spanner is the buffering of writes in the client. Although it could allow for a better performance, a transaction cannot read the effects of its own writes, which can be problematic for some applications.

Calvin is formed by multiple clusters for high availability, each one composed by multiple nodes. Each cluster holds the entire copy of the data while each node in a cluster holds a partition. To execute transactions, a sequencer places the requests into a global input sequence, distributed across all replicas. Paxos is used to ensure coordination between clusters in a synchronous mode, while in an asynchronous mode a cluster is designated the primary. Then, transactions will have to acquire all locks for the data they will access, managed by a scheduler. By using deterministic locking to prevent conflicts, Calvin avoids the need to use the expensive *two-phase commit*. This comes with the drawback that a transaction must specify beforehand its entire read and write-set, which can be unfeasible for some workloads.

ConfluxDB, that ensures CGSI, shards the data among multiple primary nodes. However, these nodes do not directly process clients' requests. That task is instead left to secondary servers, that hold the entire copy of the data. The secondaries receive requests and process read-only transactions, while write transactions are redirected to the respective participant primaries. *Two-phase commit* is again used, to ensure atomic commitment among primaries. After commit, data is asynchronously replicated from the primaries to a node that executes log merging, proving data to the secondaries. This architecture thus allows scaling the system based on the type of workload. If the workload is read-heavy, more secondaries can be added. If the workload is write-heavy, more primaries can be added. However, it is still subjected to the performance penalties of distributed transactions and increased abort rates due to ensuring a stricter consistency level with partitioned snapshots (e.g. total ordered transactions).

Granola, a transaction coordination infrastructure applied to partitioned data, avoids *two-phase commit* by relying on a form of consensus for its *independent distributed transactions*, i.e. transactions that execute on different partitions but always reach the same commit decision. Consensus is used to determine the total order by which transactions are executed. In detail, each participant assigns a timestamp to a transaction, based on several counters. Then, each participant broadcasts the timestamp to the other ones, in the form of a vote. Finally, after receiving all votes, a participant chooses the maximum value to be the transaction's actual timestamp, which will indicate its position in a globally serialized queue. Since each participant will agree on the same timestamp, the order is consistent throughout different partitions. The main disadvantages of Granola are the fact that transaction processing is mostly sequential, as its execution must respect the queue order, and the fact that it only supports single-round transactions, i.e. there is no interaction between the client and the server at runtime. Furthermore, as total order must be kept, a participant's crash can cause the stall of the entire system, since a transaction can be stuck at voting phase and other transactions with higher timestamp will have to wait. Granola also employs *coordinated distributed transactions*, where the decision might vary from node to node due to applicational logic. To achieve this, it relies on preemptive locking of all resources needed and the exchange of votes related to commit/conflict/abort decisions among participants (the conflict vote is based conflict inherent from acquiring the needed locks, while abort relates to applicational logic).

The TAPIR keystore also employs sharding to scale performance and replication to ensure fault tolerance. In contrast to relying on a classic algorithm such as Paxos to implement replication, TAPIR relies on a method designated *Inconsistent Replication*, allowing transactions to be executed in a different order by different replicas but in the end returning the same result to the application (in its *consensus* mode). Briefly, a client sends the transaction request to the replicas, which in turn execute it and reply with the result. In the best-case scenario, if the client (which acts as the coordinator) receives a $\frac{3}{2}f + 1$ replies with the same result (where f is the maximum number of tolerated failures), returns to the application protocol and asynchronously sends the final result to replicas. If the client doesn't receive the required majority, it has to decide which will be the actual result and synchronously reply it to the replicas, awaiting new confirmation to guarantee consensus. This, in turn, leads to a resolution time of one round for the best-case scenario, just like Fast Paxos [65]. The fact that some replicas might have missed transactions or executed them out of order means that they need to be periodically synchronized, using a merge process. Because of this, replication can only be used to implement fault tolerance and not scale reads/writes in strong consistency systems. Furthermore, since replica commits are asynchronous, a client might not be able to read/update its immediate writes.

The Carousel keystore, similar to TAPIR, that partitions data and replicates it in a consensus group using Raft, aims to reduce transaction latency in sharded and replicated systems by running its execution and certification (*two-phase commit* and consensus) in parallel. To do this, in its optimized mode (based on Fast Paxos [65]) the client sends a *prepare* message with the read and write-set keys to each replica in each participant consensus group (the coordinator also receives a similar message, replicating to its consensus group). Each replica then independently certifies against the other pending (i.e. concurrent) transactions, setting it as pending if it passes and returning the response to the coordinator (a participant leader local to the client). If the coordinator receives a super-majority of equal responses related to some partition (in the condition that the partition leader must belong to the majority and all majority members must be up-to-date), it can safely consider it as the partition response (fast path). If not, it waits for the certification, replication and reply by the partition leader (slow path). Both slow and fast paths are simultaneously executed to reduce latency and the coordinator ignores the slow path messages when the fast path succeeds. Meanwhile, the client is receiving the data it requested to read. Having received all reads, the client prepares the writes and sends them to the coordinator, which replicates it to its consensus group. After knowing the responses for all partitions, the coordinator returns the result to the client. Then, it asynchronously tells all the replicas in all partitions to commit or abort. Serializability is still ensured since replicas perform read-set certification. While this allows to reduce, for the most part, latencies inherent from sequentially executing different phases, it suffers from a few drawbacks. Namely, a client must know and provide beforehand its entire read and write-set keys (if the write-sets are unknown, the paper proposes a workaround with a *reconnaissance transaction*, i.e. first determine the write-set and then execute the actual transaction), and read-write transactions are limited to two rounds: first all reads and then all writes. Additionally, *two-phase commit* effectively locks the records in the write-set keys. Because of this, client stalls or a slow/distant partitions can limit the performance of several other transactions in the system. Given that the commit at the partitions is asynchronously executed, immediate reads and writes by the same client to the same data can be met with transactional aborts. Furthermore, executing the fast and slow paths concurrently generates repeated work, as each replica must both certify and receive the replicated data from the leader. Moreover, since not all partitions are guaranteed to be replicated in all zones, remote reads might be necessary.

The Ocean Vista architecture is also similar to TAPIR and Carousel, aiming to provide low latency to geo-distributed serializable transactions. It is comprised by multiple clusters, each with multiple data partitions and a gossip server (replicated for fault tolerance) that is used to exchange timestamps (or *watermarks*). Contrary to both TAPIR and Carousel, it does not rely on the locking of records inherent from implementations based on *two-phase commit*. Instead, it uses multiple versions per object, allowing the concurrent execution of even conflicting transactions, on what the authors call *Asynchronous Concurrency Control*. Read-write transactions are provided in the form of stored procedures. In detail, the client sends the transaction data to any partition, which becomes its coordinator. The coordinator assigns it a unique timestamp based on its id, a monotonically incremented clock and a physical clock (synchronized using NTP). Then, the S-Phase is executed: the transaction's data is replicated to the other partitions, which then store the writes as placeholders, whose real value might depend on the values read in other partitions; the confirmations are then returned to the coordinator (this execution is based on Fast Paxos [65]). After that, a transaction needs to wait until its timestamp is lower than *VWatermark* (value which guarantees that all transactions with a lower timestamp concurrently being processed have completed the S-Phase). Then, the E-Phase is executed: the replicas that write request the needed read values from other replicas, if necessary; after collecting all values, the transaction logic is executed, i.e. the placeholder values are replaced, and the result is returned to the coordinator; the coordinator can return after receiving the first response, since the system assumes deterministic execution. Finally, the values are asynchronous replicated to all replicas belonging to the write-set. Some problems with this approach include the fact that transactions are limited in their logic (all reads then writes; deterministic) and require the entire read and write sets to be known (just like Carousel, it is suggested a workaround with an extra *reconnaissance transaction*). In addition, the read operation might need to be processed by multiple, possibly distant replicas, if the *replica watermark* (which guarantees that transactions with a lower timestamp are fully replicated in all replicas) is lower than the transaction's timestamp, increasing response times and reducing scalability as work is repeated. To ensure serializability, transaction processing also requires a delay until the *VWatermark* reaches the required value, possibly leading to higher response times even for non-conflicting transactions. This can be further aggregated since the timestamps rely on a physical clock, meaning that a transaction might need to wait even more if its coordinator clock is ahead.

Jessy, that ensures NMSI [11], shards objects across multiple processes. Unlike similar solutions, Jessy relies on partial replication, meaning different processes hold different subsets of data and an object is held by multiple processes, in order to achieve fault tolerance and balance loads. To start a transaction, a client contacts any process in the system, which becomes the transaction's coordinator. Writes are buffered in the coordinator, while reads are either executed locally, if the coordinator has the object stored or the transaction being executed has overwritten it, or remotely if not. When the client wants to commit a transaction, the coordinator atomically multicasts its write-set to every process that stored a copy of at least one of the modified objects. When a process receives a commit request, it certifies the data against previously committed transactions, multicasting the vote to all participant processes and the coordinator. The votes must be exchanged since a process might not hold the transaction's entire write-set, meaning it might not be able to completely certify it on its own. Each process can commit after it received enough votes to account for every object in the write-set.

This means consensus can be effectively avoided if a transaction only updates data stored in a single process. The main disadvantages are the fact that a transaction might need to rely on remote reads and its certification might require votes from multiple participants and the exchange of multiple messages that have to be atomically multicasted. These problems are further aggravated in the context of geo-distribution, given that network latency is higher.

Blotter, like Jessy, also ensures NMSI [11] and also has the writes buffered, this time by the client, not appearing to support transactions reading their temporary writes. On commit, the transaction manager executes *two-phase commit* over the participant data managers. Blotter provides cluster geo-replication by relying on Paxos to implement a replicated state machine (full-replication, unlike Jessy). Apart from reads, which are local, every operation needs to be executed by every cluster, thus replication serves to reduce read latency and provide high availability but not to improve write scalability. Blotter improves state machine replication throughput by allowing conflict-free transactions to execute in parallel, effectively implementing a state machine per object instead of one per database.

A hybrid solution between replicas and partitions can be found in the Walter keystore [109]. Walter uses the concept of *preferred sites* to execute transactions, where data is replicated across many nodes but only one can modify some record. This way, all transactions affecting one particular record all go to the same replica, avoiding the need for distributed coordination. Walter justifies this technique on the assumption that most transactions will only affect a local subset of data (such as users' interaction in a social network). Despite that, there is still the downsides of having a sharded system, such as forced coordination in transactions that affect multiple replicas (Walter uses *two-phase commit* to achieve this). To ensure PSI, Walter uses vector clocks that represent the number of transactions applied for each node. These vectors are also assigned to a transaction when it begins and to each object in the database. Applying transactions thus needs to follow the order indicated by those vectors.

The concept of preferred sites can also be found in the SLOG system [101], where all writes and consistent reads to an object must processed by its home region. SLOG supports two types of transactions: *single-home* transactions, that affect a single region; and *multi-home* transactions, that affect multiple regions. To ensure isolation, multi-home transactions must be redirected to the same region to be totally ordered. In addition, SLOG requires that both types of transactions must lock all records used before the transaction starts, as well as relying on a deterministic execution to avoid *two-phase commit*. This means that the entire transaction must be provided to the system, limiting client-server interaction.

There solutions whose purpose is to provide transactional guarantees in the form of a middleware over a storage engine that does not support them (e.g. Cassandra and

HBase). Examples include Apache Omid [42], pH1 [26] and CloudTPS [125]. Since they are layered on top of existing systems, they cannot avoid some performance penalties arising. Omid executes transactions over Snapshot Isolation, requiring an underlying store that supports multi-versioned data, such as HBase, where it also stores metadata. It uses a centralized transaction manager in order to ensure transactional consistency. pH1 uses a distributed cache to store multi-versioned data, so transactions execute under the Snapshot Isolation without the need for any changes to the underlying database engine, which only stores the most recent data, unlike Omid. pH1 runs transactions under a optimistic model, making use of a transaction manager to certify them at commit time. Since both Omid and pH1 use multi-versioned data, they enable non-blocking reads. CloudTPS, like pH1, is also not limited to a store that supports multi-versioned data. It uses multiple transaction managers, each one being responsible for handling a subset of data, in order to reduce the bottleneck of a single central one. Because of this, it has to rely on *two-phase commit* in order to execute transactions, since the data they depend on can span multiple managers.

Since providing strong consistency guarantees in distributed database systems negatively impacts both performance and availability [21], operations can be executed by different nodes without any form of synchronization, other than convergent post-commit replica merges. In this architecture, conflicts are often dealt with using vector clocks and rules such as *last-writer-wins* [113] and/or rules defined by the application developers. Systems that run under weak consistency guarantees (e.g. eventual consistency [118], causal+ [71]) include Amazon's DynamoDB [34], Postgre-BDR [1], Bucardo [22], SQL Server Merge Replication [79], Apache CouchDB [10], Microsoft's CosmosDB [78], Riak [103], Cassandra [63], AntidoteSQL [73], COPS [71], Eiger [72], Cure [6] and others. This provides full availability, since no node depends on any other to write and commit data, at the cost of a weaker consistency.

To prepare against a large variety of use cases, some systems offer the option to choose between strong and weak consistency. For example, AntidoteSQL [73], CosmosDB [78] and Riak [102]. AntidoteSQL uses a distributed multi-level locking to prevent conflicts, while both CosmosDB and Riak rely on consensus.

Finally, a way to obtain stronger consistency in multi-writer databases without coordination is to model workloads in a way that avoids conflicts altogether. By the nature of some use cases, the data of different replicas could be easily merged if the writes were made to disjoint data. Since this is not always possible to ensure, some research has been made in creating data types, such as counters and sets, that allow for concurrent writes in the same object to be merged without conflict, as seen in Section 2.4. Examples of databases using these data types include Riak [104], Redis [20], CosmosDB [107], AntidoteSQL [73], Walter [109], Cure [6], among others.

2.6 SUMMARY

Database systems design has to weigh between strong consistency and high performance. Centralized databases can offer strong consistency guarantees using efficient concurrency control protocols; however, they lack the scalability to process a relatively high number of, possibly geographically distant, clients' requests. Distributed databases that ensure strong consistency rely on expensive synchronization control protocols, that often require the exchange of multiple messages, the synchronization of multiple sites or repeated work, which can hinder performance and limit scalability of geo-distributed databases. On the other hand, distributed databases that favor performance over consistency are limited in the functions they provide, which in turn increases applicational completely if stronger consistency is desired.

There is still room for improvement, namely in the research of different concurrency control/isolation protocols and different ways to implement them. This could further improve transactional performance on distributed databases with strong consistency guarantees, especially ones that are aimed at geo-distribution.

3

THE PRIMARY SEMI-PRIMARY ARCHITECTURE

3.1 MOTIVATION

As an application's load increases, so does the need to scale the data layer. Since a server can't be indefinitely vertically scaled, relying on multiple ones becomes a necessity to process high loads, as different sites can process different requests. In addition, multi-instance databases can also decrease the response time for clients in different geographical locations, by reducing the client-server distance. However, multi-instance architectures must deal with the fact that providing strong consistency guarantees to distributed databases is more complex than for centralized ones [17].

Some systems opt to provide weak consistency guarantees in order to obtain higher performance and availability. This, however, makes them unfeasible for some use cases such as financial or retail, and attempts to use them in such areas demands introduced applicational complexity. Other systems provide stronger consistency guarantees by relying on *two-phase commit*, distributed consensus, distributed locking, among others [17]. Such protocols have visible performance penalties, as they can require the exchange of multiple messages and/or the need for transactions to be processed by multiple or all nodes in the system.

The main motivation is therefore the design of a database architecture that can provide both high performance/availability and strong consistency, by reducing the impact caused by latencies of distributed synchronization protocols. The next section proposes an architecture designed to achieve this.

3.2 OVERVIEW

The *Primary Semi-Primary* architecture is a distributed database system designed to provide strong transactional guarantees (in a form of Snapshot Isolation), in both local and geo-distributed environments, without incurring high performance penalties. It comprises two components: the *primary* (single node), that handles transaction certification and data replication but not client operations, and the *semi-primary* (multiple nodes), that processes client's reads and writes. The *Primary Semi-Primary* operates under full-replication as a means to ensure low latencies to clients in different geo-locations and to avoid the need for expensive distributed transactions. Because of this, the *primary* and *semi-primary* have an entire copy of the database. It can be seen as a middle ground between the multi-primary architecture, where all nodes can commit/certify transactions, and the primary-standby one, where just one of the nodes is responsible for writes/commit. Just like systems that don't offer transactional guarantees, transactions executed on one *semi-primary* are eventually replicated to the others. Even though data is eventually replicated, the fact that there is a central certification still allows to ensure strong consistency guarantees, since the *primary* certifies a transaction against all previously committed data.

3.3 ARCHITECTURE

3.3.1 *Overview*

The *Primary Semi-Primary* architecture is composed by a single *primary* replica and multiple *semi-primary* replicas. The *primary* is a central component, responsible for transactional isolation and recovery semantics. It does not handle client's query execution. Each *semi-primary* provides read and write access to clients, thus being directly responsible for query execution. They relay transactional write-sets to the *primary* for certification and persistency.

Figure 1 outlines the interactions with and within the *Primary Semi-Primary* architecture as follows: ① To start a transaction, a client performs a *begin* operation against the chosen *semi-primary*, that assigns it a unique identifier and a snapshot for reading. This operation does not have to wait and the snapshot contains all previous local writes. ② Reads are done from a data snapshot based on the assigned timestamp, using an MVCC method like Snapshot Isolation [16] and writes are stored in the memory of the respective *semi-primary*, indexed by the transaction's id. ③ When the client issues a commit request, the *semi-primary* processes it and ④ sends the write-set to the *primary*[1], which in turn ⑤ certifies it against all committed data. If the global certification succeeds, the *primary commits* the changes, storing them into its database. ⑥ The commit response is then sent to the *semi-primary*, which in turn *applies* the transaction in case of commit ⑦, by persisting its data and updating the timestamps of the now obsoleted data. This operation does not have to wait for pending updates from remote transactions. Finally, ⑧ the *semi-primary* replies to the client. In case of success, ⑨ the transaction changes are asynchronously propagated to the other *semi-primaries*.

1 Read only transactions don't require certification.

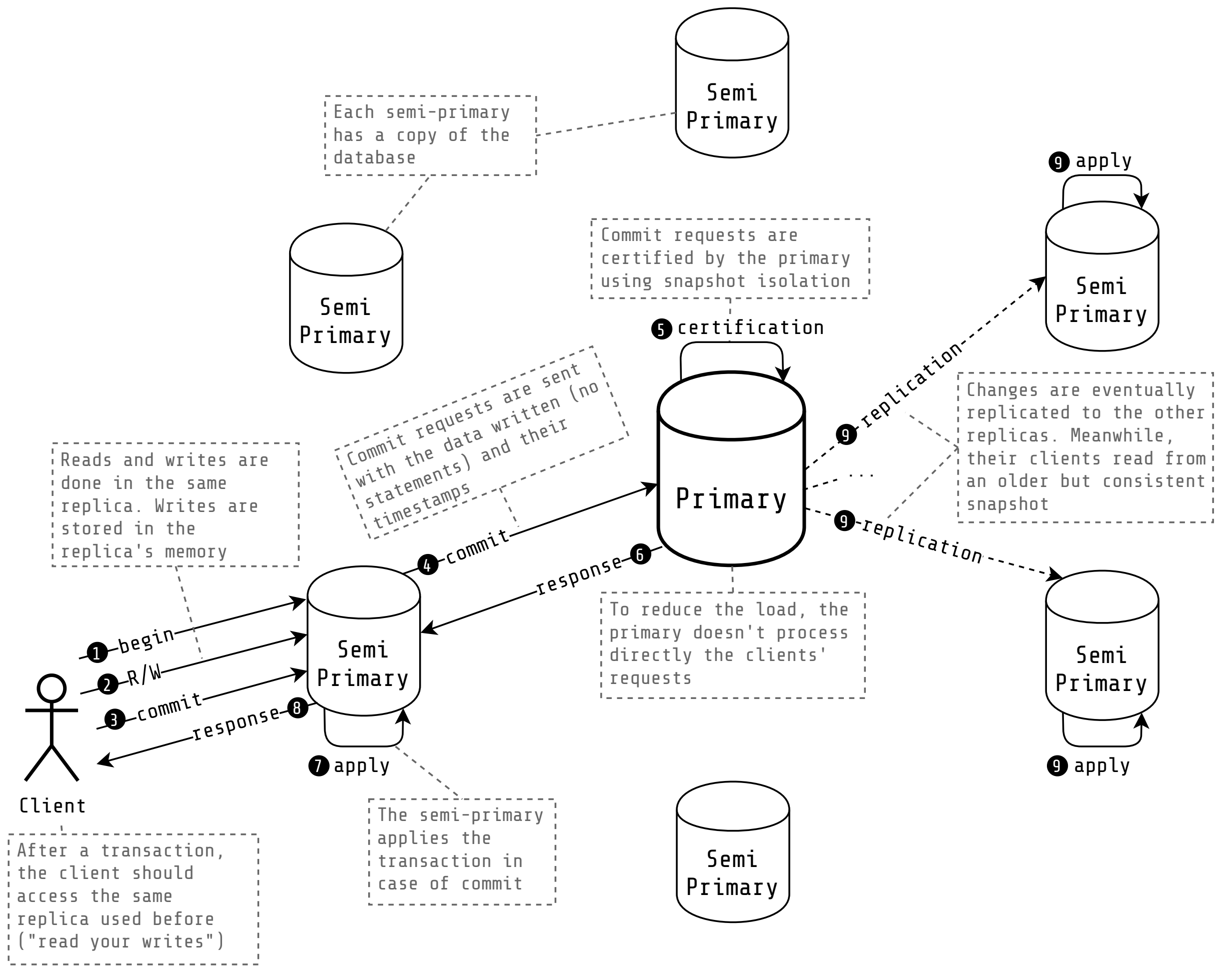

Each semi-primary has a copy of the database
Semi Primary
Semi Primary
Commit requests are certified by the primary using snapshot isolation
5 certification
9 apply
Semi Primary
Commit requests are sent with the data written (no statements) and their timestamps
Primary
9 replication
Changes are eventually replicated to the other replicas. Meanwhile, their clients read from an older but consistent snapshot
9 replication
Reads and writes are done in the same replica. Writes are stored in the replica's memory
4 commit
response 6
To reduce the load, the primary doesn't process directly the clients' requests
1 begin
2 R/W
3 commit
response 8
Semi Primary
Semi Primary
7 apply
9 apply
Client
The semi-primary applies the transaction in case of commit
After a transaction, the client should access the same replica used before ("read your writes")
Semi Primary

Figure 1: *Primary Semi-Primary*'s architecture overview.

Although Figure 1 represents the *semi-primaries* as single instances, the storage itself can rely on multiple ones. The architecture only predicts that each *semi-primary* holds the entire copy of the data, but not how they do it. In case of sharding, it would still not require *two-phase commit* since the storage nodes don't need to vote, just write to disk.

The remainder of this section focuses on how the system as a whole enforces transactional isolation, a high degree of parallelism, replicates data and recovers from faults while ensuring that a client is never blocked while starting or after committing a transaction and can still read all previous local writes.

This architecture is designed to work with either SQL, NoSQL or even a combination of both. For example, the *semi-primary* could use a SQL database for their clients to interact while the *primary* could use a NoSQL database for the write-set storage. The remaining issue is how the components of the *Primary Semi-Primary* architecture map to actual SQL and NoSQL database engines, thus allowing the architecture to be realized with existing software packages. This is addressed in Section 3.4.

3.3.2 *Isolation and Consistency*

Ensuring Snapshot Isolation, in which each transaction is provided with the most recent snapshot globally, would mean that a client might need to wait (either on begin or on commit) for all preceding remote transactions to be propagated and locally applied to read their most recent writes. On the other hand, if transactions start with an outdated snapshot to avoid waiting [39], it would lead to missing local writes by previous transactions and an increase in the number of aborts, as more transactions are concurrent.

