A Survey of Consistency and Isolation in Replicated Databases

José Calvo-Villagrán
jcalvovi@uwaterloo.ca

ABSTRACT

Database replication is an important technique to enable higher performance. Two important features that affect performance on replicated databases are isolation and consistency. While isolation directly affects concurrent execution of transactions on the overall system, consistency affects replication management by determining how synchronization of replicas is performed. Understanding how these two features are designed on a replicated database system is important because it determines the trade-offs between application needs and performance. This survey explores existing lazily replicated database systems and categorizes them according to their isolation and consistency levels.

1. INTRODUCTION

Database replication is a popular technique for increasing availability and performance. It is used by a wide variety of systems, for different application domains. Examples of such variety can be found across industry: Yahoo!'s PNUTS [8] for geographical replication, Google's F1 [23] for scalability, Amazon's Dynamo [11] for high availability and SAP's HANNA [14] for mixed transactional (OLTP) and analytical (OLAP) workloads, to name a few.

This diversity of applications is a reflection of how complex it can be to design a replicated database system. Many issues need to be considered, and desired features for replication protocols will sometimes conflict with each other. Figure 1 is an example of a trade-off in replication. A system that offers good performance and strong isolation (e.g. serializability) is desired. Despite that, no solution has yet been devised to achieve this ideal. System designers have to decide whether to sacrifice performance in favour of high isolation, or vice versa. As the isolation level decreases, the effects that transactions have on each other while concurrently accessing common items increases.

Database consistency is another feature that affects performance on replicated environments. It refers to how synchronized are all replicated items of a database. The ultimate goal of replication protocols is to reach a state called *mutual consistency*, in which all copies of each data item have the same value [17]. Careful considerations regarding when to update replicas are taken while designing replication protocols. Systems can be found that are designed to be in a perpetual state of mutual consistency, by eagerly updating all replicas of an item. These systems will have diminished performance because transactions have to wait for all replicas to apply the updates before committing. On the contrary, lazily replicated systems can relax consistency constraints and perform better. Choices on when and how to update replicas still have to be made in lazy systems, as this can affect communication between replicas and have an effect on performance.

The relevance of these two features regarding performance is clearly exemplified by eventually consistent key-value data stores. Systems like Amazon's Dynamo [11], Facebook's Cassandra [16] and LinkedIn's Project Voldemort [25] are explicitly designed to be highly performant and tolerant to network and hardware faults. Such goals are achieved by sacrificing transactional support (no isolation) and consistency guarantees (eventual consistency). These systems are often considered as alternatives to more "traditional" replicated relational database systems, like PostgreSQL [20], SQL Server [22] and MySQL [21] which, at their strictest levels, can offer serializable isolation and mutual consistency. However, other alternatives have been proposed by both academia and industry. By working with different isolation levels and offering more relaxed consistency guarantees, system designers can find a proper balance between application needs and performance.

This paper is a survey on existing lazily replicated database systems. A generalized architecture of the systems considered for this survey is depicted in Figure 2. Clients can send their transactions to either replica (also known as secondary) sites or to the master (also known as primary) site. All update transactions are executed first at the master, which determines an ordering of transactions and then propagates updates to all replicas. Replica sites can execute read-only transactions if required. Variations to this architecture are also considered: multiple master sites [7], multi-purpose sites (a site can be master to some items and replica to others) [11] and introduction of other components (shared disks, message brokers, etc.) [8]. A common set of features can be used to evaluate systems with similar architectures after these have been surveyed. For this survey, the features being considered are isolation and consistency because of their direct effect on performance.

The rest of this survey is organized as follows. Section 2 introduces the features used to compare between systems. It also presents a categorization for each feature. Section 3 briefly introduces each of the systems and categorizes them accordingly. It also shows the isolation/consistency grid that

serves as a quick visual aid to compare systems, as well as a discussion of the findings in this survey. Finally, Section 4 concludes this survey.