However, it is expected that clients that access one *semi-primary* mostly update a different subset of data than other clients that access other *semi-primaries*. This assumption is exploited to allow for local writes to be made immediately available to new transactions such that a client does not need to wait. This makes snapshots in *semi-primaries* evolve independently from each other while ensuring that all conflicting updates are seen in the same order by all clients, achieving PSI [109], which extends Snapshot Isolation to allow different sites to apply non-conflicting, non-causally dependent transactions in different orders.

To achieve this, in addition to a global timestamp incremented sequentially by the *primary* when each transaction is committed ($global\_t$), there is for each *semi-primary* a local timestamp ($local\_t$) that is incremented when a transaction is applied locally. A client transaction that initiates on a *semi-primary* is assigned a snapshot at the start based on $local\_t$ at that *semi-primary* ($begin\_t$).

All participants, both the *primary* and the *semi-primaries*, store for each item the $global\_t$ of the last transaction that modified it. Each *semi-primary* also stores, for each record, two local timestamps that control its validity: from when it can be read ($from\_t$), to when it became obsolete (to_t, which can be infinite). Clients can also read their own temporary writes, despite not being yet committed and not having been assigned any timestamps. Those temporary writes will have to, when selected, overwrite persistent records based on their key, as otherwise they would violate primary key constraints.

For deletions, a flag will be placed next to every one stating if it was deleted or not (also known as a "tombstone"). Clients will then only be able to read values that succeed both time and deleted restrictions. Given a transaction T with begin time $begin(T)$, that modified records Temp_T and started in a *semi-primary* with P data, the functions $from(x)$ and $to(x)$, that return the $from\_t$ and to_t values of the record x, respectively, $key(x)$, that returns the primary key/id of the record x,

$keys(X) = \{key(x) \mid x \in X\}$ and $del(x)$ as the boolean function that indicates if a record x was deleted, values that can be read by a transaction are defined as follows:

$$\begin{aligned} S_T = \quad & \{x \mid x \in \text{Temp}_T \;\wedge\; \neg del(x)\} \\ \cup \;& \{x \mid x \in \text{P} \;\wedge \neg del(x) \;\wedge from(x) \leq begin(T) \leq to(x) \\ & \quad \wedge\; key(x) \notin keys(\text{Temp}_T)\} \end{aligned} \tag{1}$$

When a client requests that a transaction is committed, the write-set composed by the modified items and their global timestamps that are stored locally are sent to the *primary*. To ensure consistency, the *primary* will only commit it if no previously committed transaction has also updated any of the same keys concurrently. If a transaction T modifies a record r that has a *global_t* of g in its respective *semi-primary*, the *primary* will only commit T if the most recent *global_t* of r is equal to g, i.e. T started in a snapshot with the most recent version of r. If the *global_t* of r stored in the *primary* is bigger than g, this means that r was already modified by a concurrent transaction, implying that T started before the most recent data of r reached the *semi-primary* it started on. Algorithm 1 formally presents the *primary*'s certification. Note that this can be efficiently implemented using indexes and relational joins (Section 3.4).

Algorithm 1: *Primary*'s global certification definition

```
Function $certify(T)$:
    foreach record $r \in T.write\_set$ do
        $curr \leftarrow get(r.pk)$
        if $curr \neq NULL \wedge curr.global\_t > r.global\_t$ then
            $abort(T)$
            return $False$
        end
    end
    $commit(T)$
    return $True$
```

When some transaction T is certified by the *primary*, it gets assigned a unique *global_t*. When T is applied by each *semi-primary*, for each record in its write-set, the *semi-primary* sets the $from\_t$ to its own current $local\_t + 1$, the *to_t* to infinity and the *global_t* to T's *global_t*. Records deprecated by the new modifications in T have their *to_t* set to the current *local_t*. Finally, the *semi-primary* increments its current *local_t*, making transactions that start after T finished be able to read T's modifications. Algorithm 2 formally specifies the *semi-primary*'s *apply* operation.

Note that neither Algorithms 1 nor 2 account for conflicts arising from concurrent executions. That will be handled later in Section 3.3.3.

Algorithm 2: *Semi-primary*'s apply operation definition

```
Function $apply(T)$:
    foreach $record\ r \in T.write\_set$ do
        $old \leftarrow get(r.pk)$
        if $old \neq NULL$ then
            $old.to\_t \leftarrow current\_local\_t$
        end
        $r.from\_t \leftarrow current\_local\_t + 1$
        $r.to\_t \leftarrow \infty$
        $r.global\_t \leftarrow T.global\_t$
        $add(r)$
    end
    $current\_local\_t \leftarrow current\_local\_t + 1$
```

It is also ensured that if a transaction T_2 started on a snapshot modified by T_1, T_2 is applied after T_1 in all *semi-primaries* (causal order is kept, more detail in Section 3.3.3). If two transactions are applied by different *semi-primaries* in different orders, this means that at least one of them is local and that they were not related to each other, and whichever order they are applied, the final result is always the same. The only exception to this are *write-skew* anomalies[2], also not prevented by the Snapshot Isolation [16].

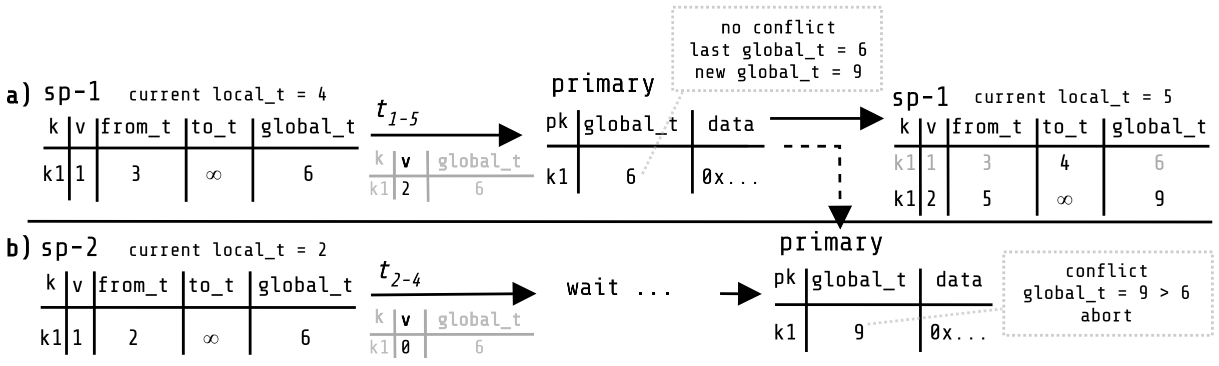


Figure 2: Example of the *Primary Semi-Primary* isolation.
It presents two concurrent transactions being executed at the same time on different *semi-primaries*. a) displays a successful transaction while b) displays one that aborts.

Figure 2(a) shows an example of how timestamps are stored and transactions certified. In detail, *semi-primary sp*-1 initially has *local_t* of 4, meaning that it has applied 4 transactions, and $k1$ set to 1. This value was set by a transaction with global timestamp 6 that was applied locally as timestamp 3. *Semi-primary sp*-1 then executes a transaction t_{1-5} that sets $k1$ to 2 and tries to commit. This sends $k1$ with its locally known *global_t* of 6 to the *primary*. Assuming that no other change has been made

2 The *primary semi-primary* architecture could be extended to prevent *write-skews*, by supporting read-set certification. However, since it is expected that the majority of applications don't need this requirement, the performance penalty and implementation complexity would not be worth it.

to $k1$, the *primary* will compare the timestamp for each key with the one stored there and determine that there was no conflict, issuing it a new global timestamp of 9. Upon reception the acknowledgement of commit with timestamp 9, the *semi-primary* will immediately apply the transaction issuing it local timestamp 5. Note that the difference between $local\_t = 5$ and $global\_t = 9$ at this time means that there are 4 remote transactions that have already been committed but not yet locally applied. At this time we know that none of those transactions conflicts with t_{1-5} – otherwise it would have aborted.

Figure 2(b) shows an example of a transaction that cannot be committed. In detail, *semi-primary* sp-2 executes transaction t_{2-4} whose snapshot also contains $k1$ with $global\_t = 6$ and tries to assign it a new value 0. When the primary gets its write-set it has to wait until its turn to be certified, in this case, until t_{1-5} has been decided (more details in Section 3.3.3). Then, as the $global\_t$ for $k1$ is still 6, lower that the current 9, a conflict is discovered, leading to t_{2-4} to be aborted.

Finally, since we rely on a MVCC method, there must exist a mechanism in place to remove obsolete versions of records. In addition, temporary structures that hold the transactions' write-sets need to be cleared. Having an asynchronous operation to periodically do this instead of doing it every time before returning to the client ensures a lower response time.

3.3.3 *Parallelism*

The *Primary Semi-Primary* architecture exploits parallelism as much as possible to improve throughput and mask communication latencies, in particular, in a geo-distributed setting.

Semi-Primary

The key opportunity to exploit parallelism in *semi-primaries* is to allow snapshots to advance independently, such that clients don't need to wait before starting new transactions. In addition, since we rely on an MVCC method, clients' reads can be executed concurrently, without needing to acquire any locks. Finally, as clients' writes are redirected, together with their transactions' ids, to temporary data structures, we can think of all inserts/updates/deletes as insert operations that do not physically interfere. Because of this, writes themselves are also completely independent from each other. These three points make clients' transactions able to execute without conflicting with each other, guaranteeing maximum parallelism.

As for *applying* transactions, we have two different cases:

1. local transactions – transactions that were executed in the same *semi-primary* that applies them;
2. replications – transactions that were executed on another *semi-primary*.

Local transactions applied at the same time never conflict with each other. This is because if they did conflict, one them would have been aborted by the *primary* based on the *global_t* of the conflicting record(s). Because of this, local transactions don't need to wait for other local transactions to be applied. When it comes to updates propagated from other replicas, however, we have to control the order they are applied, to guarantee that if some transaction T_2 modifies data written by transaction T_1, then T_2 is applied after T_1 in all *semi-primaries*. This can be done either by having dependency information in each transaction so they can be applied by their causal order (similar to CSI [95]) or by applying replications sequentially by the order of their *global_t* (similar to PSI; the latter also needs to wait for local transactions with smaller *global_t* to be applied first).

Although transactions can be applied in parallel, special consideration must be had when evolving the snapshot, i.e. assigning and updating timestamps. If the underlying *semi-primary* database supports Snapshot Isolation, this task is the responsibility of the engine. If not, a possibility is to leave that assignment/update to the end of the *apply* operation and execute it under an exclusive lock.

Primary

As there is only one *primary* handling the global certification of all the transactions in the system, it has to be designed with special care in order to explore parallelism. Transaction certification can't be sequential since it would gravely impact performance under heavy load. At the same time, it also can't be completely parallel, because different transactions could update the same data, resulting in undefined behavior and possibly breaking consistency. With that said, transactions that update disjoint data (parallel transactions) can be certified at the same time. Two transactions are parallel (||) if the intersection of their keysets, i.e. set of records' primary keys/ids, is empty:

$$T_1 \mid\mid T_2 \iff keys(T_1) \cap keys(T_2) = \emptyset \tag{2}$$

The concurrency problem is addressed by exploiting an existing database engine to implement the *primary*, as database engines are already optimized for this case. In fact, in the *Primary Semi-Primary* architecture the *primary* does not handle query execution load and can be fully dedicated to this. Briefly, if the *primary*'s underlying engine supports Snapshot Isolation, we can simply send all transactions to the database and not worry about conflicts, as the database will handle the concurrency control itself.

As an alternative, for instance, if the primary is implemented with a NoSQL database engine that does not provide the desired isolation, the following mechanism implemented in middleware is used instead. First, to improve performance, transaction certification is split into two parts:

1. *Scheduler*: pre-certification (sequential) – checks if a transaction can be certified or has to wait, based on the set of all keys currently being certified/committed (K) :

   $$pre\_cert(T) \iff keys(T) \cap K = \emptyset \iff (\forall\, T' \in C)(T \,||\, T') \tag{3}$$

   , where C is the set of all transactions currently being certified/committed.

2. *Executor*: certification (parallel) – certifies the transaction against already committed data and commits/aborts it.

This improves performance as transaction certification is slower than pre-certification, since it compares against a larger set of data and needs to commit modifications to disk. However, pre-certification itself can be parallelized if we know beforehand that two transactions modify disjoint data. To achieve this, transactions are preemptively classified based on the tables they modify or on some arbitrary partition key defined by the application's developers (e.g. geographical region). Transactions with different classes are then processed by different schedulers. To protect against the possibility of transactions spanning multiple classes be certified at the same time, the schedulers must acquire a lock of that class. Transactions with the same class are always assigned to the same scheduler, in order to reduce lock contention (transactions with multiple classes are assigned to the worker responsible for the smallest one, based on its hash). Locks must always be acquired by the same order to avoid deadlocks. Since we don't want to restrict the data model by having static classes defined, class assignment is dynamic, meaning that when a transaction with a new class arrives to the *primary*, it is assigned to some scheduler. We can now update the certification stages above with an extra one:

1. *Classifier*: classification (parallel) - classifies a transaction based on the data it modifies; can be implemented in the *semi-primary* or the *primary*;

2. *Scheduler*: pre-certification (sequential in the same class, parallel between classes);

3. *Executor*: certification (parallel).

3.3.4 *Replication*

Who sends the data

Data replication is handled directly by the *primary*, since it is already connected to all the *semi-primaries* in system. If replication was peer-to-peer, i.e. *semi-primary* to *semi-primary*, we could have all *semi-primaries* knowing each other or knowing a small subset, making sure that there is a path between every *semi-primary*. Either way, *semi-primary* connection or disconnection would become more complex. Also, although *primary* load would be reduced, it would be expected that replication would take far longer to occur, since source and target *semi-primaries* could be distant from each other or the message could take multiple hops to reach the destination. Finally, a *semi-primary* would require the same availability guarantees as the *primary*, since we need to make sure that a transaction's data reaches all *semi-primaries*, increasing implementation complexity.

How the data is sent

As for how data is packaged and sent, we have two different choices:

1. Send a transaction's data as soon as it finishes. Advantages:
   - reduced replication delay, possibly reducing aborts;
   - can reuse the transactions certification's payload directly as the replication message;
   - easier implementation ("just send").
2. Batch multiple transactions' data in a single replication message or send what's available after a timeout. Advantages:
   - smaller TCP/IP overhead;
   - lighter network load;

The disadvantages for each option is the other's advantages. Unlike the decision of who sends the data, either choice about how the data is sent is feasible, and its implementation will be dependent on the system's requirements (smaller delay *vs* lighter network load).

3.3.5 *Recovery*

Since both *primary* and *semi-primaries* can fail, recovery is a needed functionality the system must provide, being a crucial one in order to maintain consistency.

In recovery, data is also replicated from the *primary* to the *semi-primaries*, as the former is the only component guaranteed to have all the data persisted. Every time a *semi-primary* connects or reconnects to the *primary*, it issues a recovery request. To decrease recovery time, the *primary* only needs to send the data the *semi-primary* might have missed. To do this, a *semi-primary* sp with sp_T transactions applied sends the maximum *global_t* ($recovery\_t_{sp}$) which guarantees that all transactions with a smaller one are applied in sp. Briefly, assuming that $globals(X) = \{global(x) \mid x \in X\}$ and $global(T)$ returns the *global_t* of a transaction T:

$$\begin{aligned} recovery\_t_{sp} = max(\{x \mid x \in \{1, ..., max(globals(sp_T))\} \\ \wedge \{1, ..., x\} \subseteq globals(sp_T)\}) + 1 \end{aligned} \tag{4}$$

With a practical example, considering that $globals(sp_T) = \{1, 2, 3, 5\}$. The set of elements that would pass the condition $\{1, ..., x\} \subseteq globals(sp_T)$ would be $\{1, 2, 3\}$. 5 would not pass since $\{1, 2, 3, 4, 5\} \nsubseteq \{1, 2, 3, 5\}$. Therefore, $recovery\_t_{sp}$ would be $max(\{1, 2, 3\}) + 1 = 4$. The *primary* then only needs to send the records whose *global_t* ≥ 4, minimizing data sent. The *semi-primary* then ignores the recovery values that were already applied by it (in the above example it would be the data with *global_t* $= 5$).

The recovery process must also handle transactions that were running at the time of crash. To ensure that the client, with or without *primary*/*semi-primary* failure, always receives the correct status of a transaction, running transactions can't be simply considered aborted because they could have been committed by the *primary* without being applied by the *semi-primary*. For the recovery to work, each transaction will have a status assigned to it, stored in its *semi-primary*'s log. The list of possible statuses is presented below:

- *running* (`R`) – transaction is being executed by the client;
- *waiting* (`W`) – transaction certification was received and processed by the *semi-primary*; global certification may or may not been issued to the *primary*;
- *committed* (`C`) – transaction committed and applied in the *semi-primary*;
- *aborted* (`A`) – transaction aborted;
- *pending* (`P`) – special status that implies that the transaction is waiting resolution from the recovery.

When a *semi-primary* issues a recovery request after it crashes, it sets all *running* transactions as *aborted*, since it means that temporary data was lost, as it is kept in memory. It also sets all transactions in *waiting* state to *pending*, knowing its final state after processing the recovery message from the *primary*.

In addition to the data, the *primary* also has to send a log that displays the result for every transaction. To minimize the message's size, *recovery_t* is again used to only send the missing transactions.

When the *semi-primary* receives the recovery message, it applies the new data (akin to a normal replication) and sets the *pending* transactions to their real result, based on the log received. If some transaction was *pending* but wasn't present in the recovery log, this means it was lost[3]. For these transactions, the *semi-primary* can safely set their status to *aborted*.

Since a *semi-primary* has no knowledge of the clients that were waiting for a response, the clients themselves have to retry the commit request if they sense that the system crashed (e.g. timeout). When a *semi-primary* receives a repeated recovery request, it does the following: if the status is *committed* or *aborted*, returns the status; if the status is *waiting* (can happen if the *primary* crashed), retries the global certification request; finally, if the status is *pending*, it does nothing, since it is waiting for the recovery reply.

If the *primary* crashed, it only has to send a *welcome* message to the *semi-primaries* when they reconnect. When a *semi-primary* receives a *welcome* message, it can assume that either the *primary* crashed or the connection between the two was lost. Either way, it issues a recovery request, this time without updating the *running* transactions, since they are all still valid.

The flowchart presented in Figure 3 resumes the recovery process.

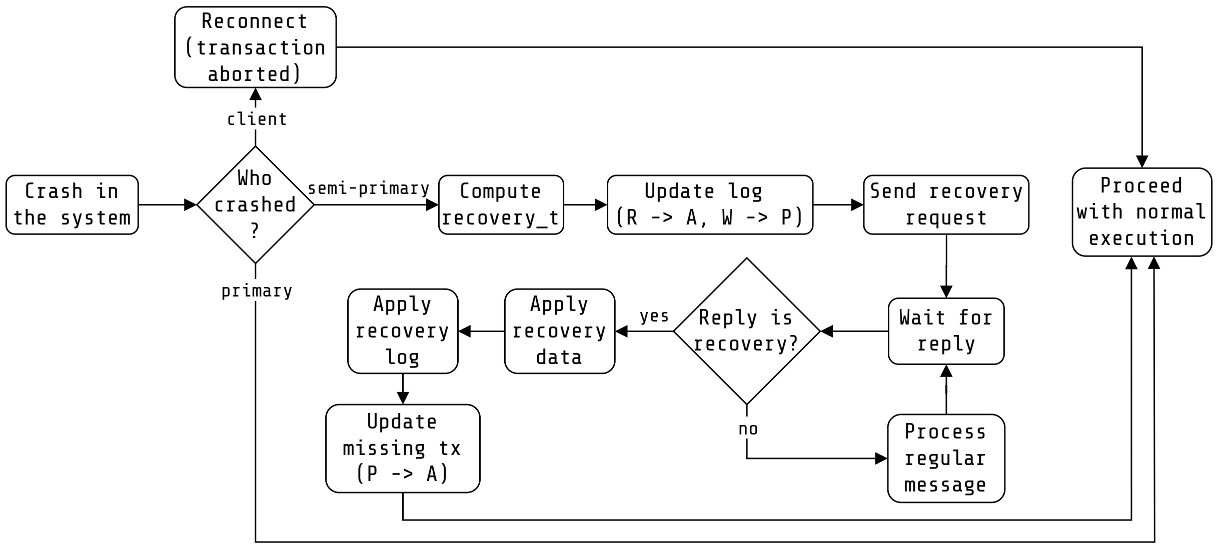


Figure 3: Recovery in the point-of-view of the *client*, the *primary* and the *semi-primary*

3 This is ensured by making the *primary* process the recovery request only after finishing the certification of that *semi-primary*'s transactions, to avoid the possibility of having a transaction that was supposed to go in the recovery response but was not yet certified.

3.3.6 *Discussion*

The proposed architecture aims to minimize the disadvantages of current replicated and partitioned architectures, by avoiding the need for geo distributed transactions or distributed synchronization methods – such as *two-phase commit*, distributed consensus, distributed locking, among others – respectively, which can negatively impact response times. In addition, *semi-primaries* can be deployed strategically next to clusters of clients in order to reduce network latency (reads and writes are done with small latencies, only the commit step requires a possible longer network distance). Also, we can deploy as many *semi-primaries* as we want to scale both reads and writes, as transaction execution and certification performance is not limited by its number. The fact that *semi-primaries* evolve independently avoids the need for clients to wait before starting/finishing transactions. Moreover, the fact that transactions are only certified once, by the *primary*, means that there is no repeated work at the replicas (unlike systems where multiple replicas execute the certification such as MySQL Group Replication [92], Hyder [19], TAPIR [124], Carousel [121] and others), freeing *semi-primary* resources to execute more useful work. Furthermore, even though data is eventually replicated, a client can immediately read its own writes as long as it uses the same *semi-primary*. Finally, since the *semi-primary* presents a regular interface, it is not limited to key-store models, to *one-round* or other restricted transactions, or requiring to know *a priori* the read and write-set keys (unlike Calvin [114], TAPIR [124], Carousel [121] and others).

Perhaps the most similar architecture to the *Primary Semi-Primary* is the central algorithm that ensures PCSI [39], as described in Section 2.5.2. Just like the *Primary Semi-Primary*, it is based on multiple replicas and the use of a central certification node. There are, however, several differences between the two architectures. First, PCSI requires total ordering of transactions, which forces a transaction to commit only after all previous committed ones have been applied. The *Primary Semi-Primary* instead only requires for causal order to be respected, allowing snapshots to evolve independently and leading to faster response times. Second, the algorithm that enforces PCSI relies on asynchronous but lazy replication, sent in the reply of a transaction certification response. Although this reduces the number of messages in the network, it delays the transmission of the certification reply. In addition, this could mean that a replica that only executes a few update transactions will lag behind, since it does not receive new data frequently. Third, the algorithm's definition states that a transaction must be compared to all previously committed ones, which points to a sequential certification, reducing throughput and limiting scalability. In contrast, the *Primary Semi-Primary* only requires sequential certification for transactions whose write-sets conflict. Finally, the log-based storage of the transaction manager means that the algorithm that enforces PCSI might need to rely on aborting transactions that might have been conflict-free.

Conversely, the *primary* in the *Primary Semi-Primary* has a copy of the most recent data present, preventing this problem. In addition, a new *semi-primary* can be connected at any time and be able to recover all data, unlike the central algorithm that enforces PCSI, whose transaction manager only stores the data that might not be applied by the other connected replicas.

Unlike middleware over NoSQL databases that also uses central certification such as pH1 [26], the *Primary Semi-Primary* reads and writes are done directly to the database, without having the need use a proxy in order to store the writes sets, reducing overhead. Furthermore, since it has full control of the distribution, it avoids writes on multiple nodes, possibly distant nodes, again improving response times.

The main disadvantage of having a centralized certification is the fact that it becomes a single point of failure, so it has to rely on failover. Another disadvantage is the fact that since in the *semi-primary* the client writes are redirected to temporary structures as unique inserts, executing transactions under a OCC method, it is not feasible to detect conflicts while a transaction is running, potentially leading to wasted times. However, this is mitigated with the *semi-primary*'s local certification.

3.4 IMPLEMENTATION

The implementation of the *Primary Semi-Primary* architecture is useful as a prototype, that can be experimentally evaluated and compared to state of the art alternatives, but is also a demonstration of how components of the architecture can be mapped to existing database engines and leverage existing query processing and transactional systems. In fact, a key feature of the proposed implementation is expressing key operations (e.g. certification) in terms of database queries. The source code can be found at `https://github.com/nuno-faria/p-sp`.

3.4.1 *Semi-Primary*

Database engine

The integration of SQL query processing in the *semi-primary* is desirable to fully demonstrate the proposal and to be able to compare it with existing alternatives running industry standard benchmarks. In addition, SQL systems can adapt to a wider range of use cases, given most can now directly index and query JSON fields [98]. We have two implementation options: use a traditional ACID SQL database or use a SQL engine layered over a NoSQL store. Using an ACID SQL database would mean that only a small part of the concurrency control would have to be implemented, as some operations could rely on the native isolation, but would be harder to integrate due to its monolithic nature. Using a SQL engine on top of a NoSQL store could provide better performance, given that we want to implement our own custom isolation, that could be more easily integrated as a middleware layer. In short, we have to consider both performance and ease of integration.

Regarding performance, to find out which is the better option, different query engines are deployed on top a Cassandra and/or MongoDB stores. The native performance of Cassandra, MongoDB and PostgreSQL is also tested. The micro benchmark uses the query `SELECT x, SUM(y) FROM Z GROUP BY x` (or equivalent), with `Z` being defined as `CREATE TABLE Z(x varchar, y int)`, with `x` indexed. The database size consists of 250k entries, with 20% of unique `x` values. Each test runs the query for 20 times, sequentially, reusing the initial connection in order to reduce overhead. The first run is ignored. The means obtained are displayed in Figure 4[4]. Given that the PostgreSQL results surpassed even the native results of both Cassandra and MongoDB, it was decided to use a traditional SQL database.

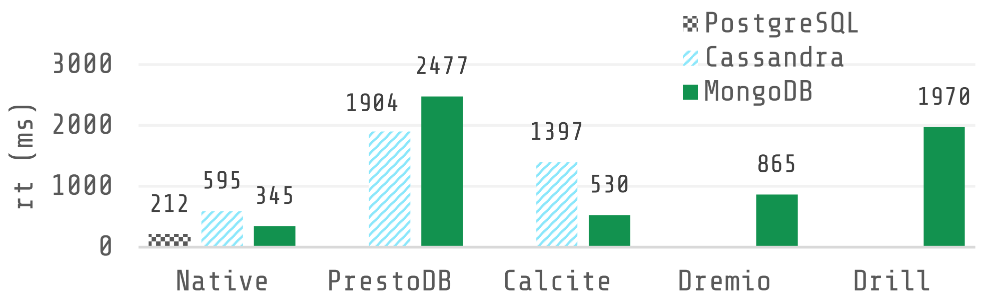


Figure 4: Comparison of `SELECT x, SUM(y) FROM Z GROUP BY x` using various Engines/DBMS

Since it is going to be implemented possibly complex queries, the Orca[43] query optimizer is tested, using the Greenplum database system (based on PostgreSQL), also designed for analytical workloads. When a complex query is executed (namely nested SELECTs), the Greenplum database reverts to using the PostgreSQL planner. This limitation lead to ignoring this engine.

Regarding the ability to integrate the proposed isolation mechanisms, the focus is on the ability to do so using the SQL language itself. In fact, the reason for choosing PostgreSQL is that it, unlike MySQL [5], offers a way to completely rewrite the logic of INSERT, UPDATE and DELETE statements [53], which allows to implement the isolation in a way that hides most of the complexity in the database itself.

Finally, in order to improve response times, the PostgreSQL `synchronous_commit` is set to `off` [52]. This allows for the client to receive a quicker response at the cost that a transaction can be lost if the database crashes. The disadvantage, however, is not a problem in this system, as there is the guarantee that data is persisted in the *primary*.

4 The SparkSQL engine was also tested with Cassandra. However, it presented abnormal results.
5 MySQL requires for the underlying table of a view to have a one-to-one relation to be updatable, not allowing any type of logic rewrite [89].

Schema and Operations

To implement the custom isolation defined in Section 3.3.2, given that PostgreSQL is used as the *semi-primary* database, we only need to store the deleted flag and the *global_t* timestamp. We don't need to store the *from_t* and *to_t* since we can use Snapshot Isolation in PostgreSQL (selected with Repeatable Read) to provide local snapshots. Table 1 represents the schema transformation needed. It takes the original table and converts it into two, to store the persistent (`T_persistent`) and temporary data (`T_temp`), and a view (*T*), that returns the correct data from both tables , i.e. no deleted rows, only the current transaction's temporary writes and no auxiliary columns). This transformation makes it possible to hide the implementation behind the view, making it easier to be adopted by application developers.

Table 1: Before (a)) and after (b)) converting the table *T* in the *semi-primary* to use the custom isolation.
Bold column names represent primary keys.

a)

T

k1	***k2***	*v1*	*v2*

b)

T (view)

k1	*k2*	*v1*	*v2*

T_persistent

k1	***k2***	*v1*	*v2*	*del*	*global_t*

T_temp

tx_id	***k1***	***k2***	*v1*	*v2*	*del*	*global_t*

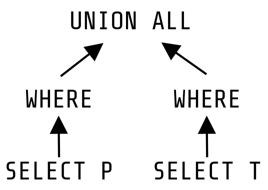
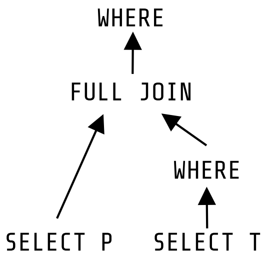
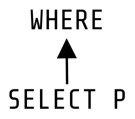
For the Select statement, three different implementation options are considered, as presented in Table 2. Since all three options can be advantageous in certain situations, an option is provided to change the Select mode at any time, even during a transaction's execution.

When it comes to writes, they must be inserted in the temporary tables. The PostgreSQL concept of *rules*, that allow us to define the behavior of the Insert, Update and Delete statements made to a view, is used to achieve this task. The rules simply make the new row(s), generated by the statement, be inserted to the temporary tables, together with the transaction id and the respective *global_t*[6] (or 0 if it is a new insert). For the Delete operation, it is also set the *deleted* field to true.

With these changes, the exchange between the client and the database is as close as possible to regular SQL store. The only extra required steps are setting the Select mode and issuing *begin* and request certification to every transaction (the former two directly to the database and the latter one to the *semi-primary*'s middleware). To facilitate schema creation, there is a Python script to automatically fulfil this task.

6 Due to a limitation of the database engine, to be able to get the respective *global_t*, it is necessary to return the *global_t* column in the view.

Table 2: Comparison between different types of Select in the *semi-primary*

Name	Description	When to use	SQL
union	Queries the persistent and temporary tables, combining the results with a Union	Query returns a relatively small number of rows, since the Append operation to implement the Union is heavy for a large number; makes better usage of the available indexes.	
fulljoin	Full joins the persistent and filtered temporary tables and then filters the result	Query returns a large number of rows, since its faster than Append, but might not be able to use the necessary indexes because of the join.	
nryotw	Doesn't consider temporary writes ("not read your own temporary writes")	Whenever the query doesn't need to read its temporary writes; faster performance than all the other options.	

P – persistent table
T – temporary table

Middleware

The middleware handles the *semi-primary* logic the database can't handle itself. This includes receiving and processing certification requests, requesting global certification and receiving and applying transactions. The choice of language for this middleware is C++, chosen because of its high performance capabilities. The system is comprised by five different components:

- *main* (number of copies: 1) – receives messages from the clients, its workers and the primary, assigning jobs to its workers and replying to client;
- *lworker* (local worker) (1..*) – processes jobs related to transactions local to that *semi-primary*, such as preparing and issuing global certification requests and applying local transactions after *primary* response;
- *rworker* (replication/recovery worker) (1) – processes replications and recovery response from the *primary*; applies transactions by the order of their *global_t*;
- *gc* (garbage collector) (1) – responsible for periodically calling the database procedure that cleans temporary data;
- *monitor* (1) – periodically sends heartbeats to the primary, so it can know the *semi-primary* is still active.

To facilitate network communication, ZeroMQ[7] is used. Not only this abstracts network components such as low level sockets, making communication easier, but also allows to easily implement the *main* event thread in a non-blocking fashion, since ZeroMQ directly handles multiple connections to be processed by one thread without the need of having one listener thread per connection. The need for a monitor comes from the fact that ZeroMQ, being a high level communication framework, also used by the *primary*, does not provide the ability to tell if a connection was closed.

Jobs are distributed from the *main* to the *lworkers* using a blocking queue[8]. For the replication/recovery jobs, it is also used a blocking queue to abstract the wait/notify paradigm. The queue used can't store the replication jobs directly since it does not guarantee total order, thus being used only as a notification framework.

To serialize message communication between *semi-primary* and *primary*, Protocol Buffers[9] is used, mainly due its fast speeds and simple usage.

To enable customization, a YAML config file[10] contains several options that can be tweaked to user preference, such as number of *lworkers*, the garbage collector timer, database and *primary* connections, and so on. Among these, it is important to refer to the *local_certification* and *wait_for_apply* options. The former tells the *semi-primary* to or to not process the local certification, which can allow for a faster response time and reduce *semi-primary* load if we know that the probability for transactions to abort is small. The latter tells the *semi-primary* to or to not wait to reply after the transaction is applied. Turning *wait_for_apply* off should yield better response times by sacrificing consistency, since the client might not be able to read its modifications right after receiving confirmation that the transaction committed.

To summarize, Figure 5 presents an overview of the *semi-primary*'s internal architecture and the communication with the other components.

3.4.2 *Primary*

Database engine

Unlike the *semi-primary* database, the *primary* one does not process clients' requests, and therefore does not need to provide any specific interface. For this reason, the choice to pick a database will be solely based on performance and isolation. The *primary* database has two tasks: store data to provide to the *semi-primaries* that issue a recovery request and to check if a transaction can commit. The first task does not demand any special requirements, since we can afford the recovery to be a slower

7 `https://zeromq.org/`
8 `https://github.com/cameron314/concurrentqueue`
9 `https://developers.google.com/protocol-buffers/`
10 The configuration options can be overwritten by passing command line arguments.

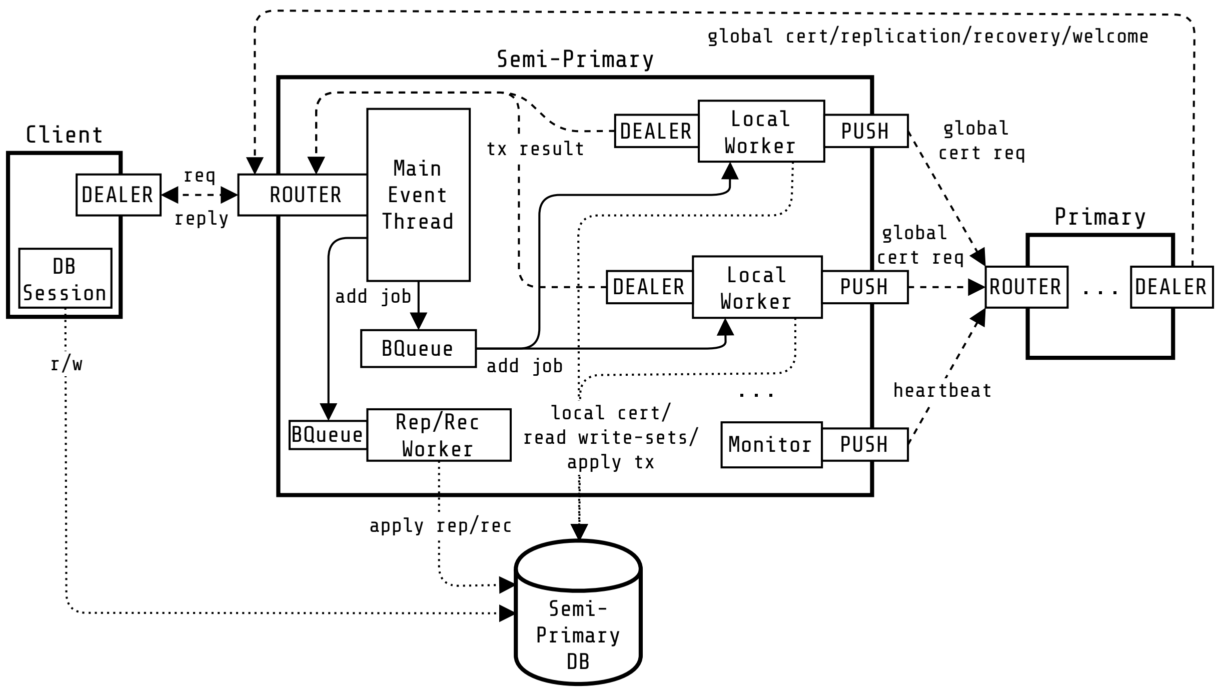


Figure 5: Architecture of the *semi-primary* implementation

process, given that it should be rarely performed. The second one, however, must be as fast as possible, as it is executed once for every transaction. To guarantee a fast global certification time, we need to not only rely on a fast database, but also make sure our data access is as efficient as possible, by performing the certification directly in the database (as to not load large datasets into memory). Since the certification is a process that requires some operations that are easily solved by joins (namely finding conflicts and updating old data), a SQL database is used, namely PostgreSQL. Although it could be possible to solve the certification entirely in database using a NoSQL store such as MongoDB, the implementation would definitely be more complex, with no guarantees that it would actually be any faster. The choice of a database that natively supports Snapshot Isolation also means that we can push the entire certification to the engine itself, certainly leading to reduced overheads.

Storage and Certification

To reduce certification complexity, all persistent data is stored into a single table, where the primary key is the original table's name concatenated with the original column's primary key (e.g. `customer.jdoe12`). This way, conflict detection can be achieved with a single SQL query instead of multiple ones, greatly improving performance. The actual original column's data (primary key(s) and values) is stored in a single, binary field. There is also a table that stores the results of all executed transactions (log), used in the *semi-primary* recovery and duplicate client requests.

Briefly, the certification process for a transaction T is implemented as follows:

1. First, it is verified if T was already processed, by checking its id (*tx_id_*) in the log (in case of a crash and duplicate request), returning its result if it was;

2. If not, the transaction's write set is inserted into an in-memory, unlogged table;

3. Next, T's write set is joined with the persisted data by their primary keys and the result is filtered to only return the columns whose persistent *global_t* is bigger than the respective temporary columns' *global_t*:

```
SELECT COUNT(*) INTO n_conflicts
FROM Data_Temp
JOIN Data ON Data_Temp.pk = Data.pk
WHERE Data_Temp.tx_id = tx_id_
    AND Data.global_t > Data_Temp.global_t;
```

4. If the above query finds any conflicts (i.e. `n_conflicts > 0`), T aborts;

5. If not, T gets assigned a `GLOBAL_T_` and its data is persisted (and the log updated):

```
INSERT INTO Data (pk, global_t, table_name, data)
SELECT pk, GLOBAL_T_, table_name, data
FROM Data_Temp
WHERE Data_Temp.tx_id = tx_id_
ON CONFLICT (pk) DO UPDATE
SET data = EXCLUDED.data,
    global_t = GLOBAL_T_;
```

The certification process runs on a transaction under the Repeatable Read isolation. This prevents possible conflicts from concurrent certifications, guarantees atomicity and ensures that the underlying concurrent control is efficiently implemented.

Middleware

Just like the *semi-primary*, the *primary* middleware is implemented using C++. It is composed by:

- *main* (number of copies: 1) – receives the *semi-primaries'* transaction certification and recovery requests and assigns jobs to the executors;
- *executor* (1..*) – certifies and commits transactions; assigns jobs to *replicators*;
- *replicator* (1..*) – sends committed transactions' data to all the *semi-primaries* connected (except the one that issued the transaction), with total order guarantees;

- *recoverer* (1..*) – processes the recovery requests;
- *monitor* (1) – receives heartbeats from the *semi-primaries* and removes inactive *semi-primaries'* connections;
- *connections* (1) – stores the sockets of all *semi-primaries* connected (static class).

The frameworks used for the *primary* are the same to the *semi-primary* when it comes to communication and job distribution.

To handle replication, it was decided that it would occur as soon as a transaction finishes (no batching). The *replicators* reuse the Protocol Buffer messages sent by the *semi-primaries*, updating just a few fields. This way, there is no need to build a replication message from scratch, saving processing power, trips to the database and reducing semi-primaries' snapshots age.

Figure 6 summarizes the internal architecture of the *primary*.

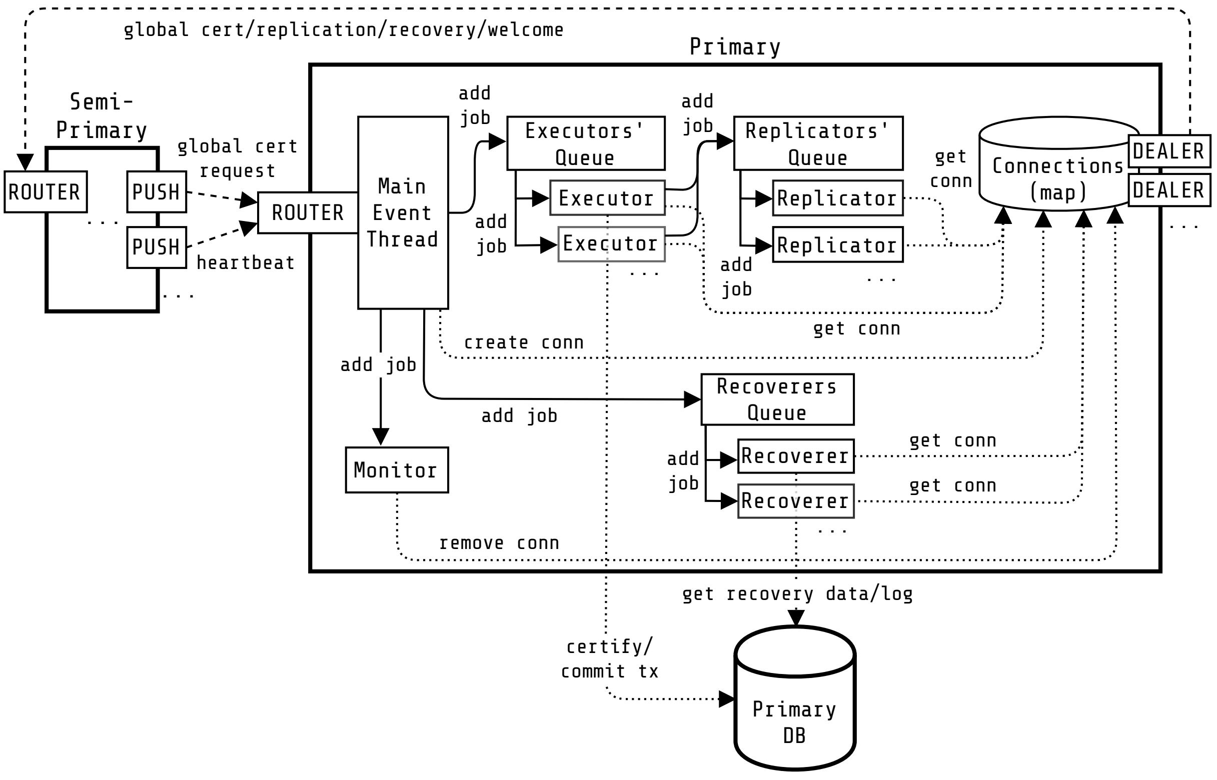


Figure 6: Architecture of the *primary* implementation

Availability

High availability can be effortlessly implemented using *synchronous streaming replication* in PostgreSQL. With this, there is no need to implement a new one, as the existing one presents guarantees of a stable and efficient implementation. When the *primary* fails, we only need to promote the standby database, start a *primary* that uses it and assign the floating IP to the new machine. Because we are using ZeroMQ, there is no need to manually reopen connections. Also, because the new *primary* has no memory of the previous *semi-primaries*, each time a new one sends a request or a heartbeat it will receive a "welcome" message, triggering its recovery. The diagram in Figure 7 sketches an example of a top level architecture to ensure high availability.

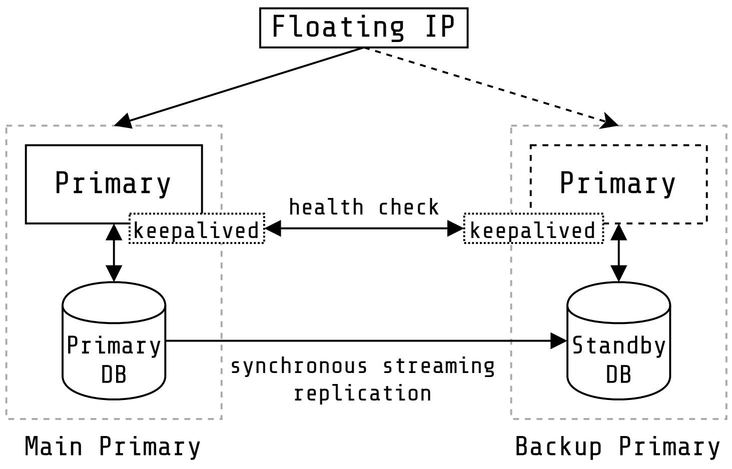


Figure 7: Example of a possible implementation to ensure high availability in the *Primary*.

3.4.3 *Client*

Given the fact that we are using PostgreSQL for the database and ZeroMQ for the communication with the *semi-primary*, the client can be implemented in several languages. The implemented API only needs to implement the methods `connect` to connect to both the database and the *semi-primary* (can be done directly in a constructor if using object oriented programming), `begin` to start a transaction and return its id and `commit`, which sends the transaction's id to the *semi-primary* and returns the result. The API should also support the methods `commit_async`, that issue a commit request without waiting for the response, and `get`, that waits for the *semi-primary* response, blocking if not yet available. To ensure correct system operation under failures, the client must retry the transaction request in case the primary or *semi-primary* failed (e.g. retry every 10 seconds).

3.5 EVALUATION

This section evaluates and discusses the performance of the *Primary Semi-Primary* using the implementation described in Section 3.4, to answer the following questions: What is the overhead of the certification protocol? How well does the *primary* scale to multiple *semi-primaries*? Does the architecture offer advantages when using multiple *semi-primaries* in a local network? What about when geo-distributed? How does the *Primary Semi-Primary* compare to other multi-database solutions? What is the cost of ensure high availability?

All the scripts used to deploy and obtain the results can be found at `https://github.com/nuno-faria/p-sp`.

3.5.1 *Benchmark and architectures*

The performance is evaluated using the TPC-C benchmark.[11] When using multiple client threads, the databases are populated once at the start using 64 warehouses.

The *Primary Semi-Primary* (`p-sp`) is compared to the baseline PostgreSQL (`native`) and three state-of-the-art competitors:

- *Primary Standby* - a single node that handles writes, replication and load balancing, with multiple ones that process reads. Implementation by Pgpool-II [48] (`pgpool`);
- *Multi-Primary Replica* - multiple nodes that handle reads and writes, with certification processed by all nodes in the same order, determined with a consensus algorithm. Implementation by MySQL Group Replication [92] (`mysqlgr`);
- *Multi-Primary Shards* - a single coordinator node that distributes transactions to one of multiple worker nodes that handle both reads and writes, each with a subset of the data. Implementation by Citus [30] (`citus`).

It would be also worth comparing the *Primary Semi-Primary* to other commercial available systems, such as Amazon's Aurora or Google's Spanner. However, the fact that they are closed source solutions makes them difficult to directly compare against, as its not feasible to ensure similar deployments.

3.5.2 *Environment*

All tests are executed on Google Cloud Engine (GCE) virtual machines (Series `N1` CPUs), using Ubuntu 18.04 LTS. The version of PostgreSQL used is always PostgreSQL 12, as well as Pgpool-II 4.1.2 and Citus 9.4. MySQL 8.0.17 is used for the native MySQL and MySQL Group Replication tests, the latter with the number of applier threads for executing replications in parallel (`loose-slave-parallel-workers`) configured to 1024. Both the TPC-C client and the MySQL servers are deployed using Docker containers (19.03.12), in order to simplify deployment.