2. FEATURES

This section introduces all the features that are being considered for the survey. The criteria to choose the following features is because they are commonly found within lazily replicated databases and because of their significant effect on performance.

2.1 Architectures

The architecture of a system refers to where updates are first performed. This survey considers only lazy master replication techniques, in which updates are first executed on a master copy of an item and then propagated to replicas. There are two lazy master architectures that can be used by replication protocols:

Single master: There is only one master database in the system. It executes all updating transactions and later updates replicas.

Multi-master: There is more than one master database in the system. Each database item has only one master, and data is partitioned between them. Updating transactions will generally require some level of coordination among master sites, which later have to update replicas.

Multi-master systems are generally considered to perform better, especially if data can be partitioned in a way such that the amount of transactions that execute in a distributed fashion is minimized. If distributed transactions (transactions that execute at more than one master) do not happen, performance will generally grow linearly with respect to the number of master databases in the system (assuming that the load is balanced across servers). Otherwise, it grows sublinearly [9].

2.2 Consistency Levels

Databases that hold replicas can be categorized based on the consistency guarantees that they provide. This survey adopts the consistency categorization proposed by Zhuge et al. [26], which defines four different levels:

Convergence: Only the final states[1] on both the master and replica sites are guaranteed to be the same, once the last update has executed. Eventually consistent systems are a great example of convergence.

Weak consistency: Convergence holds and every state seen on a replica reflects a valid state at each master site. This valid state is the result of the replica applying the same transactions that were committed at the master. However, the schedule of transactions executed at the replica might be different from the one that executed at the master, causing a re-ordering of transactions. This has a consequence on distributed transactions because

[1]A database state changes whenever an update is applied.

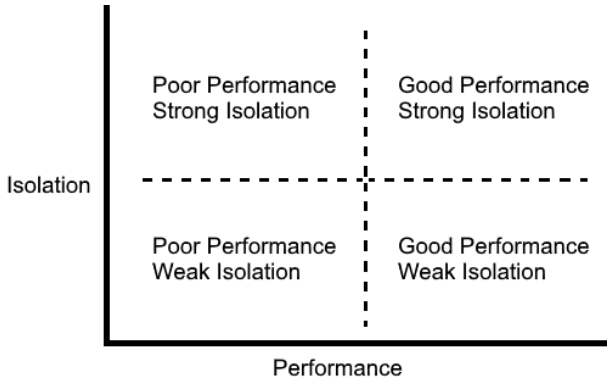


Figure 1: A trade-off between performance and isolation. It is desirable to have good performance for strong isolation levels.

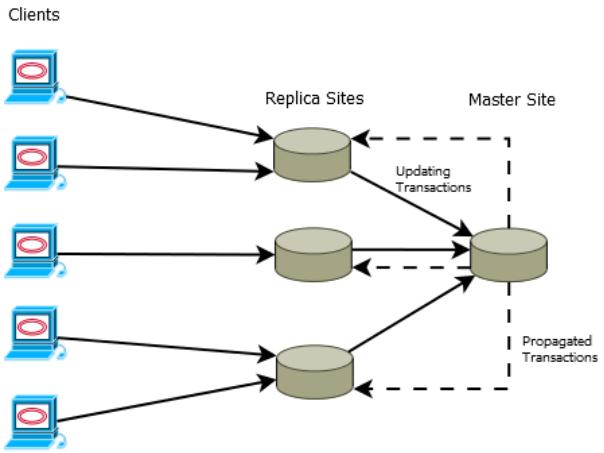


Figure 2: Generalized architecture of the systems being surveyed for this paper. Transactions can be sent to replica sites, but updating transactions must be executed at a master site. The master site later updates replicas by sending propagated transactions (represented by non-solid arrows).

they can be partially installed at the replicas. It is valid for the replicas to see the effects that distributed transactions had on a proper subset[2] of the master sites, instead of the global effects that it had on all of the sites. An example of replication under weak consistency would be the following: Server A is master of item x and Server B is master of item y. Server C holds replicas of x and y. If a distributed transaction T_d updates both x to x' and y to y', it is acceptable for C to hold any of the following states: ($\{x, y\}$, $\{x, y'\}$, $\{x', y\}$, $\{x', y'\}$).