11 `http://www.tpc.org/tpcc/`, Implementation by `https://github.com/Percona-Lab/sysbench-tpcc`

The number of *semi-primary* local workers is set to double the number of CPU cores [12], while the number of *primary* executors is set to 8 times.

Pgpool-II relies on PostgreSQL's streaming replication, as recommended by the developers [46]. In addition, to better simulate the *Primary Semi-Primary* architecture, `synchronous_commit` is set to `local`, meaning the primary can return to the client before replication occurs. Unlike the *Primary Semi-Primary*, however, this does not guarantee that a client can read its own writes immediately after their transaction finishes. The parameter `disable_load_balance_on_write` is set to `off`, meaning reads in a transaction can be balanced even after writes [47]. The primary only processes writes, meaning all reads are balanced across the replicas.

The Citus test is configured with 64 shards, using the warehouse identifier as the partition key for all tables expect *Item*, as it is not specific to a single warehouse. It is instead modeled as a reference table, that creates a copy at each worker [33]. Since all operations in the TPC-C benchmark are contained in a single shard, this should be the optimal distribution method.

Finally, both PostgreSQL and MySQL use their default configurations, the exception being the maximum number of connections.

3.5.3 *Results*

Certification overhead

The first test evaluates the overhead of the *Primary Semi-Primary* when compared to a native PostgreSQL database. The `native` runs an instance with 8 vCPUs, 8 GB of RAM and 250 GB SSD, while the `p-sp` uses two of those same machines, one for the *primary* and the other for the *semi-primary*. The benchmark runs on a 4 vCPUs instance, using 1, 2, 4, ... and 1024 threads, each with a duration of 5 minutes and a cooldown of 10 seconds. All instances are deployed in the same network. The throughput results are presented in Figure 8.

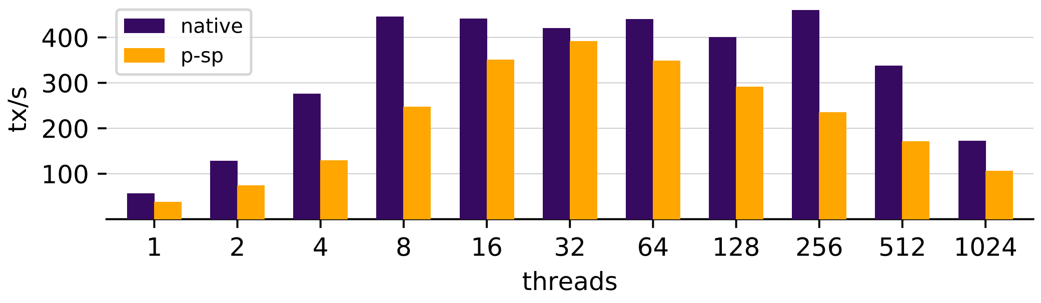


Figure 8: Throughput comparison between the *Primary Semi-Primary* and native PostgreSQL (single *semi-primary*).

12 1 Core = 1 vCPU = 1 execution thread

The *Primary Semi-Primary* throughput is between 45% and 90% the native PostgreSQL. This means that, on average, we would need at least two *semi-primaries* in order to match native performance (assuming linear scalability). The main reason for this can be visualized in Figure 9, which reports that 32% of the transaction time is spent on certification. In addition, there is also some execution overhead, as all operations are rewritten by views and rules. This is the cost of implementing the *Primary Semi-Primary* without changes to the database engine level, using only high level SQL queries and middleware code.

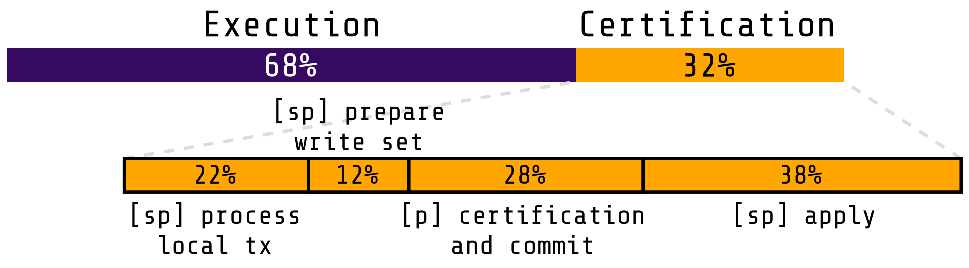


Figure 9: Percentage of a transaction's time used in execution and certification's in the *Primary Semi-Primary*.

Primary scalability

The second test evaluates how well does the *primary* scale relatively to the scale of *semi-primaries*, as it is a centralized component and potentially the bottleneck of the system. To do this, several *primaries* with 1, 2 and 4 cores (and 4 GB RAM) handle the workload from a 16 core, 16 GB RAM *semi-primary*. The client, using a 4 core machine, starts at 1 thread and doubles every 120 seconds, until 1024. There is a 10 second cooldown between every number of threads. The plot in Figure 10 shows the average time taken by the *primary* to process a certification request.

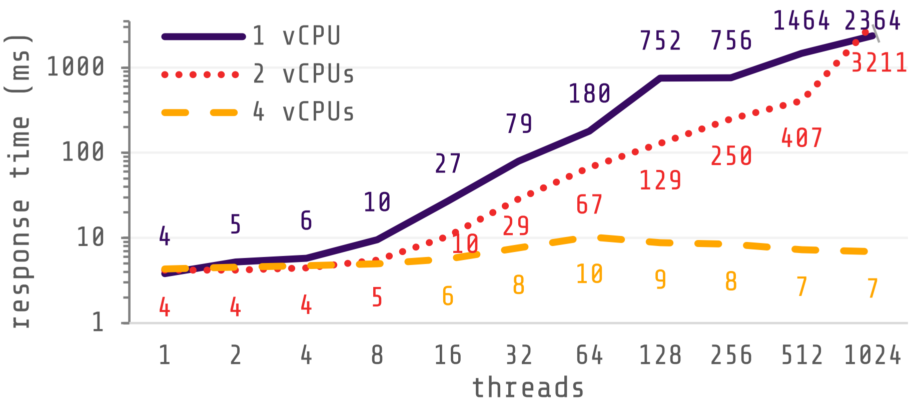


Figure 10: *Primary* certification time with 1, 2 and 4 cores.

These results tells us that for a low to medium load (1 to 32 threads) a single core is enough to quickly handle the certification. For a medium to high load (32 to 128 threads), we need to have a *primary* with at least 2 cores for a relatively quick response time (less than or around 100ms). 4 cores looks to be enough to comfortably handle any load of a 16 threaded *semi-primary*. These results let us conclude two things: 1) just a single core *primary* can handle a significant number of client threads and 2) the primary greatly benefits from multiple cores, therefore proving it can indeed vertically scale. As a side note, one has to keep in mind that the default TPC-C benchmark only has about 8% of read-only transactions. On an application with a higher percentage, we would need relatively less *primary* processing power to handle the same number of *semi-primaries*, as read-only transactions do not require global certification.

Multiple semi-primaries

The next test evaluates the overall *Primary Semi-Primary* performance when using multiple *semi-primaries*. To achieve this, the benchmark is tested against clusters with 1, 2, 4 and 8 *semi-primaries*, together with a *primary* node. Each *primary* and *semi-primary* runs on different machines, each with 2 vCPUs, 8 GB RAM [13] and 50 GB SSD. The native test runs on a single machine with the same specs. The throughput ratio between `p-sp` and `native` (i.e. $\frac{\texttt{p-sp}}{\texttt{native}}$) is compared to the throughput ratio between other architectures and their respective native version [14]. Pgpool tests use 1 primary plus 1, 2, 4 and 8 standbys. Citus tests use 1 coordinator plus 1, 2, 4 and 8 worker nodes. MySQL Group Replication tests use 1, 2, 4 and 8 nodes. To better model this benchmark as a real application, warehouse access is sharded across *semi-primaries*/MySQL Group Replication nodes, to simulate locality. For example, if we have 2 *semi-primaries*/MySQL Group Replication nodes, transactions to the first one updates warehouses 1 to 32 while the second updates warehouses 33 to 64. The client, deployed on a machine with 6 vCPUs, executes with 1, 2, 4, ... and 1024 threads, each for a duration of 2 minutes and a cooldown of 10 seconds. Client's threads are evenly assigned across *semi-primaries*/MySQL Group Replication nodes. In the Citus test, each worker has the same number of shards. The client and servers are deployed in the same network. Throughput ratios between the architectures and the single native counterparts are shown in the heat maps of Figure 11. Response time rations are displayed in a similar fashion in Figure 12. Figure 13 plots the average throughput and response time ratios for each size, to visualize how each architecture scales.

13 The relatively high RAM is required for the MySQL GR to be populated quickly.
14 Citus is compared to PostgreSQL's Read Committed since it is the maximum isolation supported by it.

threads	p-sp				pgpool				citus				mysqlgr			
	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8
1	0.54	0.79	0.71	0.65	0.23	0.31	0.29	0.29	0.55	0.65	0.62	0.78	0.91	0.87	0.75	0.46
2	0.75	0.52	0.78	0.75	0.46	0.48	0.45	0.46	0.77	0.82	0.76	0.83	0.99	0.63	0.84	0.83
4	0.81	0.89	0.73	2.05	1.29	1.37	1.46	1.00	0.89	1.65	1.61	1.68	0.99	0.92	0.59	0.73
8	0.78	1.12	1.53	1.29	1.17	1.22	1.45	1.01	1.11	2.76	2.60	2.76	0.98	0.93	0.84	0.58
16	0.55	1.10	2.05	2.30	0.83	1.05	1.17	0.81	1.07	2.64	2.53	2.74	0.99	0.95	0.89	0.74
32	0.57	0.95	2.26	2.53	0.91	0.99	1.01	0.78	0.98	1.87	1.92	2.15	1.00	0.99	1.08	0.98
64	0.48	0.92	2.38	3.84	0.94	0.88	1.15	0.76	0.75	1.74	1.76	1.84	1.01	1.14	1.21	1.29
128	0.47	0.89	2.24	3.73	0.99	0.95	1.27	0.84	0.80	1.83	1.82	1.88	1.00	1.06	1.21	1.59
256	0.50	0.98	2.25	3.48	1.01	1.07	1.35	1.06	0.72	1.81	1.85	1.90	0.98	1.05	1.22	1.74
512	0.48	1.02	2.53	3.70	1.16	1.19	1.26	1.43	0.77	1.91	1.87	1.95	1.02	1.08	1.29	2.00
1024	0.29	0.68	2.09	3.04	1.11	0.98	1.28	0.92	0.75	1.78	1.77	1.80	1.31	1.37	1.65	2.56

Figure 11: Throughput ratio between the *Primary Semi-Primary* / PostgreSQL and other alternative architectures and their respective native counterparts, for a variable number of nodes and threads, in a local network.

threads	p-sp				pgpool				citus				mysqlgr			
	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8
1	1.84	1.27	1.40	1.53	4.36	3.24	3.42	3.46	1.83	1.54	1.61	1.28	1.10	1.14	1.33	2.19
2	1.33	2.00	1.28	1.35	2.19	2.10	2.22	2.15	1.30	1.23	1.32	1.21	1.01	1.63	1.26	1.31
4	1.24	1.14	1.47	0.51	0.78	0.73	0.68	1.00	1.12	0.61	0.62	0.60	1.01	1.10	1.76	1.57
8	1.30	0.90	0.66	0.78	0.86	0.82	0.69	0.99	0.90	0.36	0.39	0.36	1.02	1.12	1.19	1.74
16	1.82	0.91	0.49	0.44	1.21	0.95	0.85	1.23	0.94	0.38	0.40	0.37	1.01	1.06	1.12	1.45
32	1.76	1.07	0.44	0.40	1.10	1.02	0.99	1.29	1.02	0.54	0.52	0.47	1.00	1.01	0.93	1.03
64	2.07	1.09	0.42	0.26	1.06	1.14	0.87	1.31	1.33	0.58	0.57	0.54	0.99	0.87	0.83	0.78
128	2.11	1.13	0.45	0.27	1.01	1.04	0.79	1.19	1.26	0.55	0.56	0.54	1.00	0.94	0.83	0.63
256	1.97	1.02	0.45	0.29	1.00	0.94	0.74	0.94	1.42	0.57	0.56	0.54	1.02	0.95	0.82	0.58
512	2.01	0.98	0.40	0.28	0.86	0.85	0.80	0.70	1.32	0.54	0.55	0.53	0.98	0.92	0.77	0.50
1024	3.46	1.52	0.51	0.35	0.94	1.05	0.82	1.11	1.36	0.58	0.59	0.58	0.76	0.72	0.60	0.38

Figure 12: Response time ratio between the *Primary Semi-Primary* / PostgreSQL and other alternative architectures and their respective native counterparts, for a variable number of nodes and threads, in a local network.

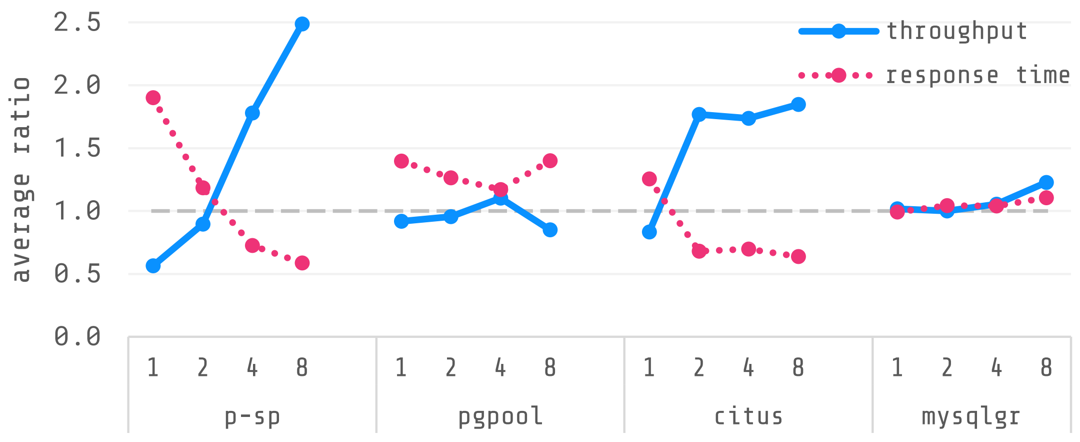


Figure 13: Average throughput and response time ratios between the *Primary Semi-Primary* / PostgreSQL and alternative architectures and their respective native counterparts, in a local network.

In the *Primary Semi-Primary* results we can observe that having just one *semi-primary* yields approximately half the single PostgreSQL throughput, which is in line with the data in Figure 8. Using 2 *semi-primaries* under medium loads results in a similar throughput to the native. For 4 and 8 *semi-primaries*, however, the results are considerably better under medium to high loads, with throughput reaching up to 2.5 times higher with 4 and 3.8 times higher with 8, while response time reaches down to 0.4 times lower with 4 and 0.3 times lower with 8. These results show that the *Primary Semi-Primary* architecture can indeed scale, almost linearly, to multiple *semi-primaries* in a local network. The drop in performance after 64 clients for 8 *semi-primaries* can be justified by the fact that the primary is under more load than it can handle (Figure 10).

The Pgpool results do not display relatively significant performance increases, reaching at most 1.4 times the throughput of the native counterpart. Since the TPC-C is a write heavy benchmark, the majority of operations won't be able to be balanced. In addition, all operations, including Begin and Commit, need to be first processed by the primary node, increasing its load. Furthermore, we can see that having 8 nodes performs worse than having 1, 2 and 4, which can be explained by the fact that, at some point, adding more nodes only increases the replication overhead on the primary without offering any advantages apart from increased availability.

The Citus tests present a significant increase in performance when using 2 workers over 1, reaching up to 2.8 times the throughput and reaching down to 0.4 times the response time of a single native PostgreSQL. However, adding more nodes does not seem to impact the overall performance. By analyzing the CPU load, it was discovered that, for 4 and 8 workers, the coordinator was under full usage before the workers. Just like the Pgpool, all operations have to be processed first by the coordinator, which reduces scalability. To solve this, Citus allows us to add more than one coordinator [32] or the option route the operations directly to the workers [31], although this last solution is commercial only. One has to keep in mind that, for this particular benchmark, this architecture didn't have to deal with distributed transactions, which could have negatively impacted the performance.

Looking at the MySQL Group Replication tests, we can immediately see that using one node has no significant impact on the performance, unlike the *Primary Semi-Primary* and the other architectures. However, unlike the *Primary Semi-Primary*, adding more nodes does not appear to have the same level of impact in the throughput. In fact, at low to medium loads, more nodes actually end up generating less throughput. This can be justified by the fact that, as more and more nodes are added to the system, the more expensive becomes the consensus step to determine certification order. Despite this, using multiple MySQL Group Replication nodes has its advantages. As we can analyze, there is a clear benefit when using it under high loads, with throughput ratio reaching up to 1.7 times higher with 4 nodes and 2.6 times higher with 8, while response time reaches down to 0.6 times lower with 4 nodes and 0.4 times lower with 8.

In conclusion, we can see that the *Primary Semi-Primary* architecture scales relatively well to multiple *semi-primaries* in a local network environment.

Geo replication

This test evaluates the *Primary Semi-Primary* performance in a geo-distributed environment. Since now there is a greater network latency, it is expected that the difference in throughput between *Primary Semi-Primary* and native be higher comparatively to the results in the previous section. Clients with 2 vCPUs are placed in 7 locations across the globe, namely: `us east1`, `us central1`, `us west1`, `southamerica east1`, `europe west4`, `europe north1` and `asia east1`.[15] The geographical locations of these zones can be inspected in Figure 14.

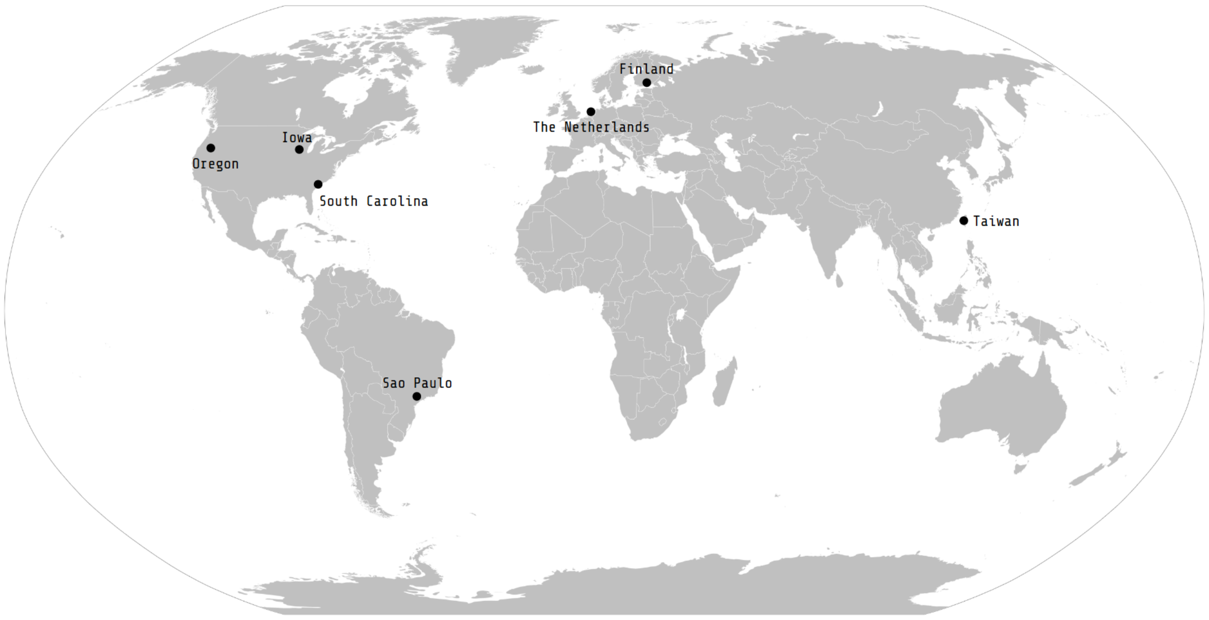


Figure 14: Locations of the clients for the geo replication test.

All clients execute simultaneously, each running tests with 1, 2, 4, ... and 64 threads (so $7 \times N$ threads in total), each with a duration of 5 minutes and a cooldown of 1 minute. The *primary* and the native are deployed in `us east1`. A *semi-primary* is deployed next to each client. In the `p-sp` test, each client accesses the closest *semi-primary*. The throughput difference between `p-sp` and `native` is once again compared to the difference between `mysqlgr` and `mysql`, using a similar deployment. Just like the last test, warehouse access is balanced across clients. Neither Pgpool nor Citus are evaluated since, it does not seem to be possible to specify the standby where the Pgpool load balances a query nor the locations of the shards in Citus. This means that a

15 https://cloud.google.com/about/locations

client could be routed to a distant server in these implementations, which would make it an unfair comparison. Each server is deployed on a 2 vCPUs, 8 GB RAM and 100 GB SSD machine. Figure 15 displays, for each zone, the average throughput for all number of threads. Figure 16 displays the average response times in a similar fashion. Both figures present the zones in ascending order of distance to `us east`. The throughput and response time differences between *Primary Semi-Primary*/PostgreSQL and MySQL Group Replication/MySQL are displayed in Figures 17 and 18, respectively.

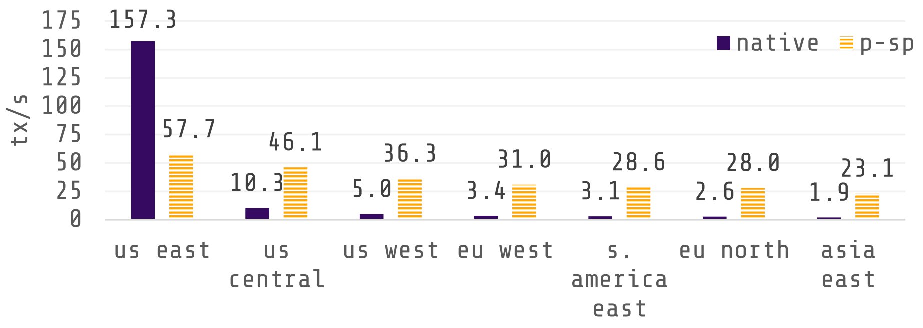


Figure 15: Throughput of the *Primary Semi-Primary* and the native PostgreSQL, in a geo-distributed environment.

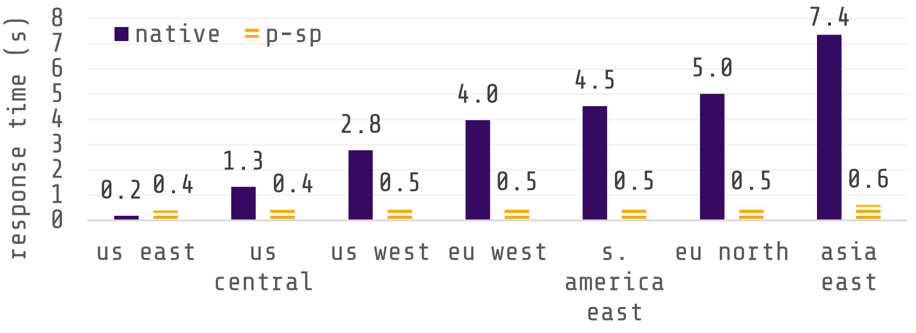


Figure 16: Response time of the *Primary Semi-Primary* and the native PostgreSQL, in a geo-distributed environment.

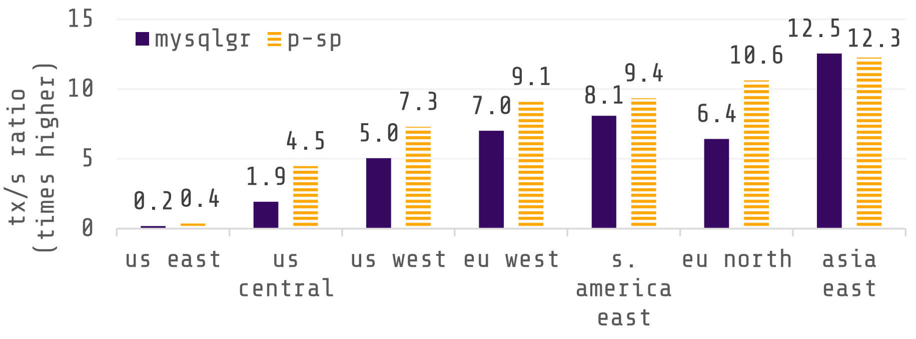


Figure 17: Throughput difference between the *Primary Semi-Primary* / PostgreSQL and MySQL Group Replication / MySQL, in a geo-distributed environment.