Strong consistency: Convergence holds and every state s_j seen on a replica is the result of applying the first j transactions that executed on master sites. Furthermore, the order of replica states matches the order of states seen on master sites. This necessarily implies that the schedule of transactions used on both the master and the replica sites is the same. Also, distributed transactions are seen on replicas as only one transaction, instead of dividing them into per-site transactions. Using the same example as before, Server C can only transition from state ($\{x, y\}$) to state ($\{x', y'\}$).

Completeness: Strong consistency holds and there is one-to-one mapping between every state seen on master sites to a state seen on the replicas.

In simpler terms, the differences between consecutive levels are the following:

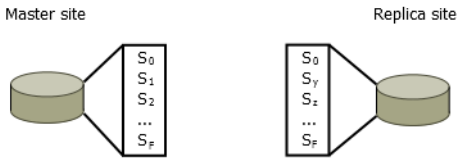
Convergence & Weak Consistency: Convergent replicas are allowed to break the atomicity of transactions and apply a subset of the updates as its own. Weakly consistent replicas are not allowed to do this. However, they can group several transactions together and apply them as one.

Weak & Strong Consistency: Strongly consistent replicas follow the same scheduling of transactions than the one executed at master sites. They also ensure that distributed transactions are not partially installed.

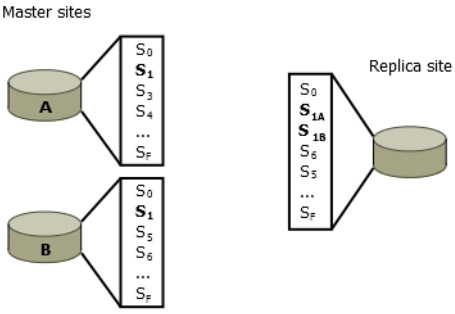
Strong Consistency & Completeness: Completeness must show every state seen at master sites, whereas strong consistency is allowed to "skip" some states by grouping transactions together.

All levels proposed by Zhuge et al. allow replicas to be non-mutually consistent with respect to master sites, a common feature of lazily replicated systems. These levels are mostly concerned about how updates are applied in replicas, instead of when. This survey uses this categorization of consistency because it further explores some of the complexities of designing lazy replication protocols. Under this categorization, performance increases as the consistency level decreases, completeness being the highest consistency level.

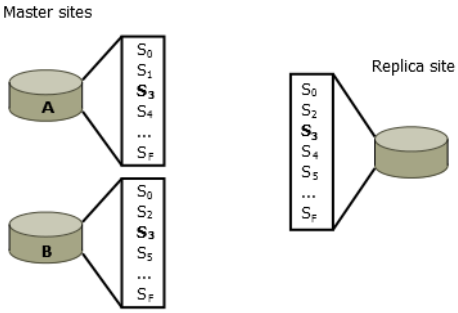
[2]A proper subset S' of S is a subset that necessarily excludes at least one member of S.



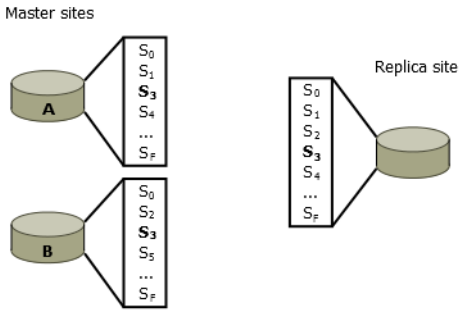
(a) **Convergence:** Given the same initial state (S_0), the replica site only guarantees that the final state (S_F) will be the same on both sites. All intermediary states (S_1, S_2,...) might never be seen on the replica.