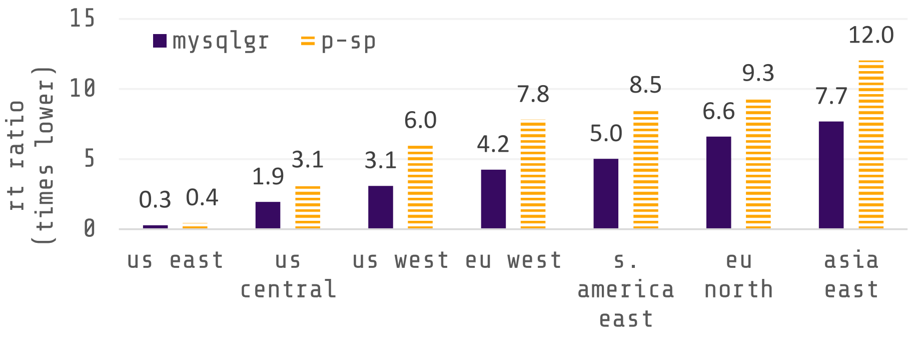


Figure 18: Response time difference between the *Primary Semi-Primary* / PostgreSQL and MySQL Group Replication / MySQL, in a geo-distributed environment.

Analyzing the throughput comparison (Figure 15) we can see that, in `us east` zone, the native PostgreSQL is clearly superior to the *Primary Semi-Primary*, which can be justified by the latter's overhead. However, as the distance between the client and the native database server increases, the *Primary Semi-Primary* presents substantial higher throughput, up to 12 times higher. This is due to network latency between client and server adding up to response time (up to 12 times higher) in native PostgreSQL, as we can analyze from the response time results (Figure 16). For example, if a transaction has 5 read/write operations and the client-server latency is 100ms, the transaction needs to wait 1 second just in communication. In contrast, the *Primary Semi-Primary* would only be affected by that 100ms latency twice, at certification (request and response). The other operations would have a lower response time, as long as the there is a *semi-primary* close to the client. This also explains why the throughput in the *Primary Semi-Primary* decreases relatively less when the distance to the *primary* increases.

As for the results that compare the difference between *Primary Semi-Primary*/native PostgreSQL and MySQL Group Replication/native MySQL (Figures 17 and 18), the *Primary Semi-Primary* shows a relatively higher difference in the response time ratio, as it is expected since MySQL Group Replication relies on a more expensive certification protocol. When it comes to throughput, it is also higher in the *Primary Semi-Primary*, except for the `asia east` zone. Despite not shown here, the native PostgreSQL presented, unexpectedly, a considerably higher response time than the MySQL (but at the same time a higher throughput) for all zones except `us east`. In addition, the overall abort rate was, on average, much higher in the native PostgreSQL (19%) and *Primary Semi-Primary* (25%) than in the MySQL Group Replication and the native MySQL (both less than 1%). Since aborts have a bigger impact on geo distributed systems, as they lead to more wasted work, it is not easy to make a direct comparison between the two architectures, since PostgreSQL and MySQL behave differently. Overall, the *Primary Semi-Primary* presented a throughput 37% higher than the native PostgreSQL while the MySQL Group Replication was 5% higher than the native MySQL. Furthermore, on average, each client in the *Primary Semi-Primary* displayed 7.6 times higher the throughput and 6.7 times lower the response time of the native PostgreSQL while each one in the MySQL Group Replication presented 5.9 times higher the throughput and 4.1 times lower the response time.

Despite not being evaluated, it is expected for both Pgpool and Citus tests to perform worse than the *Primary Semi-Primary*/MySQL Group Replication, as each operation needs to be first routed to a single node.

High availability overhead

Finally, it is evaluated the performance of the *Primary Semi-Primary* with high availability guarantees, given it is an integral part of any database system. To do so, two deployments will be tested: one where the *primary* has a standby copy (`p-sp-HA`), implemented with PostgreSQL's synchronous replication, with the `max_wal_senders` [16] option set to 128, and one without (`p-sp`). Each server is deployed on a 6 vCPUs, 6 GB RAM and 165 GB SDD machine. The tests with high availability use three of those instances (one for the *semi-primary*, one for the *primary* and one for the *primary* backup), while the tests without high availability use two. The client runs on a 4 vCPUs machine, with a variable number of client threads, each executing for 5 minutes with a 10 second cooldown. All instances are deployed in the same local network. Figure 19 presents the throughput of the two deployments.

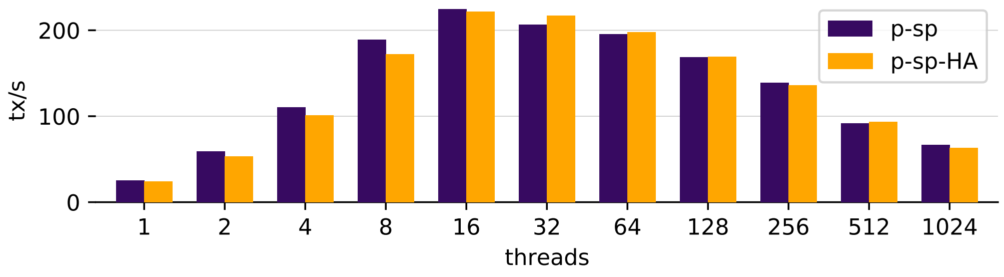


Figure 19: Throughput comparison between the *Primary Semi-Primary* with and without high availability guarantees.

By looking at both results we can infer that there is not a substantial difference between the two deployments. In fact, on average, adding high availability guarantees only incurs a 3% throughput loss. This relatively low variance can be justified by the fact that most of the transaction execution is centered in the *semi-primary* (Figure 9), so the overall throughput will rely mainly on its performance. As we can see, the difference under low loads is higher than the difference under high ones. Therefore, providing high availability guarantees to the *Primary Semi-Primary* is something with little impact of the overall performance.

16 `max_wal_senders` specifies the maximum number of concurrent connections from standby servers to the primary [50].

4

MULTI-RECORD VALUES

4.1 MOTIVATION

Transactional database management systems are designed to safely handle a large number of concurrent requests. Snapshot Isolation has become the choice for many systems to achieve this, as it ensures that read operations do not conflict among them or with concurrent updates and can always proceed, providing optimal performance for read-intensive workloads.

When it comes to handling concurrent updates, the choice is less clear. Database systems offering Snapshot Isolation rely on first-committer wins and rollback transactions to maintain isolation. In particular, when some items are frequently updated, this results in a large number of rollbacks and wasted work. This applies to both centralized and especially distributed systems, where network latencies play a major role in the overall performance.

To better understand how this problem can be improved, the topmost common types of conflicts in the TPC-C[1], a frequently used OLTP benchmark, are analyzed. The results are presented in the Table 3.

The results presented show that most aborts are generated by simple additions/subtractions on the same record. These aborts can be divided in two types:

1. successive increments of one unit to the value to generate unique identifiers, e.g. *increment the next order identifier of some district* in TPC-C (`UPDATE District SET d_next_o_id = d_next_o_id + 1 WHERE ...`)

2. additions to a value, e.g. *add an order's total amount to its warehouse's year-to-date value* in TPC-C (`UPDATE Warehouse SET w_ytd = w_ytd+__h_amount WHERE ...`);

In the first case, conflicts can often be avoided with auto incremented fields (common in SQL), at the expense of being non-transactional – in case of rollback, the counter is not decremented back. There is no high throughput solution to this operation if we want to ensure monotonicity [45].

1 `http://www.tpc.org/tpcc/`, implementation by `https://github.com/rmpvilaca/EscadaTPC-C`

Table 3: Top 5 most common abort causes in the TPC-C benchmark (Repeatable Read). The presented queries concern *updating a warehouse's year-to-date by some order's amount* (1), *incrementing a district's next order identifier* (2), *updating a district's year-to-date by some order's amount* (3), *deleting the information of some new order* (4) and *updating a stock's year-to-date by some amount and incrementing its number of sales count* (5).

TX	**statement**	%
payment	`UPDATE warehouse SET w_ytd = w_ytd + __h_amount WHERE w_id = __w_id`	36.68
neworder	`UPDATE district SET d_next_o_id = d_next_o_id + 1 WHERE d_w_id = _w_id AND ...`	28.27
payment	`UPDATE district SET d_ytd = d_ytd + __h_amount WHERE d_w_id = __w_id AND ...`	27.06
delivery	`DELETE FROM new_order WHERE no_w_id = _w_id AND no_d_id = _d_id AND ...`	3.07
neworder	`UPDATE stock SET s_ytd = s_ytd + _li_qty, s_order_cnt = s_order_cnt + 1 WHERE ...`	2.89

The second case can be mitigated by acquiring exclusive locks early in the transaction, either implicitly in an `UPDATE` statement or explicitly with a `SELECT ... FOR UPDATE`. Although it reduces wasted work, it results in serially executing conflicting transactions, thus impacting efficiency. In addition, it can also be handled with some high-performance solutions already presented in Section 2.4. However, they do not come without limitations. Namely, failing to ensure a limit restriction to the values modeled, meaning they are not viable to model fields that require strong consistency guarantees, such as stocks or balances.

Since both aborting and locking negatively impact transactional performance, especially on geo distributed databases given the significant network latencies, the main motivation is the creation of a data structure that helps ease this problem. The next sections present such structure, together with its algorithms, physical implementation and evaluation.

4.2 OVERVIEW

Multi-Record Values (MRVs) consist in partitioning numeric values over multiple data records whose sum equals the original value. Concurrent operations – namely *add* and *subtract*[2] – on the same value can be spread over multiple records, avoiding aborts or

2 Updating a numeric value without explicitly adding or subtracting an amount (e.g. `SET value = 10` instead of `SET value = value + 2`) also counts as an *add* or *subtract* operation, depending on the

locking contention. This is possible since those operations are commutative, meaning they can be applied in any possible order and the final result will always be the same.

The key property of MRVs is that they are able to preserve a lower limit invariant – for example, a requirement that the value cannot be lower than zero. They could also be implemented to deal with upper limit invariants. However, since its usage is not as widespread as the lower limit ones, it was not considered in the design of the presented MRVs. Besides, an upper limit can be easily modeled as lower limit by inverting the way the operations are done – for example, modeling ticket sales as tickets remaining instead of tickets already sold.

Another interesting property is that they are independent of application semantics and do not require the developer to determine and maintain sensible partitions. MRVs are assisted by background workers that dynamically manage the total number of records, based on the abort rate at any given time. In addition, there are also workers that balance the amount between records of an MRV, to avoid high deviations.

MRVs take advantage of the underlying isolation to handle conflicts. If the underlying database does not support transactional guarantees, records will have to be manually locked at applicational level.

4.3 ARCHITECTURE

The proposal for MRVs to reduce the impact of update hotspots addresses the key challenges in partitioning: How to balance and maintain the appropriate number of partitions? How to enforce global invariants without having to read, update, or lock all the partitions? The approach is inspired by consistent hashing and its use in distributed hash tables (DHTs) [60, 111]. In particular, the use of randomness for spreading the load and in structure to enforce global invariants with local knowledge.

4.3.1 *Overview*

For each numeric column that is transformed into an MRV, an additional table and an n-to-1 relation with the original table is assumed, established by the original's primary key (pk_i). We have thus one or more partition records $v_{i,j}$ for each original value v_i. Each of these is identified by the pair $(pk_i, rk_{i,j})$, where $rk_{i,j}$ is a unique integer between $[0, N-1]$ and contains a partition of the value. Therefore, the original value v_i corresponding to pk_i is reconstructed as the sum of partitions $v_{i,j}$ for all j.

The assumption is that this aggregation is not often needed. Frequent operations will only *add* to or *subtract* from v_i, while enforcing some limit, e.g. $v_i \geq 0$. To achieve

difference between the new and old values. If the difference is negative, it counts as a subtraction, else it counts as an addition.

this efficiently, partitions of a value – records with the same pk_i – are seen as organized in a ring structure of size N, akin to the one found in the Chord DHT [111] and depicted in Figure 20. Solid black circles represent the record partitions, while empty circumferences represent indices that do not correspond to any record.

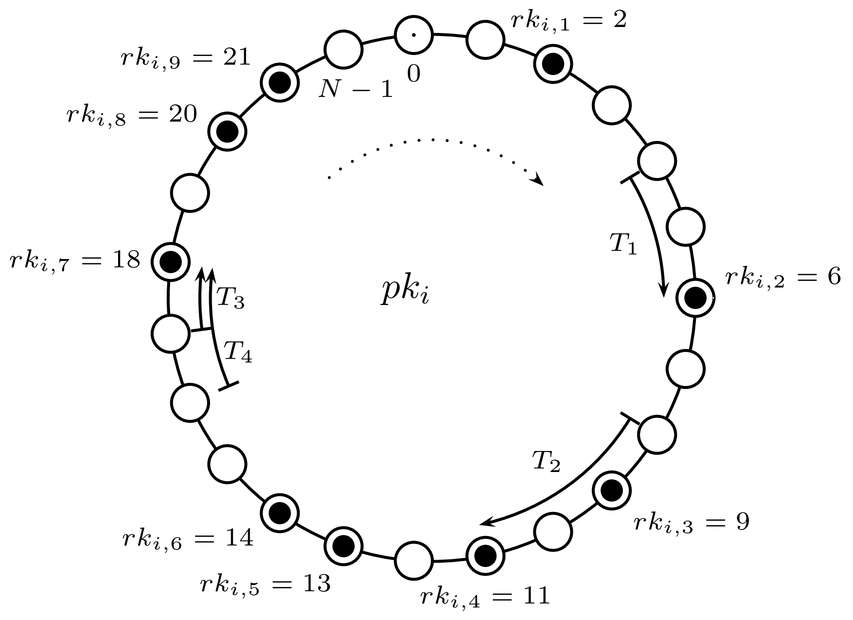


Figure 20: Diagram of the structure of one MRV and various possible transactions alternatives.

To add some value to v_i, one must first pick a random rk', in the range $[0, N-1]$, that will be used to *lookup* a random record. The record selected for update is the one whose rk is equal to rk' or, if none exists, the one that comes immediately after rk' in terms of natural numbers order. This is shown in Figure 20 as T_1, where $rk' = 4$ and the selected record is the one identified by $r_{i,2} = 6$.

To subtract some δ from v_i while enforcing that $v_i \geq 0$, one starts the same way with a random rk' and if the next $v_{i,j} \geq \delta$, simply subtracts the value and then its done. If not, $v_{i,j}$ is set to 0 and the remainder carries on to the next partition. Figure 20 illustrates this with T_2, that starts with $rk' = 8$, sets $v_{i,3}$ to 0 and subtracts the remainder from $v_{i,5}$. A key advantage of this strategy is that, assuming an index on $(pk_i, rk_{i,j})$, a second *lookup* can be avoided by iterating to the next value.

Note that transactions T_1 and T_2 do not conflict and can be adding, subtracting, and checking the invariant at the same time, possibly in different nodes in a distributed system, without interfering. The probability of conflict can be decreased by increasing the number of partitions, but conflicts cannot be fully avoided. For instance, in Figure 20 T_3 starts with $rk' = 16$ and T_4 starts with $rk' = 17$ and both end up trying to modify the value associated with $rk_{i,7} = 18$. MRVs are designed to take advantage of the underlying database's concurrency control. This means that one of T_3 or T_4 is automatically rolled back/locked by the engine, as both write on the same physical record. This can also be achieved manually at applicational level, in case the underlying engine doesn't offer transactional guarantees. Another advantage of this strategy is that by acquiring locks in a predictable order, according to increasing j in $rk_{i,j}$, deadlock probability is greatly reduced.

Finally, there is the case where there is an attempt to subtract from v_i more than the current value, thus violating the invariant. In this case, it will iterate over the entire structure and, once it reaches back to the initial record, it will still have an amount left over to deduct. In this case, the transaction will have to rollback to preserve the lower limit of zero. This is the only scenario where v_i is fully materialized and conflicts with all other transactions on the same item.

The remainder of this section describes each of the operations in detail, then how the number of partitions can be adjusted, and finally how partitions can be balanced to further reduce the need to *lookup* more than once in each *sub* operation.

4.3.2 *Operations*

The detailed description of operations in this section does not explicitly include concurrency control. Instead, it is assumed that these operations run within a transaction in an existing database system that enforces Snapshot Isolation, aborting conflicting transactions, or uses a locking protocol that acquires and holds fine-grained row locks at least for the duration of these procedures. Coarse granularity locking (e.g., table locks) would defeat the purpose of MRVs. We also assume that possible deadlocks are dealt with by eventually aborting transactions.

Algorithm 3 describes the basic *lookup* at the core of every *sub* and *add* operation. It is used to find the first (or next) relevant record while iterating over partitions. In detail, it filters rows corresponding to the same *pk* (line 2) and selects the record with the minimum value of *rk* that is greater or equal to the input (line 3). If not found, it wraps around the ring (lines 4-6). Note that, in practice, this reduces to a single index traversal for the first lookup and an index iteration for subsequent records.

Algorithm 3: Lookup of a record in an MRV (definition)

```
1 Function $lookup(pk, rk)$:
2     $filtered \leftarrow \{ r \mid r \in \text{Values} \wedge r.pk = pk \}$
3     $selected \leftarrow \min_{x \to x.rk} (\{r \mid r \in filtered \wedge r.rk \geq rk\})$
4     if $selected = Null$ then
5         $selected \leftarrow \min_{x \to x.rk} (filtered)$
6     end
7     return $selected$
```

Algorithm 4 shows how to add a positive amount to an MRV: first, perform a *lookup* using a random starting point (line 2) and then update the quantity in the obtained record (line 3). This is simple since the value can always be added to a single partition.

Algorithm 4: Add a value to an MRV (definition)

```
1 Function add(pk, v):
2     $r \leftarrow lookup(pk, random(0..N))$
3     $r.v \leftarrow r.v + v$
```

Algorithm 5 shows the *sub* operation, i.e. subtracting a positive amount from an MRV while guaranteeing that the result is non-negative. In contrast to *add*, it is not as straightforward as we want to guarantee the lower limit invariant. Therefore, after performing a first *lookup* (line 2), the *sub* operation must consider the remaining amount left in the retrieved record (line 6). If the lower limit is set to zero,[3] the operation must guarantee that the remaining amount is greater than or equal to zero, to ensure the MRV's total value is kept above or equal to zero. Thus, just a single *lookup* and subtract might not be enough to deduct the entire desired amount (line 9). To circumvent this, the *sub* operation must keep updating the next records until it subtracted everything it needed to subtract, by performing the *lookup* operation with the current record's $rk + 1$ (line 10). These lookup operations are likely to, in practice, reduce to iterating over the index and thus be highly efficient.

Algorithm 5: Subtract a value to an MRV where the final value ≥ 0 (definition)

```
1  Function sub(pk, v):
2      $r \leftarrow lookup(pk, random(0..N))$
3      $rk_0 \leftarrow r.rk$
4      $done \leftarrow$ False
5      while $v > 0 \wedge \neg done$ do
6          $to\_sub \leftarrow min(r.v, v)$
7          $r.v \leftarrow r.v - to\_sub$
8          $v \leftarrow v - to\_sub$
9          if $v > 0$ then
10             $rk \leftarrow r.rk + 1$
11             $r \leftarrow lookup(pk, rk)$
12             if $rk_0 = r.rk$ then
13                 $done \leftarrow$ True
14             end
15         end
16     end
17     return $v = 0$
```

3 Limits different than zero can be convert them into a zero one, by removing or adding the difference to the original value. For simplicity sake, the remainder of this thesis establishes zero as the lower limit.

Finally, Algorithm 6 is the *total_value* operation, that simply sums the amounts of the entire record set to compute the value of an MRV. Note that with Snapshot Isolation this still does not conflict with any other concurrent operation.

Algorithm 6: Compute the total value of an MRV (definition)

```
1 Function total_value(pk):
2     $partial\_amounts \leftarrow \{\, r.v \mid r \in \text{Values} \wedge r.pk = pk \,\}$
3     $total \leftarrow \sum partial\_amounts$
4     return $total$
```

In terms of time complexity based on the number of partitions for a single record (N), the *lookup* operation is $O(\log(N))$, assuming a tree structured index. The *add* operation is $O(\log(N))$, as only one *lookup* and update are required. The *sub* operation is $O(N)$ as one lookup is needed and then possibly the traversal of all partitions. However, as long as the records are properly balanced, a single partition will suffice and in practice it will be as efficient as *add*. Finally, the *total_value* is $O(N)$ as it cannot avoid traversing all partitions.

4.3.3 *Adjusting to workload*

As the number of records increases in an MRV, the collision probability decreases. However, considering that the number of records in an MRV has a negative impact on read performance and wasted space, the number of records cannot be blindly set to an arbitrarily high value. In addition, the number of total records needs to be limited by the total amount, since a high number of records for a small value is counterproductive, namely for *sub* operations, as it increases the chance having to visit and change multiple partitions. Defining a static number of records is not feasible, given the fact that the load an MRV is subjected to and its total amount can vary over time. To solve these problems, MRVs are aided by a background task responsible for controlling the number of records, designated *adjust worker*.

To highlight the relevance of this issue, it is used a simulation that shows what is the expected abort rate given the number of records, number of transactions per second (tx/s) and transaction duration (fixed to 5ms). It is assumed that each transaction starts randomly between 0 and 995ms and they all update the same value. The abort rate is computed by averaging 10 runs of the given parameters. The results in Figure 21 show the average amount of records necessary to reach some target abort rate, with a variable number of tx/s.

The results of Figure 21 show that as the target abort rate decreases, the number of records required to reach it grows exponentially, therefore configuration needs to

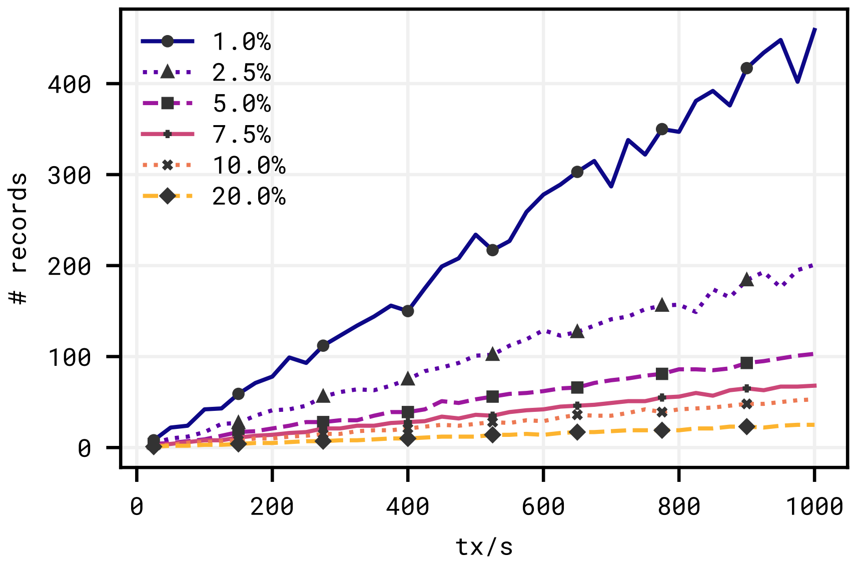


Figure 21: Average number of records in an MRV necessary to reach abort rates of 1%, 2.5%, 5%, 7.5%, 10% and 20%, given a variable number of tx/s, for the custom simulation.

balance an acceptable abort rate (ar_{goal}) with space and time overhead resulting from increasing the number of partition records.

Having set the target abort rate, it is now necessary to define the behavior of the *adjust worker*. We want to have, for some MRV, at any given time, the ideal number of records to reach the target abort rate. However, it is not possible to directly compute what is the number of records required to add or remove from the current structure just based on the current abort rate, as this depends on the workload. The *adjust worker* thus needs to incrementally add or remove records and monitor the changes in the abort rate. Moreover, as this has to be done individually for each record, we want to avoid storing additional state, for instance, to implement a classical PID controller for each record.

Considering that we desire the number of records to converge to the target as swiftly as possible, various algorithms for adjustment can be used, namely:

- *binary* – adds or removes one record (Algorithm 7);
- *linear* – adds or removes the current records times the current abort rate (Algorithm 8);
- *quadratic* – adds or removes the current records times the current abort rate, squared (Algorithm 9).

Moreover, in addition to the target abort rate, it is also necessary to define the minimum abort rate (ar_{min}) at which the worker should remove records. It could also be set to same value; however, it would be expected for the worker to be constantly adding and removing records while trying to keep the abort rate exactly at the target. Due to this, the abort rate at which the worker starts removing records should be set to a lower value. Although these targets vary with applications, for all the remaining results, the target abort rate is set to 5% and the rate at which records are removed is set to 1%.

The adjust worker also has two extra parameters that specify the minimum (nr_{min}) and maximum (nr_{max}) number of records each MRV should have. The latter is especially important since it can avoid MRVs with an unlimited number of records, since those have a negative impact on read performance.

Algorithms 7, 8 and 9 return, for a given abort rate, number of records, record bounds, target abort rate and minimum abort rate, the number of records necessary to add to the MRV (or remove if the returned value is negative).

Algorithm 7: Binary adjust records strategy

```
1 Function binary($ar, nr, nr_{min}, nr_{max}, ar_{goal}, ar_{min}$):
2     if $ar > ar_{goal} \wedge nr < nr_{max}$ then
3         return $1$
4     else if $ar < ar_{min} \wedge nr > nr_{min}$ then
5         return $-1$
6     else
7         return $0$
8     end
```

Algorithm 8: Linear adjust records strategy

```
1 Function linear($ar, nr, nr_{min}, nr_{max}, ar_{goal}, ar_{min}$):
2     if $ar > ar_{goal} \wedge nr < nr_{max}$ then
3         return $min(1 + nr \cdot ar, nr_{max} - nr)$
4     else if $ar < ar_{min} \wedge nr > nr_{min}$ then
5         return $max(-(1 + nr \cdot ar), nr_{max} - nr)$
6     else
7         return $0$
8     end
```

Algorithm 9: Quadratic adjust records strategy

```
1 Function quadratic($ar, nr, nr_{min}, nr_{max}, ar_{goal}, ar_{min}$):
2     if $ar > ar_{goal} \wedge nr < nr_{max}$ then
3         return $min(1 + (nr \cdot ar)^2, nr_{max} - nr)$
4     else if $ar < ar_{min} \wedge nr > nr_{min}$ then
5         return $max(-(1 + (nr \cdot ar)^2, nr_{max} - nr)$
6     else
7         return $0$
8     end
```

The different methods are tested using the same simulation used in the results of Figure 21. The number of transactions per second is initially set to 1000, increasing 15% when the simulation reaches the half mark. Each test starts with an MRV with one record, adjusting every second for the duration of 150 seconds. Figures 22 presents the evolution of the abort rate over time with the different algorithms, as well as the current number of records.

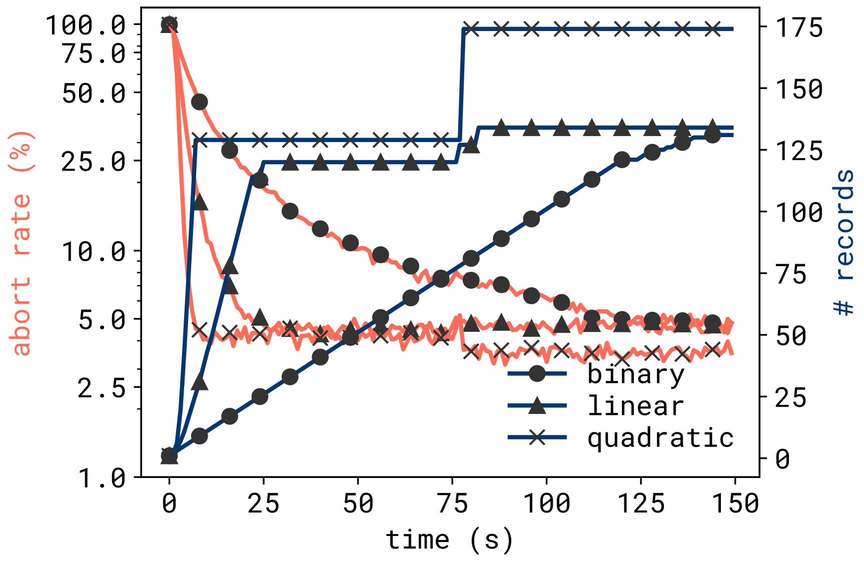


Figure 22: Comparison of the evolution of the abort rate and number of records in an MRV, using different adjust records strategies.

The results in Figure 22 shows that both the *linear* and *quadratic* strategies converge relatively quickly to the ideal abort rate of 5%, while the *binary* seems to take the entire simulation duration and thus can be discarded. Between the *linear* and *quadratic* strategies, we can infer that while the *quadratic* converges almost immediately, it also overshoots the ideal number of records when the simulation increases the load by 15%, meaning it is less stable than the alternative. As a result, the *linear* strategy is considered the optimal one and as such is implemented by the *adjust worker*.

Another consideration when designing the *adjust worker* is the MRV's total value. If the total value is decreased to the point of having more records than units, then we could have a situation where *sub* operations keep aborting because they will always end up updating the same records, as the majority will have zero amount. The *adjust worker* can look at the increased aborts and keep adding more records, not helping solve the problem. To prevent this, there is a configuration that tells the worker to apply the adjust strategy if and only if the ratio between the total amount and the number of records is greater than some `min_average_amount_per_record`. If there is a sudden decrease in the value that pushes the average amount per record below this parameter, the worker will start to remove records to meet the criterion.

The design of the *adjust worker* thus rests on being able to observe aborted transactions as an input. This can be achieved with an additional *tx_status worker* that collects transaction results – asynchronously – to be used by the *adjust worker*. By making this a background task, it minimizes the impact on response time.

Finally, there might be situations where the modeled fields do not generate any aborts and yet multiple records might still be useful to reduce locking overhead. In that case, we could refactor the *adjust worker* to also have in consideration the response time of the updates, together with a *rt_goal* and *rt_min* parameters. In this thesis, however, the number of records is simply set to a static value and the adjust records turned off for those situations.

4.3.4 *Balancing partitions*

Common workloads may lead to situations that would degrade the performance of MRVs. For example, a stock of some product would probably have frequent *sub* operations of a few units – i.e. clients buying the product – and less frequent *add* operations of many units – i.e. the store restocking the item. This leads to most partitions having a zero value and some have a high value, thus defeating the purpose of partitioning. To prevent this from happening, MRVs need to periodically balance the existing value between partition records.

The ideal balancing would be obtained by reading the total value and equally dividing the amount between all records. Although this ensures that the records end up perfectly balanced in just one iteration, it also means that one has to update the entire set simultaneously, which most likely will require a few tries to succeed given that the balancing operation would conflict with any other concurrent update.

Another option is to select just a few random records and balance their sum among them. Although it needs multiple iterations to balance the entire set, it has a low collision probability. Furthermore, picking just two records should be enough for a quick convergence to a low deviation [81].

Finally, there is also the option of selecting the minimum and maximum records based on their amount. This technique allows for a considerably faster convergence when dealing with a relatively high number of records. However, computing the maximum and minimum records takes up more time than just selecting two at random.

All three possibilities above are feasible and can all be useful in different situations. For the results presented in this thesis, the *balance worker* uses the second strategy when dealing with an overall number of *adds* greater than or equal to *subs* (microbenchmark and TPC-C) and the third one for situations with more *subs* than *adds* (STAMP Vacation).

4.3.5 *Discussion*

In theory, MRVs allow for higher throughput and lower response time than the single-record alternative, as they reduce abort probability and/or locking contention to updates on the same numerical object. The usage of workers to adjust the number of records and balance their amounts not only removes responsibility from the developers, but also enables low overhead in both operation complexity and space utilization. MRVs, in comparison to alternatives such as CRDTs, are designed for objects subjected to strong consistency guarantees, such as account's balance or item's stock, as they ensure limit invariants. The fact that MRVs are modeled using a generic structure means that they can be implemented in any database that employs numeric records.

The main disadvantage of MRVs is the fact that they increase the time complexity of read operations, since the real value has to be computed each time. This problem is alleviated by the usage of the *adjust worker*, that tries to ensure that the number of records is kept at the minimum necessary in order to meet the target abort rate. Nevertheless, this problem is, to a certain extent, unavoidable for hotspots. In addition, computing the value each time also stops MRVs from being indexed. However, it is not expected that the type of values that would benefit from MRVs – for example, stock of some item – would require indexing. Given MRVs target write-heavy hotspots, these problems should be outweighed by its performance gains.

Table 4 summarizes the comparison of MRVs and different alternatives. In short, MRVs are the only solution that support limit invariants, are viable in distributed systems - unlike escrow locking - and maximize parallelism - unlike RedBlue, where *sub* operations with lower limit invariants are not commutative, and *timestamp splitting*, that doesn't target same-field concurrency.

Table 4: Comparison between MRV and similar alternatives.

	Locking independent?	Has limit invariant?	Primary parallelization target	Consistency level
MRVs	***yes***	***yes***	***numeric fields***	***strong***
Escrow L.	*no*	*yes*	*numeric fields*	*strong*
Delta txs.	*yes*	*no*	*numeric fields*	*strong*
Timestamp splitting	*yes*	*yes*	*records*	*strong*
CRDTs	*yes*	*no*	*records and various fields*	*strong eventual*
Counting Sets	*yes*	*no*	*set fields*	*strong eventual*
Operation transformation	*yes*	*no*	*various operations*	*strong eventual*
Post-Commit Rules	*yes*	*no*	*various operations*	*eventual*
RedBlue	*yes*	*yes*	*commutative operations*	*strong/strong eventual*

4.4 IMPLEMENTATION

Two main implementation strategies are considered: The first is to rewrite application queries to use MRVs where appropriate, used for the microbenchmark. The second alternative is to make use of views and stored procedures in a SQL database system. For example, in PostgreSQL one can create a view and specify rules on how the INSERTS/UPDATES/DELETES are processed on that view [53]. This alternative is used in the TPC-C and STAMP Vacation benchmarks. A third possible alternative would be to implement MRVs within the database engine itself, but the second approach underlines that MRVs are feasible on existing database systems and that they can be layered on existing transactional and query processing mechanisms.

The first step in both implementations is to modify the schema to accommodate MRVs. As an example, consider table T in Figure 5(a), with a composite key $(k1, k2)$ and two value columns $v1$ and $v2$, for which we want to transform $v1$ into an MRV. As shown in Figure 5(b), this is transformed in tables *T_Orig*, from which $v1$ is removed, and an MRV support table *T_v1*, which includes the original primary key $(k1, k2)$, the partition key *rk* and partition value column $v1$. In the database level implementation, in PostgreSQL, the schema transformation is done by a Python script that automatically generates the schema to convert columns into MRVs. It also generates a view, with the same name as the original table, that joins the two tables and computes the MRV value, together with INSERT/UPDATE/DELETE rules that call the respective procedures. In MySQL that is not possible, as there needs to exist a one-to-one relationship between the view and the table we want to modify [89].

Table 5: Before (a)) and after (b)) converting the column $v1$ of table T into an MRV.

a)

T

k1	**k2**	v1	v2

b)

T (T_Orig + T_v1)

k1	k2	v1	v2

T_Orig

k1	**k2**	v2

T_v1

k1	**k2**	**rk**	v1

The second step is to implement the operations which is straightforward for *lookup*, *add* and *total_value* given the definitions. The exception is the *sub* operation, as it requires one *lookup* per record updated. Two alternatives are considered: First, a single statement implementation that relies on *window* functions, but neither PostgreSQL nor MySQL planners could optimize them well enough within complex queries. The second alternative, that results in good performance, is to use a cursor that performs an extended *lookup* query at the start of the procedure, providing records as they are needed. In addition, the *sub* operation does not perform an update on a record if its amount is equal to zero. The reasons for this are 1) the update would be meaningless, as the value would be same and 2) updates, even on values that doesn't change, are still processed, as in locks are acquired and new versions of that row are created, which impacts performance.

The *sub* code used for the MongoDB database was similar to the formal definition, given the fact that it has limited stored procedure functionally[85]. This should result in a higher overhead for this particular system, as it requires multiple *lookups* and round trips to the database.

As for the *adjust* and *balance workers*, the microbenchmark directly implements their logic together with application code, while the remaining benchmarks use generic, standalone workers implemented in Java. They also provide the option to configure the number of *runners* per worker, to speed up processing.

4.5 EVALUATION

This section aims at answering to the following questions: What is the read and write overheads of MRVs comparatively to the native solution? How does the read overhead change when the number of records per value increases? Does the read/write overhead stay the same in both single and multi-threaded workloads? How effectively do MRVs reduce conflicts? Is it worth to use MRVs in situations where there are no aborts but rather just locking? Does the MRV overhead justify its usage in centralized databases? What about distributed? Do MRVs bring performance advantages to both SQL and NoSQL databases?

To answer these questions, different benchmarks with different database systems are evaluated, comparing the throughput, abort rate and response time of MRVs *versus* the native solution. This section is divided in three parts:

1. Benchmarks – an overview of the benchmarks that will be used to evaluate the performance;

2. Database engines – an overview of the different type of database engines that will be used to evaluate the performance;

3. Results – charts displaying the obtained results, accompanied with a discussion of them.

All the code of the benchmarks, scripts, and raw data is available online at `https://github.com/nuno-faria/mrv-benchmarks`, together with a description on how to build and run them.

4.5.1 *Benchmarks*

Microbenchmark

A microbenchmark, that simply models the stock of several products, is used to evaluate both the read/write overheads and maximum performance gain. For the read test, it obtains the stock of a random product. For the write test, it issues either add or subtract operations of a few items to a random product (read and write tests run separately). In this test, given that both add and subtract operations are of similar frequency and as there is a *balance worker* in place, it is expected for the *sub* operation's time complexity to average its best-case scenario, i.e. one update. By creating a microbenchmark that purely focus on the problem MRVs are trying to solve, it highlights the effective overhead/performance gain, without having to worry about operations other than updating or reading numeric values.

The parameters that configure this microbenchmark are listed and described in Table 6, together with the values used in the experiments. One thing to point out is the number of initial records per value used in the benchmark. Because it is expected for the *adjust worker* to take some time to converge to ideal number of records if each product starts with just one, the number of initial records is set to the same number clients, unless otherwise stated.

The microbenchmark is implemented in Java and compiled with `openjdk-11`. There is a cooldown period of 10 seconds between each test except for the multi-writer cluster architecture (MySQL Group Replication), where the cooldown is set to 30 seconds.

TPC-C

The `sysbench` implementation[4] of TPC-C is used as an example of how MRVs perform in the context of a more general workload.

To determine which columns should be converted to MRVs, the updates that are responsible for update conflicts and lead to concurrency control aborts are examined. The results, summarized in Table 3, show that updates to columns *w_ytd* in *warehouse* and *d_ytd* in *district* are the main hotspots and therefore will be modeled as MRVs.

For the *adjust worker*, each client, after every transaction that uses an MRV, inserts in a table what was the outcome, together with the identification of the MRV(s) used in that transaction. Workers are only used when running under the Repeatable Read isolation.

It is populated with a variable number of warehouses, between 1 to 8, and tested against an also variable number of clients (1, 2, 4, ..., 512). Each test has a duration of

4 https://github.com/akopytov/sysbench/

Table 6: MRV microbenchmark parameters and respective values.

Name	Description	Value(s) used
mode	Mode of the benchmark	read, write
time	Benchmark duration in seconds	60
clients	Number of clients used (each client is represented by a thread)	1, 2, 4, ..., 2048
size	Number of products	1, 2, 4, ..., 128
initialStocks	Initial amount of units for each product in the database	8192
amountLimits	Upper limit on the amount to add/subtract in each transaction (between [1,amountLimit])	3
isolations	Database isolation level	RC*, RR**
maxRecords	Maximum number of records per product allowed	1024
minRecords	Minimum number of records per product allowed	1
initialRecords	Number of initial records per product	0*** (write), 1, 2, 4, ..., 128 (read)
workers	Whether workers are used	true (RR**), false (RC*)
adjustDelta	Time between records adjustment in milliseconds	2000
balanceDelta	Time between records balancing in milliseconds	500
arGoal	Target abort rate ([0, 1])	0.05
arMin	Abort rate below which the *adjust worker* starts removing records ([0, 1])	0.01

*Read Committed **Repeatable Read
***Records per product equal to the number of clients

60 seconds. There is a cooldown period of 10 seconds between each test. The *adjust* and *balance* workers have the same configurations as the microbenchmark.

STAMP Vacation

The Vacation benchmark, included in the STAMP suite [80], simulates a travel reservation system. This benchmark, originally designed for software transactional memory (STM) systems, has a high number of add and subtract operations on numeric values. It comprises five tables: *customer*, *reservation_info*, *car_reservation*, *flight_reservation* and *room_reservation*. Each table is populated with *r* rows.

It offers five different tasks:

- Make reservation (probability: $U\%$) – the client checks the price of n items and reserves a few; items can be rooms, flights and cars;
- Delete customer (probability: $\frac{100-U}{2}\%$) – the total cost of a customer is computed and associated reservations are released; the customer information is deleted;
- Add items (probability: $\frac{100-U}{4}\%$) – increases the stock of n random items;
- Delete items (probability: $\frac{100-U}{4}\%$) – decreases the stock of n random items.

Table 7 presents the statements that generate most conflicts in this benchmark. These results leads the columns *numfree*, *numtotal* and *price* of the tables *car_reservation*, *flight_reservation* and *room_reservation* to be modeled as MRVs.

Table 7: Top 5 most common abort causes in the STAMP Vacation benchmark. The presented queries concern *reducing a reservation's stock in one unit* (1, 2, 4), *increasing a reservation's stock in one unit* (3) and *restocking a reservation and updating its price* (5).

Statement	**%**
`UPDATE car_reservation SET numFree = numFree - 1 WHERE id = $1`	25
`UPDATE flight_reservation SET numFree = numFree - 1 WHERE id = $1`	22
`UPDATE car_reservation SET numFree = numFree + 1 WHERE id in (...)`	22
`UPDATE room_reservation SET numFree = numFree - 1 WHERE id = $1`	19
`UPDATE car_reservation SET numTotal = numTotal + 100, numFree = numFree + 100, price = $1 WHERE id = $2`	2

To obtain the results, a Java implementation is used [5], compiled with `openjdk-11`. Both the number of clients and the number of rows per table are dynamically set (1, 2, 4, ..., 512 and 20, 40, 60, ..., 200, respectively). The percentage of user requests (U) was set to 80%, n was set to 10 and each test ran for 60 seconds. There is a cooldown period of 10 seconds between each test. The amount of initial records per value is the same as the number of clients, the *adjust worker* is called every 4 seconds while the *balance worker* is called every 100 milliseconds.

5 Based on `https://github.com/jopereira/vacationdb`

4.5.2 *Database architectures*

Single instance

A traditional single database instance using PostgreSQL 12, with the default configuration apart from the `max_connections` variable. Both Repeatable Read and Read Committed isolation levels are considered. Note that selecting Repeatable Read isolation in PostgreSQL results in Snapshot Isolation [49] and aborts conflicting update transactions. In contrast, the Read Committed isolation level does not generate aborts on concurrent updates, relying only on statement locking to serialize them.

Single-writer cluster

A MongoDB 4.2.2 cluster with a single-writer database deployed with three nodes in a *replica set* cluster, since MongoDB only supports transactions in multi-node environments [84]. To guarantee consistency similar to Snapshot Isolation, the transactions' executions were setup with the following configuration: `Read preference`[6] – *primary*, to avoid aborts due to stale data; `Read concern`[7] – *snapshot*; `Write concern`[8] – *majority*.

Multi-writer cluster

A multi-writer database cluster, deployed with MySQL Group Replication, where transactions executed at one node are then certified by all others to commit [92]. This is an example of a distributed architecture where conflicts have a bigger impact comparatively to single-writer architectures, given that it relies on a relatively more expensive certification protocol. MySQL Server 8.0.17 is used with the Repeatable Read isolation. The number of applier threads for executing replications in parallel (`loose-slave-parallel-workers`) is set to 1024.

4.5.3 *Environment*

All tests are executed on a single GCE virtual machine, configured with 24 vCPUs (Series `N1`), 24 GB of RAM and a 500 GB SSD, running Ubuntu 18.04 LTS. For the multi-node tests (MySQL Group Replication and MongoDB), Docker 19.03.12 is used to evenly share the resources among them, each being assigned 8 vCPUs and 8 GB of RAM.

6 https://docs.mongodb.com/manual/reference/read-preference/
7 https://docs.mongodb.com/manual/reference/read-concern/
8 https://docs.mongodb.com/manual/reference/write-concern/

4.5.4 *Results*

The results presented in this section are rendered using either bar charts or heat maps. The latter represent the ratio of the MRV over the native performance (i.e. $\frac{MRV}{native}$). A value of $1\times$ means they both have the same result, $2\times$ means MRV has twice the amount, and so on. These heat maps are color coded red or green to represent worse or better results for the MRV architecture, respectively. The higher the throughput ratio the better it is for the MRV. On the other hand, when it comes to abort rate or response time, the lower the ratios the better it is for the MRV. Due of readability concerns, some results were omitted. However, all can be found together with the source code.

Microbenchmark

The first tests performed in the microbenchmark concern the read and write overheads, calculated by measuring the average response times. For the read tests, multiple number of records per value are evaluated (1, 2, 4, ..., 128), to see how the MRV performance varies. The number of products is fixed to 32 and both reads and writes are tested in single and multi-threaded workloads. For the SQL test, PostgreSQL is used with the Read Committed isolation (Figures 23 and 25) so no aborts would occur. As for the NoSQL test, MongoDB is used in a replica set cluster (Figures 24 and 26).

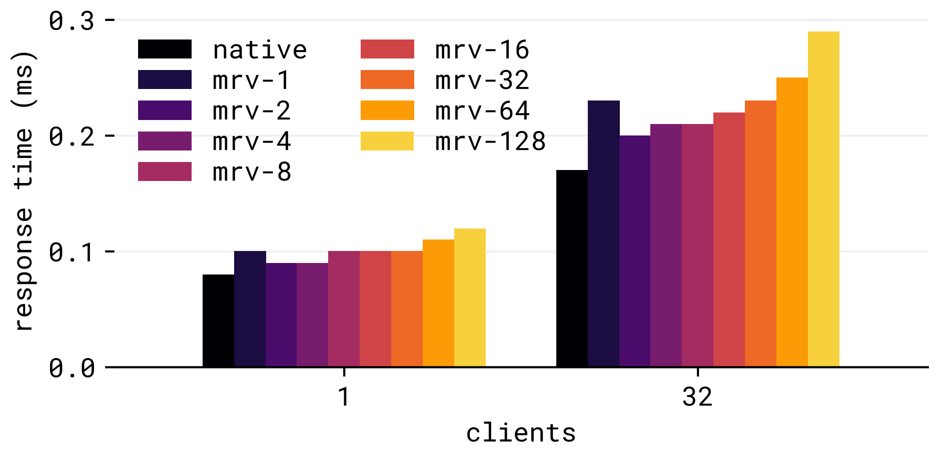


Figure 23: Comparison of the average read response time between MRV and the native, using the microbenchmark with PostgreSQL.