(b) **Weak Consistency:** A distributed transaction (represented by state $\mathbf{S}_1$) can be applied partially at the replica ($\mathbf{S}_{1A}$ and $\mathbf{S}_{1B}$). Also, states can be re-ordered at the replicas.



(c) **Strong Consistency:** A distributed transaction (represented by $\mathbf{S}_3$) cannot be partitioned at the replica. Similar to weak consistency, states can be grouped together (S_2 includes the updates installed by S_1).



(d) **Completeness:** Every state seen on master sites is seen on the replica.

Figure 3: A graphical representation for each of the consistency levels used for this survey.

2.3 Isolation Levels

The ANSI SQL standard [1] defines four levels of isolation: Serializability, Repeatable Read, Read Committed and Read Uncommitted. Each level is described in terms of *phenomena*, a sequence of operations that can lead to non-serializable executions. Explanations of each isolation level and phenomena can be found in [2].

Despite being a standard, the four level model is often not fully implemented in modern DBMS. PostgreSQL, as an example, offers three levels of isolation: serializable snapshot isolation [5], snapshot isolation and read committed. The popularity of Multi-Version Concurrency Control (MVCC) systems has directed the attention of both academia and industry towards other alternatives to the four level model. Specifically, Snapshot Isolation (SI) has attracted a lot of attention. This survey focuses on three isolation levels: Serializability, Snapshot Isolation and Read Committed. In the same fashion as consistency, performance is expected to increase as the isolation level decreases.

2.4 Scope

Scope refers to the extent to which an isolation level will be guaranteed within all sites of a replicated database. A **global scope** of an isolation level means that all sites within a replication group of a database will guarantee a unified view of the database, at least on a session[3] level [10]. A client who executes several transactions on a distributed database will perceive the database as a single site acting on a given isolation level, regardless of which site the transaction is sent to. Systems that guarantee one-copy serializability (1SR) [17] are an example of being globally scoped.

On the contrary, a **local scope** of an isolation level means that each site guarantees a given isolation level, but SQL phenomena can still happen between transactions executing at different sites. A common behaviour on local scoped systems is to allow transaction inversions [17]. A transaction inversion can be explained with an example: a client executes an updating transaction T_i that modifies item x. After T_i is committed at a master site, the same client executes a different transaction T_j that reads x. This transaction is processed by a replica site that is not up-to-date with respect to T_i. The client cannot see her own updates, regardless of the isolation level she chose to execute her transactions. The problem with transaction inversions is particularly exacerbated if T_j modifies a different data item, say y, that depends on the value of x.

Finally, **global update scope** is a middle ground between the two. Updating transactions have a global scope on the database, i.e. there are no transaction inversions between them. However, read-only transactions are allowed to work on stale versions of the data and it is possible for a client to not read her own updates.

Local scope is the option that provides better performance. Enforcing global scopes might incur some communication costs to coordinate replicas. Also, it is possible that transactions have to wait until replicas are updated before starting their execution. Please note that single master architectures can only work under two different scopes: global and global update.

[3]A session is a grouping of transactions originating from the same client.

2.5 Update Propagation

The propagation attribute defines when a transaction is propagated to replicas. This survey adopts the convention from [18] which focuses on two types of propagation strategies:

Deferred: A transaction is sent to replicas after the local transaction has committed.

Immediate: A transaction is sent to replicas without waiting for the local transaction to commit. This usually requires to also propagate the result of the local transaction (either abort or commit).

Deferred propagation can provide better performance over immediate propagation, especially in the presence of a large amount of aborted transactions. It also results in fewer messages passed between sites. On the other hand, immediate propagation is sometimes desired because it improves data freshness [18].

3. SYSTEMS

The categorization of systems with respect to the features introduced in Section 2 is presented. This survey is organized as follows: each system is briefly introduced, providing the main ideas proposed by it. Then, a justification for each of the features is provided. Table 1 shows the classification of systems according to consistency, scope, architecture and isolation criteria. Since isolation is an important feature for this survey, systems that provide no transactional support are not considered. This means that most eventually consistent key-value data stores are excluded.