Analyzing the read results for PostgreSQL (Figure 23), the MRV response time comparatively to the native counterpart is between 17% and 60% higher for 1 client and between 15% and 64% higher for 32 clients (average of 30%). As expected, computing the result using a `GROUP BY` + `SUM` is slower than retrieving a simple value. However, although the number of records was increased by a factor of two each run, the average response time increase was relatively small, which shows reads can scale well in terms of number of records for SQL systems. On the other hand, MongoDB's results

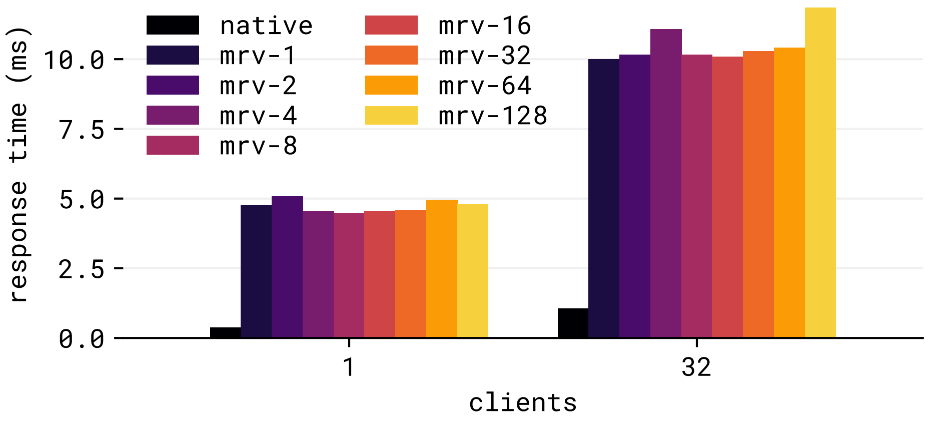


Figure 24: Comparison of the average read response time between MRV and the native, using the microbenchmark with MongoDB.

(Figure 24) show an increase in response time of about 12.5 times for 1 client and 10 times for 32 clients. This can be explained by the fact that the native solution just performs a simple filter, while the MRV solution uses MongoDB's Aggregation framework to compute the sum of the sharded amounts, which most likely has a significant setup overhead. Just like PostgreSQL, the difference in response time between different number of records is relatively small.

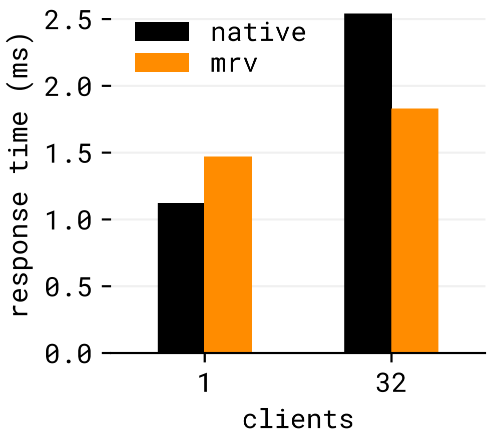


Figure 25: Comparison of the average write response time between MRV and the native, using the microbenchmark with PostgreSQL.

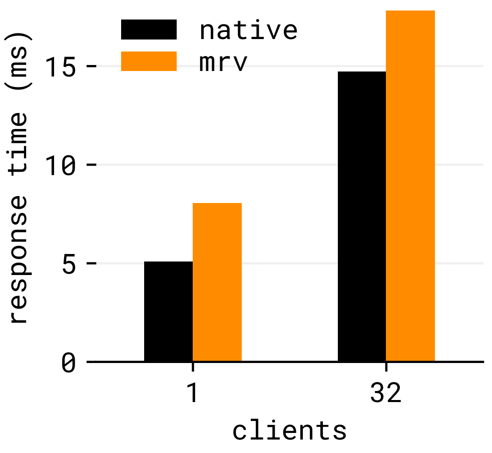


Figure 26: Comparison of the average write response time between MRV and the native, using the microbenchmark with MongoDB.

Examining the write results for PostgreSQL (Figure 25) shows us that the MRV response time is about 30% higher than the native for a single client. However, increasing the number of clients to 32 makes the MRV solution comparatively faster, which justified by the fact that it has less locking contention. With that said, the MongoDB results (Figure 26) show that the MRV is slower in both single and multi-threaded tests, averaging response times 60% and 30% higher, respectively. Just like the reads, the writes had to rely on a slower set of instructions, namely retrieving a random record and then updating it, in contrast with the native's single instruction.

The next tests measure the throughput, abort rate and response time in the different database architectures using the write workload. It starts off with the single PostgreSQL instance, evaluating both Repeatable Read (Figures 27, 28 and 29) and Read Committed (Figures 30 and 31) isolations. Next up, it is displayed the multi-writer cluster using MySQL Group Replication (Figures 32 and 33). Lastly, it is presented the single-writer cluster using MongoDB (Figure 34).

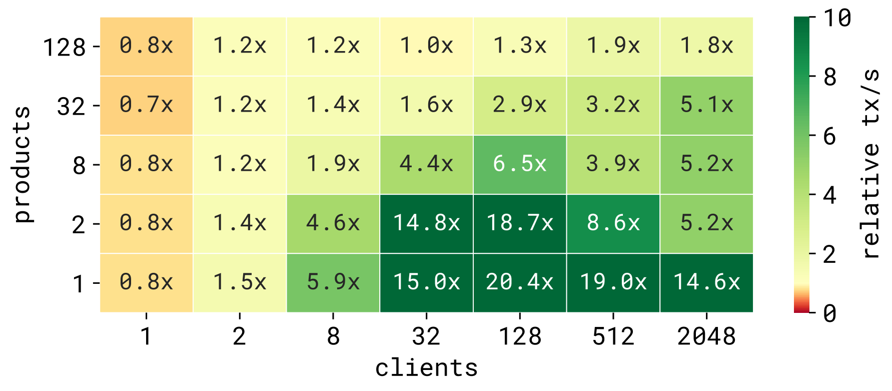


Figure 27: Throughput ratio between MRV and native, using the microbenchmark with PostgreSQL (Repeatable Read).

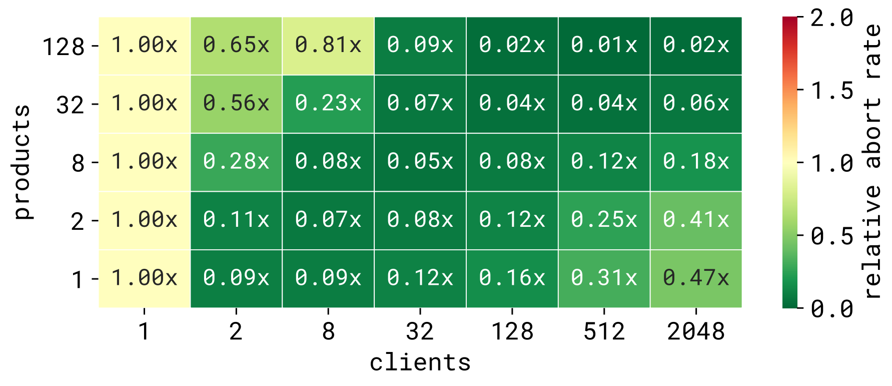


Figure 28: Abort rate ratio between MRV and native, using the microbenchmark with PostgreSQL (Repeatable Read).

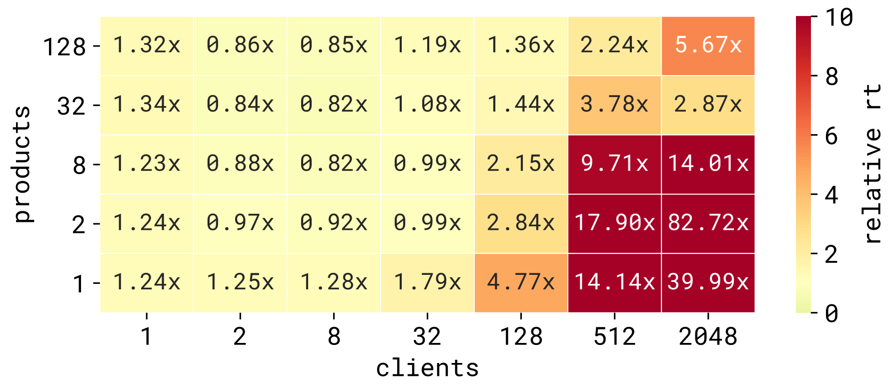


Figure 29: Response time ratio between MRV and native, using the microbenchmark with PostgreSQL (Repeatable Read).

The PostgreSQL Repeatable Read results show a clear MRV increase in throughput over the native as the number of clients increases and the number of products decreases, i.e. when the overall collision probability is higher (Figure 27). The MRV throughput ends up being up to 20 times higher than the native. The only exception is the results with 1 client, where it reaches about 80% the rate of the native, since there are no conflicts and as such MRVs offer nothing but overhead. These increases in throughput are the result of the considerable lower abort rate in the MRV architecture (Figure 28). As transactions are not prematurely aborted due to conflicts, it also causes the response time to increase as there is a lot more useful work being completed (Figure 29). Overall, these results present clear advantages in using MRVs in a single centralized SQL database.

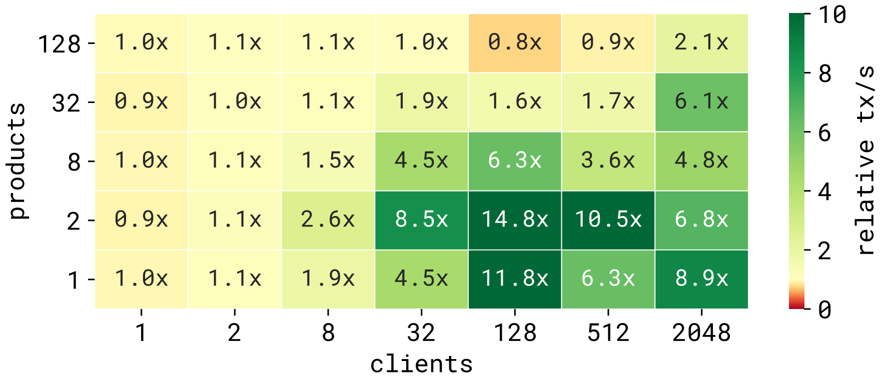


Figure 30: Throughput ratio between MRV and native, using the microbenchmark with PostgreSQL (Read Committed).

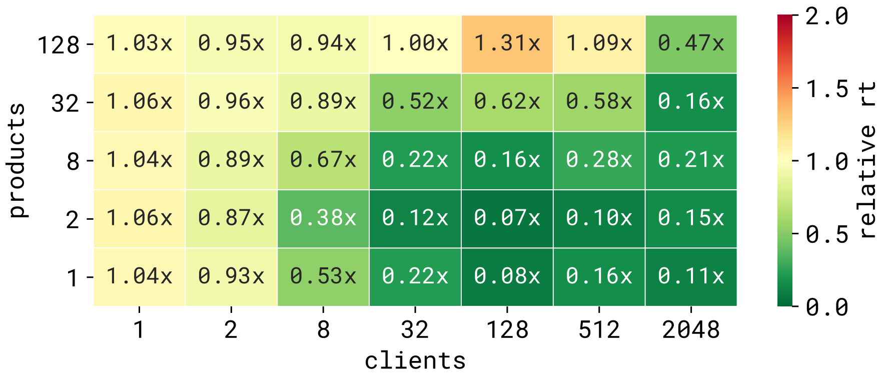


Figure 31: Response time ratio between MRV and native, using the microbenchmark with PostgreSQL (Read Committed).

As for the PostgreSQL Read Committed, MRVs also present a significant increase in throughput (Figure 30), even though the microbenchmark does not generate aborts for this isolation (unless the stock is not enough for a decrement operation, which must rollback). This increase is due to the fact that a transaction in the native solution has to wait more than the MRV when it comes to acquiring a write lock, as we can infer from the response time ratios (Figure 31). Despite the overall MRV throughput ratio not being as higher as the Repeatable Read tests, it still shows an increase in up to 15 times the native, showing that this architecture can also be applied in situations of high lock contention.

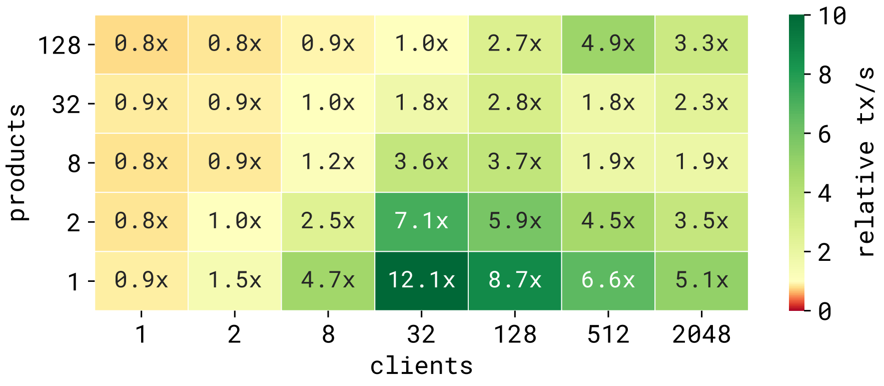


Figure 32: Throughput ratio between MRV and native, using the microbenchmark with MySQL Group Replication.

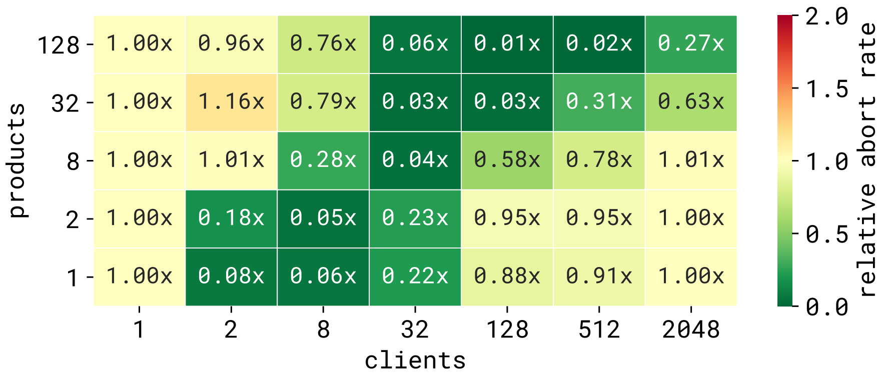


Figure 33: Abort rate ratio between MRV and native, using the microbenchmark with MySQL Group Replication.

Looking now at the MySQL Group Replication results, we can again see a significant increase in throughput comparatively to the native (Figure 32), especially when we inspect the lower middle where it reaches up to 12 times, making them also practical in multi-writer distributed databases. The relatively smaller – when comparing to PostgreSQL – but still high increase in throughput for a high number of clients can be explained by the fact that three database nodes are sharing the same storage. One thing to note is the fact that the MRV abort rate shows the biggest improvements when looking at the bottom left to top right diagonal and not necessarily when there is a high collision probability (bottom right corner). This could be justified by the fact that transactions require a more expensive certification method. To achieve similar results in a multi-writer cluster comparatively to a single writer instance, the maximum number of records per node should be higher. The fact that for 2048 clients and 1 product it shows at the same time the same abort rate (Figure 33) while providing 4 times the performance could mean the MRVs also reduce the wait time for some form of locking.

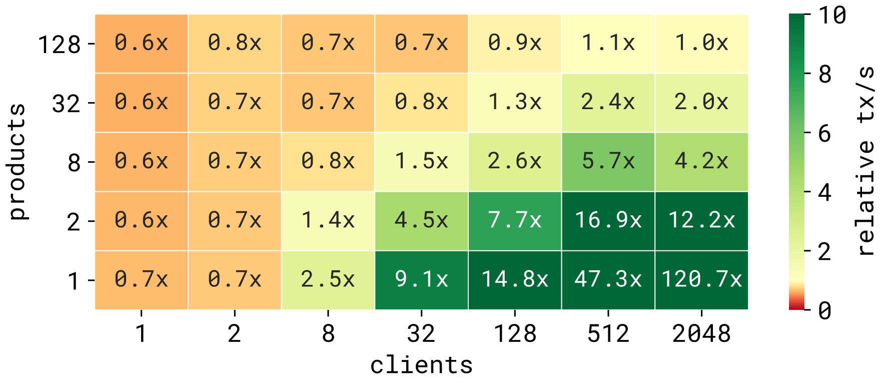


Figure 34: Throughput ratio between MRV and native, using the microbenchmark with MongoDB.

Finally, assessing the MongoDB results shows that there is a considerable reduction in the MRV throughput in the areas with lower collision probability, reaching down to 0.6 times the native performance (Figure 34). This is caused by the higher overhead of the MRV implementation in MongoDB, as seen in Figures 24 and 26. On the areas with more collisions, however, MRV presents a substantially increase in throughput, reaching more than 100 times the native performance, despite the higher overhead. These results show that MRVs are viable even for a NoSQL system, as long as the collision probability justifies its usage.

TPC-C

The next results evaluate the MRV performance on the TPC-C benchmark, using both Repeatable Read (Figure 35) and Read Committed isolation levels (Figure 36), under the single-instance architecture (PostgreSQL).

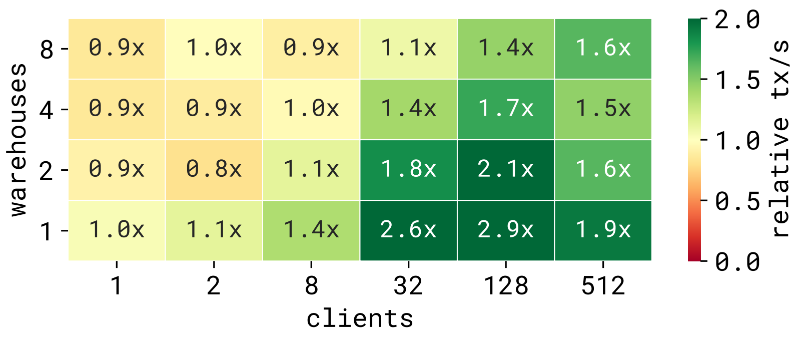


Figure 35: Throughput ratio between MRV and native, using the TPC-C with PostgreSQL (Repeatable Read).

The Repeatable Read results (Figure 35) show the MRV overhead comparatively to the native, when the number of clients is low and the number of products is high, which corresponds to a low collision probability. This makes the MRV have around 90% the throughput of the native. As the number of clients increases, so does the throughput ratio, which stays around 1.5-2 and even reaches 3, even though updating numerical values is not the only task this benchmark has to perform. Although not displayed, both abort rate and response time ratios present overall improvements, being between 1 and 0.3, and between 1.2 and 0.3, respectively.

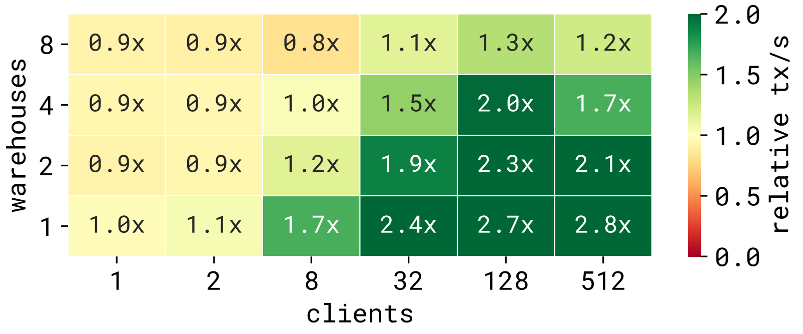


Figure 36: Throughput ratio between MRV and native, using the TPC-C with PostgreSQL (Read Committed).

The Read Committed results (Figure 36) present a similar trend to the alternative isolation. However, they surprisingly show the same or even higher MRV-to-native throughput ratio when comparing to the Repeatable Read results, even though there are no generated aborts by the fields modeled with MRVs. The reason for this is that transactions must acquire a row lock in this isolation level when updating, causing a greater negative impact on the native tests (higher contention). In addition, the wait time is higher than in the microbenchmark, since there are more operations after the numeric value update. On the other hand, the transactions in the Repeatable Read isolation level abort when simultaneously updating the same value, ceasing the need for clients to wait for a high contention lock. They instead continue and execute another transaction, which could end up committing, leading to a higher throughput[9]. The MRV response time ratio comparatively to the native in this isolation is situated between 1.2 and 0.3.

In summary, both isolation levels benefit from MRVs even if updating numeric values is not the workload's main concern, as long as the collision probability justifies its overhead.

STAMP Vacation

Finally, it is presented the STAMP Vacation results. Just like the TPC-C, this benchmark ran on the single instance architecture, using the Repeatable Read isolation level (Figures 37 and 38).

9 In the `sysbench-tpcc` benchmark, transactions do not restart the same operation with the same parameters, as they are randomly generated each time.

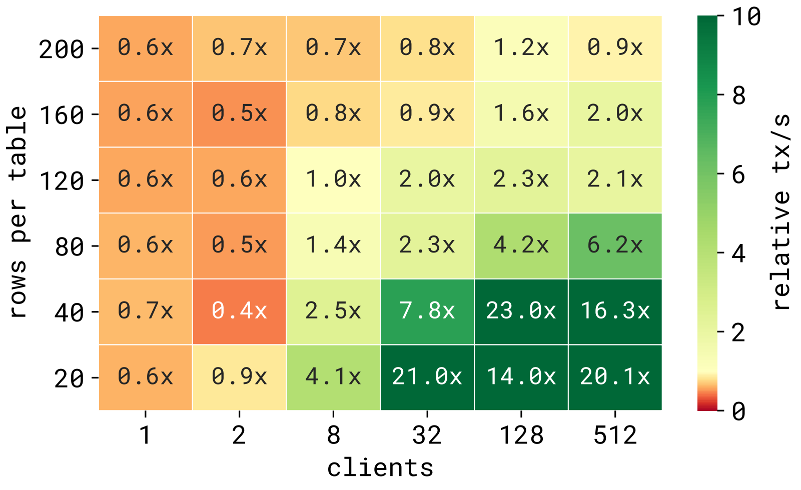


Figure 37: Throughput ratio between MRV and native, using STAMP Vacation with PostgreSQL.

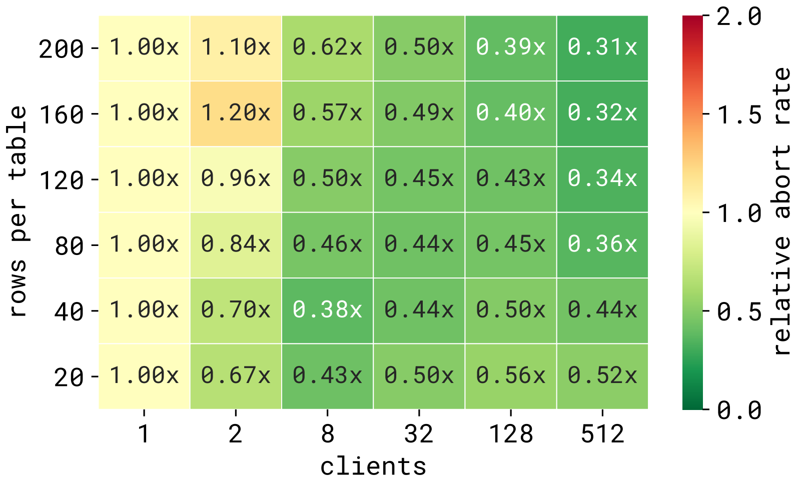


Figure 38: Abort rate ratio between MRV and native, using STAMP Vacation with PostgreSQL.

These results show a strong positive correlation between a high number of clients/low rows per table and high throughput ratio, where the MRV reaches up to 24 times the transactions per second of the native alternative (Figure 37). However, the MRV overhead is relatively higher compared to the other tests, reaching on average half the throughput of the native for a low number of clients. The cause for this is that there are many subtract operations and just few add ones, and since the overhead is higher in the former the overall performance loss is more noticeable. In addition, there is also a relatively high number of materializations of MRVs' total value, namely when the benchmark needs to know if some item is in stock before updating it (e.g. `WHERE numFree > 0`). An implementation at database engine level could cease the need to iterate over an MRV's entire set, by stopping as soon as the necessary amount has been found. However, this generic implementation is not capable of such optimizations. With that said, MRVs still present clear advantages for this benchmark when dealing with situations with high collision probability. The MRV abort rate ratio comparatively to the native is mostly below one, averaging about 0.5 (Figure 38).

Summary

A summary of the results previously presented is displayed in Table 8, where one can find the minimum and maximum throughput, abort and response time ratios between the MRV and native.

Table 8: Minimum and maximum throughput, abort rate and response time ratios between MRV and native.

		throughput		abort		resp. time	
		min	max	min	max	min	max
Microbench	PostgreSQL [RR]	0.72	20.37	0.01	1	0.78	89.4
	PostgreSQL [RC]	0.68	14.78	-	-	0.07	1.48
	MySQL GR	0.79	12.1	0.01	1.27	0.13	2.88
	MongoDB	0.61	120.7	0.02	84.43	0.7	1.65
TPC-C	PostgreSQL [RR]	0.82	3.02	0.31	1	0.33	1.22
	PostgreSQL [RC]	0.81	2.77	-	-	0.34	1.23
STAMP Vacation		0.45	24.11	0.31	1.29	-	-

5 CONCLUSIONS AND FUTURE WORK

Strong consistency, although a necessity in many cases, limits the performance of database systems, specifically geo-distributed ones. The two novel solutions presented in this work address exactly that limitation.

The *Primary Semi-Primary* architecture reduces latency and improves scalability of both reads and writes of strongly consistent transactions, especially in the context of geo-distributed deployments. It accomplishes this by using the following techniques: relying on full replication to reduce latency and avoid expensive distributed transactions, unlike systems that rely on partitioning to scale computation; a central certification to avoid distributed synchronization; and independently evolving snapshots to allow asynchronous replication, that results in reduced response time while still enabling a client to read its most recent modifications. Although it relies on a central certification that requires failover, results show that a single *primary* core can comfortably handle up to 64 concurrently executing client threads with the write-heavy TPC-C benchmark, demonstrating its efficiency. By exploiting the underlying database isolation to implement a distributed one, it simplifies implementation and possibly reduces overhead, since it relies on an efficiently and continuously updated foundation. The developed design shows a better scalability when compared to other multi-instance architectures. For similar local deployments, the *Primary Semi-Primary* reaches up to 3.8 times the native throughput, while the *Primary-Standby* reaches 1.5, the *Multi-Primary Replicas* 2.6 and the *Multi-Primary Shards* 2.8. For a geo-distributed deployment, each client in the *Primary Semi-Primary* presents, on average, 7.6 times the native client throughput for each zone, while in the *Multi-Primary Replica* it reaches 5.9 times.

Multi-Record Values (MRVs) address a classical problem in transactional data processing: Reducing the impact of update hotspots in the performance and efficiency of high throughput systems. By splitting numeric values into multiple physical records, it allows the concurrent, mostly conflict-free execution of updates, leading to reduced abort rate and response time, and consequently higher throughput. MRVs make this possible by exploiting the commutative property of addition and subtraction operations. In contrast to some alternatives, MRVs also enable modelling fields that require strong consistency guarantees, such as stocks or balances, since they ensure limit invariants (e.g., the stock of some product can't be negative), all without materializing the aggregate value, which would defeat the purpose of partitioning. Furthermore, MRVs also address the challenge of generating and managing partitions for each item, not requiring manual intervention. By not depending on locking or additional coordination, it is suitable to the novel generation of distributed and scalable transactional systems based on Snapshot Isolation [11, 24, 36, 95, 109]. The management of partitions in MRVs is inspired by consistent hashing and judiciously designed to be layered on an transactional system. It thus can be applied in existing systems at the application level or, as demonstrated with an implementation in PostgreSQL, made mostly transparent to the application by using database views and rules. It is also designed to exploit typical indexing structures and avoid deadlock probability when more than one partition has to be used. The proposal is evaluated experimentally with multiple implementations, database management systems, and benchmarks. This has shown that MRVs are beneficial even on a state-of-the-art traditional centralized database system such as PostgreSQL, where it achieved up to 20×, 3× and 24× more than the native throughput on the microbenchmark, TPC-C and STAMP Vacation, respectively. With MySQL Group Replication, it was shown that the approach is also beneficial in a distributed setting that relies on certification, reaching up to 12× the native throughput in the microbenchmark. Finally, with MongoDB, it was shown that it is also useful in transactional distributed NoSQL systems, reaching more than 100× the native throughput in the microbenchmark. The main limitation is that MRVs cannot be indexed, as this would force the materialization of the value and generate conflicts when updating the index. Moreover, MRVs should not be applied when the underlying value is zero or close to zero most of time, as they won't be able to be properly distributed them over multiple records.

The results obtained throughout this work show that both solutions can be applied to real world applications, where they can significantly improve transactional performance. This thus proves that it is worth to explore, in the future, an implementation more suitable for production, relying on code closer to the database engine instead of middleware/SQL queries, reducing the overall overhead. That implementation could combine both *Primary Semi-Primary* and MRV contributions, since they both tackle the problem of high-performance consistency in distributed databases. In addition, a future implementation can also consider exploring different decisions than the ones presented here. For example, both the *semi-primary* and *primary* prototypes were implemented using a single database engine. However, since for very large databases this might not be feasible, one can consider the usage of multiple instances per node, each with a partition of the data, to further increase storage capabilities and possibility allow higher throughput. This would still avoid the problems of geo-distributed synchronization methods, since every *semi-primary* would still have the entire copy of the database. A future implementation can also take advantage of multiple *semi-primaries* to distribute read-only queries that handle large amounts of data, reducing response time for those cases. As the MRV architecture also shows it can be applied to reduce not only aborts but also lock contention, it is also worth exploring the usage of a transaction's response time to adjust the number of nodes, since the current implementation only accounts the number of aborts. In addition, further optimization of operations that is not possible in a generic SQL implementation can be possible in a low level one, such as avoiding complete materialization for conditions (e.g. `value > 0`), being useful to cases such as the ones presented in the STAMP Vacation benchmark.

BIBLIOGRAPHY

[1] 2ndQuadrant. 2019. *Postgres-BDR*. Retrieved 2019-12-01 from https://www.2ndquadrant.com/en/resources/postgres-bdr-2ndquadrant/

[2] Daniel J. Abadi. 2019. *Demystifying Database Systems, Part 4: Isolation levels vs. Consistency levels*. Retrieved 2020-10-17 from https://fauna.com/blog/demystifying-database-systems-part-4-isolation-levels-vs-consistency-levels

[3] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. 1995. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD Record* 24, 2 (1995), 23–34.

[4] Atul Adya and Barbara Liskov. 1997. Lazy consistency using loosely synchronized clocks. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*. 73–82.

[5] Divyakant Agrawal, Arthur J Bernstein, Pankaj Gupta, and Soumitra Sengupta. 1987. Distributed optimistic concurrency control with reduced rollback. *Distributed Computing* 2, 1 (1987), 45–59.

[6] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. 2016. Cure: Strong semantics meets high availability and low latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 405–414.

[7] Amazon. 2019. *Aurora User Guide: Working with Aurora Multi-Master Clusters*. Retrieved 2020-19-10 from https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-multi-master.html

[8] Amazon. 2019. *Aurora User Guide: Working with Aurora Multi-Master Clusters - Limitations of Multi-Master Clusters*. Retrieved 2019-11-30 from https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-multi-master.html#aurora-multi-master-limitations

[9] Yair Amir, Louise E Moser, Peter M Melliar-Smith, Deborah A Agarwal, and Paul Ciarfella. 1995. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems (TOCS)* 13, 4 (1995), 311–342.

[10] J.C. Anderson, J. Lehnardt, and N. Slater. 2010. *CouchDB: The Definitive Guide: Time to Relax*. O'Reilly Media. https://books.google.pt/books?id=G4N-DPk9R5sC

[11] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. 2013. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. IEEE, 163–172.

[12] Michael Armbrust et al. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. ACM, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[13] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2013. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment* 7, 3 (2013), 181–192.

[14] Catalonia-Spain Barcelona. 2008. Mencius: building efficient replicated state machines for WANs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*.

[15] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data*. 221–230. https://doi.org/10.1145/3183713.3190662

[16] Hal Berenson, Philip Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *Sigmod Record*, Vol. 24. 1–10. https://doi.org/10.1145/568271.223785

[17] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.

[18] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1986. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[19] Philip A Bernstein, Colin W Reid, and Sudipto Das. 2011. Hyder-A Transactional Record Manager for Shared Flash.. In *CIDR*, Vol. 11. 9–20.

[20] Cihan Biyikoglu. 2018. Under the hood: Redis crdts (conflict-free replicated data types).

[21] Eric Brewer. 2000. Towards robust distributed systems. In *PODC*. 7. https://doi.org/10.1145/343477.343502

[22] Bucardo. 2019. *Bucardo*. Retrieved 2019-12-01 from https://bucardo.org/Bucardo/

[23] Michael J Carey and Waleed A Muhanna. 1986. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computer Systems (TOCS)* 4, 4 (1986), 338–378.

[24] Prima Chairunnanda, Khuzaima Daudjee, and M Tamer Özsu. 2014. ConfluxDB: Multi-master replication for partitioned snapshot isolation databases. *Proceedings of the VLDB Endowment* 7, 11 (2014), 947–958.

[25] Lei Chang, Zhanwei Wang, Tao Ma, Lirong Jian, Lili Ma, Alon Goldshuv, Luke Lonergan, Jeffrey Cohen, Caleb Welton, Gavin Sherry, and Milind Bhandarkar. 2014. HAWQ: A massively parallel processing SQL engine in Hadoop. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (06 2014). https://doi.org/10.1145/2588555.2595636

[26] Fábio Coelho, Francisco Cruz, Ricardo Vilaça, José Pereira, and Rui Oliveira. 2014. PH1: A transactional middleware for NoSQL. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, Vol. 2014. https://doi.org/10.1109/SRDS.2014.23

[27] James Corbett et al. 2013. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)* 31 (08 2013). https://doi.org/10.1145/2491245

[28] James Cowling and Barbara Liskov. 2012. Granola: low-overhead distributed transaction coordination. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*. 223–235.

[29] Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. 2016. Repairing conflicts among MVCC transactions. *arXiv preprint arXiv:1603.00542* (2016).

[30] Citus Data. 2019. *Citus*. Retrieved 2019-11-30 from https://www.citusdata.com/

[31] Citus Data. 2020. *Citus 9.4 Documentation - Citus MX*. Retrieved 2020-08-23 from https://docs.citusdata.com/en/stable/arch/mx.html

[32] Citus Data. 2020. *Citus 9.4 Documentation - Cluster Management: Adding a coordinator*. Retrieved 2020-08-23 from http://docs.citusdata.com/en/v9.4/get_started/concepts.html

[33] Citus Data. 2020. *Citus 9.4 Documentation - Concepts*. Retrieved 2020-08-23 from http://docs.citusdata.com/en/v9.4/get_started/concepts.html

[34] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Operating Systems Review - SIGOPS*, Vol. 41. 205–220. https://doi.org/10.1145/1294261.1294281

[35] Dremio. 2019. *Dremio - The Data Lake Engine*. Retrieved 2019-12-05 from https://www.dremio.com/

[36] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. 2013. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. IEEE, 173–184.

[37] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–13.

[38] Clarence A Ellis and Simon J Gibbs. 1989. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*. 399–407.

[39] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. 2005. Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. IEEE, 73–84.

[40] Hua Fan and Wojciech Golab. 2019. Ocean vista: gossip-based visibility control for speedy geo-distributed transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1471–1484.

[41] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)* 30, 2 (2005), 492–528.

[42] Daniel Ferro, Flavio Junqueira, Ivan Kelly, Benjamin Reed, and Maysam Yabandeh. 2014. Omid: Lock-free transactional support for distributed data stores. In *Proceedings - International Conference on Data Engineering*. 676–687. https://doi.org/10.1109/ICDE.2014.6816691

[43] Tom ZJ Fu, Jianbing Ding, Richard TB Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2015. DRS: dynamic resource scheduling for real-time analyt-

ics over fast streams. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE, 411–420.

[44] Google. 2019. *BigQuery Docs - Standard SQL*. Retrieved 2019-12-05 from https://cloud.google.com/bigquery/docs/reference/standard-sql/

[45] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[46] PgPool Global Development Group. 2020. *pgpool-II 4.1.2 Documentation - 3.3.2. Running mode of Pgpool-II*. Retrieved 2020-08-23 from https://www.pgpool.net/docs/pgpool-II-4.1.2/en/html/configuring-pgpool.html

[47] PgPool Global Development Group. 2020. *pgpool-II 4.1.2 Documentation - 5.7.4. Load Balancing Settings*. Retrieved 2020-08-23 from https://www.pgpool.net/docs/pgpool-II-4.1.2/en/html/runtime-config-load-balancing.html#GUC-DISABLE-LOAD-BALANCE-ON-WRITE

[48] PgPool Global Development Group. 2020. *Pgpool-II Wiki*. Retrieved 2020-08-23 from https://www.pgpool.net/mediawiki/index.php/Main_Page

[49] The PostgreSQL Global Development Group. 2019. *Postgresql 12 Documentation - 13.2. Transaction Isolation*. Retrieved 2020-06-23 from https://www.postgresql.org/docs/12/transaction-iso.html

[50] The PostgreSQL Global Development Group. 2019. *Postgresql 12 Documentation - 19.6. Replication*. Retrieved 2020-10-31 from https://www.postgresql.org/docs/12/runtime-config-replication.html

[51] The PostgreSQL Global Development Group. 2019. *Postgresql 12 Documentation - 26.5. Hot Standby*. Retrieved 2020-09-11 from https://www.postgresql.org/docs/12/hot-standby.html

[52] The PostgreSQL Global Development Group. 2019. *Postgresql 12 Documentation - 29.3. Asynchronous Commit*. Retrieved 2020-08-04 from https://www.postgresql.org/docs/12/wal-async-commit.html

[53] The PostgreSQL Global Development Group. 2019. *Postgresql 12 Documentation - CREATE RULE*. Retrieved 2020-04-29 from https://www.postgresql.org/docs/12/sql-createrule.html

[54] The PostgreSQL Global Development Group. 2019. *Postgresql 12 Documentation - SQL Commands: VACUUM*. Retrieved 2020-10-18 from https://www.postgresql.org/docs/12/sql-vacuum.html

[55] The PostgreSQL Global Development Group. 2020. *PostgreSQL Wiki - Serializable Snapshot Isolation*. Retrieved 2020-10-21 from https://wiki.postgresql.org/wiki/SSI

[56] Rachid Guerraoui and André Schiper. 1996. Fault-tolerance by replication in distributed systems. In *International conference on reliable software technologies*. Springer, 38–57.

[57] James Hamilton. 2009. *The Cost of Latency*. Retrieved 2020-10-26 from https://perspectives.mvdirona.com/2009/10/the-cost-of-latency/

[58] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for optimism in contended main-memory multicore transactions. *Proceedings of the VLDB Endowment* 13, 5 (2020), 629–642.

[59] AI Impacts. 2017. *Trends in the cost of computing*. Retrieved 2020-10-31 from https://aiimpacts.org/trends-in-the-cost-of-computing/

[60] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*. Association for Computing Machinery. https://doi.org/10.1145/258533.258660

[61] Marcel Kornacker et al. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*.

[62] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[63] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[64] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[65] Leslie Lamport. 2006. Fast paxos. *Distributed Computing* 19, 2 (2006), 79–103.

[66] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. 2011. High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment* 5, 4 (2011), 298–309.

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

[68] Mihai Letia, Nuno Preguiça, and Marc Shapiro. 2010. Consistency without concurrency control in large, dynamic systems. *Operating Systems Review* 44 (04 2010), 29–34. https://doi.org/10.1145/1773912.1773921

[69] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 265–278.

[70] Barbara Liskov and Rivka Ladin. 1986. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*. 29–39.

[71] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2011. 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*. 401–416.

[72] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2013. Stronger semantics for low-latency geo-replicated storage. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 313–328.

[73] Pedro Lopes, João Sousa, Valter Balegas, Carla Ferreira, Sérgio Duarte, Annette Bieniusa, Rodrigo Rodrigues, and Nuno M. Preguiça. 2019. Antidote SQL: Relaxed When Possible, Strict When Necessary. *CoRR* abs/1902.03576 (2019). arXiv:1902.03576 http://arxiv.org/abs/1902.03576

[74] John C. McCallum. 2020. *Disk Drive Prices 1955+*. Retrieved 2020-10-31 from https://jcmit.net/diskprice.htm

[75] John C. McCallum. 2020. *Flash Memory and SSD Prices*. Retrieved 2020-10-31 from https://jcmit.net/flashprice.htm

[76] John C. McCallum. 2020. *Memory Prices 1957+*. Retrieved 2020-10-31 from https://jcmit.net/memoryprice.htm

[77] Microsoft. 2017. *SQL Server Docs - Disaster recovery for SQL Server*. Retrieved 2020-09-11 from https://docs.microsoft.com/en-us/sql/database-engine/sql-server-business-continuity-dr?view=sql-server-ver15

[78] Microsoft. 2019. *Microsoft Azure CosmosDB Documentation - Consistency levels in Azure Cosmos DB*. Retrieved 2019-12-03 from https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels

[79] Microsoft. 2019. *SQL Server Docs - Merge Replication*. Retrieved 2019-12-03 from https://docs.microsoft.com/en-us/sql/relational-databases/replication/merge/merge-replication?view=sql-server-ver15

[80] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*. IEEE, 35–46.

[81] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.

[82] MongoDB. 2019. *MongoDB 4.2 Documentation - Replication*. Retrieved 2020-10-19 from https://docs.mongodb.com/v4.2/replication/

[83] MongoDB. 2019. *MongoDB 4.2 Documentation - Sharding*. Retrieved 2020-10-19 from https://docs.mongodb.com/v4.2/sharding/

[84] MongoDB. 2019. *MongoDB 4.2 Documentation - Transactions*. Retrieved 2019-12-03 from https://docs.mongodb.com/manual/core/transactions/

[85] MongoDB. 2019. *MongoDB 4.2 Documentation - Tutorial: Store a JavaScript Function on the Server*. Retrieved 2020-06-24 from https://docs.mongodb.com/v4.2/tutorial/store-javascript-function-on-server/

[86] Henrique Moniz, João Leitão, Ricardo J Dias, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2017. Blotter: Low latency transactions for geo-replicated storage. In *Proceedings of the 26th International Conference on World Wide Web*. 263–272.

[87] Patrick E O'Neil. 1986. The escrow transactional method. *ACM Transactions on Database Systems (TODS)* 11, 4 (1986), 405–430.

[88] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 305–319.

[89] Oracle. 2018. *MySQL 8.0 Documentation - 24.5.3 Updatable and Insertable Views*. Retrieved 2020-06-29 from https://dev.mysql.com/doc/refman/8.0/en/view-updatability.html

[90] Oracle. 2019. *Mysql 8.0 Reference Manual - 8.1.1.1 Primary-Secondary Replication*. Retrieved 2020-09-11 from https://dev.mysql.com/doc/refman/8.0/en/group-replication-primary-secondary-replication.html

[91] Oracle. 2019. *Mysql 8.0 Reference Manual - Chapter 17 Replication*. Retrieved 2020-10-19 from https://dev.mysql.com/doc/refman/8.0/en/replication.html

[92] Oracle. 2019. *Mysql 8.0 Reference Manual - Group Replication*. Retrieved 2019-12-01 from https://dev.mysql.com/doc/refman/8.0/en/group-replication.html

[93] Oracle. 2019. *Oracle 19 Concepts and Administration - Introduction to Oracle Data Guard*. Retrieved 2020-09-11 from https://docs.oracle.com/en/database/oracle/oracle-database/19/sbydb/introduction-to-oracle-data-guard-concepts.html

[94] Codership Oy. 2019. *Galera Cluster*. Retrieved 2019-12-01 from https://galeracluster.com/

[95] Vinit Padhye and Anand Tripathi. 2012. Causally coordinated snapshot isolation for geographically replicated data. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*. IEEE, 261–266.

[96] Percona. 2019. *Percona XtraDB Cluster*. Retrieved 2019-12-01 from https://www.percona.com/software/mysql-database/percona-xtradb-cluster

[97] José Pereira and Ana Nunes. 2013. Improving transaction abort rates without compromising throughput through judicious scheduling. In *Proceedings of the ACM Symposium on Applied Computing*. ACM, 493–494.

[98] Dusan Petkovic. 2017. JSON Integration in Relational Database Systems. *International Journal of Computer Applications* 168 (06 2017), 14–19. https://doi.org/10.5120/ijca2017914389

[99] David Patrick Reed. 1978. *Naming and synchronization in a decentralized computer system*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[100] Colin Reid, Philip A Bernstein, Ming Wu, and Xinhao Yuan. 2011. Optimistic concurrency control by melding trees. *Proceedings of the VLDB Endowment* 4, 11 (2011).

[101] Kun Ren, Dennis Li, and Daniel J Abadi. 2019. SLOG: serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1747–1761.

[102] Riak. 2017. *Riak 2.2.3 Documentation - Strong Consistency*. Retrieved 2019-12-03 from https://docs.riak.com/riak/kv/2.2.3/developing/app-guide/strong-consistency/

[103] Riak. 2017. *Riak KV 2.2.3 Documentation - Conflict Resolution*. Retrieved 2019-12-03 from https://docs.riak.com/riak/kv/2.2.3/developing/usage/conflict-resolution/index.html

[104] Riak. 2017. *Riak KV 2.2.3 Documentation - Data Types*. Retrieved 2020-07-04 from https://docs.riak.com/riak/kv/2.2.3/learn/concepts/crdts

[105] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[106] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

[107] Dharma Shukla. 2018. *Azure Cosmos DB: Pushing the frontier of globally distributed databases*. Retrieved 2020-07-04 from https://azure.microsoft.com/en-us/blog/azure-cosmos-db-pushing-the-frontier-of-globally-distributed-databases/

[108] Dale Skeen. 1982. *A quorum-based commit protocol*. Technical Report. Cornell University.

[109] Yair Sovran, Russell Power, Marcos Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *SOSP'11 - Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 385–400. https://doi.org/10.1145/2043556.2043592

[110] Dan Stocker. 2010. *Delta Transactions*. Retrieved 2020-07-01 from https://collectiveweb.wordpress.com/2010/03/01/delta-transactions/

[111] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review* 31, 4 (2001), 149–160.

[112] Michael Stonebraker. 2012. Newsql: An alternative to nosql and old sql for new oltp apps. *Communications of the ACM. Retrieved* (2012), 07–06.

[113] Robert H Thomas. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)* 4, 2 (1979), 180–209.

[114] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel Abadi. 2012. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1–12. https://doi.org/10.1145/2213836.2213838

[115] Ashish Thusoo, Joydeep Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - A Petabyte Scale Data Warehouse Using Hadoop. In *Proceedings - International Conference on Data Engineering*. 996–1005. https://doi.org/10.1109/ICDE.2010.5447738

[116] Alexandre Verbitski et al. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052. https://doi.org/10.1145/3035918.3056101

[117] Ricardo Vilaça, Francisco Cruz, José Pereira, and Rui Oliveira. 2013. An Effective Scalable SQL Engine for NoSQL Databases. In *Distributed Applications and Interoperable Systems*, Jim Dowling and François Taïani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–168.

[118] Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (2009), 40–44.

[119] WonderNetwork. 2020. *Global Ping Statistics*. Retrieved 2020-07-24 from https://wondernetwork.com/pings

[120] Reynold Xin, Josh Rosen, Matei Zaharia, Michael Franklin, Scott Shenker, and Ion Stoica. 2012. Shark: SQL and Rich Analytics at Scale. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (11 2012). https://doi.org/10.1145/2463676.2465288

[121] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-latency transaction processing for globally-distributed data. In *Proceedings of the 2018 International Conference on Management of Data*. 231–243.

[122] Anand Yendluri, Wen-Chi Hou, and Chih-Fang Wang. 2004. Improving concurrency control in mobile databases. In *International Conference on Database Systems for Advanced Applications*. Springer, 642–655.

[123] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*. 1629–1642.

[124] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 263–278. https://doi.org/10.1145/2815400.2815404

[125] W. Zhou, Guillaume Pierre, and Chi-Hung Chi. 2011. CloudTPS: Scalable Transactions for Web Applications in the Cloud. *Services Computing, IEEE Transactions on* 5 (01 2011), 1 – 1. https://doi.org/10.1109/TSC.2011.18