3.1 ConfluxDB

ConfluxDB [7] is a multi-master replicated database system that works under SI. Its main goal is to guarantee global SI without a centralized component that imposes a global ordering of transactions. Instead, ConfluxDB coordinates distributed transactions by using an augmented version of the two-phase commit (2PC) protocol. 2PC is used to ensure the atomicity of distributed transactions [3]. In ConfluxDB, the distributed transaction is first received by one of the master sites, which is then designated as the coordinator and is in charge of starting sub-transactions at each site that holds data items needed to complete the transaction. The coordinator is also in charge of coordinating the 2PC protocol to determine if the transaction commits. However, 2PC alone is not enough to prevent transactions from seeing different snapshots at different sites.

Consider, for example, a distributed transaction T_1 that updates x at site A and updates y at site B. Concurrently, distributed transaction T_2 reads x and updates y. Now, consider the following interleaving of actions: $r_1(x)$, $w_2(x)$, $w_2(y)$, $commit_2$ and $w_1(y)$. In this scenario, T_1 needs to be aborted because of the "First-Committer Wins" rule [2]. Instead, 2PC would allow T_1 to commit because each individual site is not able to detect any conflict between the two transactions.

			Consistency							
			Completeness		Strong		Weak		Convergence	
		Scope	Single	Multi	Single	Multi	Single	Multi	Single	Multi
Isolation	Serializability	Global	Ganymed							
		GU	KuaFu		KuaFu		KuaFu		Ganymed	
		Local		Datablitz						
	Snapshot Isolation	Global		ConfluxDB						
		GU	KuaFu		KuaFu	Tashkent	KuaFu			
		Local						Walter		Walter
	Read Committed	Global	Ganymed							
		GU	KuaFu		KuaFu		KuaFu			
		Local								

Table 1: Classification of systems according to consistency, isolation, scope and architecture criteria. All the systems surveyed used a deferred propagation strategy, except for the hybrid strategy used by Datablitz (Section 3.2). GU scope stands for Global Update (Section 2.4).

The authors of ConfluxDB propose a distributed certification algorithm that allows distributed updating transactions to see consistent snapshots at every site that participates on the transaction. Their protocol, called Collaborative Global Snapshot Isolation (CGSI), is able to identify a set of conflicting concurrent transactions happening at all sites and guarantees global snapshot isolation on master sites. When a distributed transaction T_d wants to commit, it checks at every site that each transaction T_c that belongs to the conflicting concurrent set is either executing concurrently to T_d or has not yet started executing. If this is not the case, it can only mean that T_c has already committed at that site. Because T_d started executing before T_c on at least one of the participating sites (the one that reported the transaction to be concurrent), but it also started executing after T_c on another site, CGSI determines that the distributed transaction is not seeing a consistent snapshot across all sites and it needs to abort.

Continuing with the above example, CGSI allows T_2 to commit, because site A reports T_1 as concurrent and site B has not even started its execution. When T_1 tries to commit, site A reports T_2 as concurrent, but site B reports it as committed. CGSI then proceeds to abort transaction T_1.

Figure 4 shows the replication strategy employed by ConfluxDB. Each replica site holds a copy of all tables from every partition, i.e. each replica holds the entire database. The log merger component extracts log records from each of the master sites, merges them into a unified log and sends it over to the replicas. ConfluxDB guarantees a completeness consistency level by making use of distributed transactions to establish synchronization points within the participating sites. Each sub-transaction that is part of the distributed transaction stores the same global commit timestamp on the local log. When reconstructing the merged log, ConfluxDB uses the synchronization points to ensure that all locally committed transactions that happened before the distributed transaction are written to log first. This guarantees that replicas also see a consistent snapshot when applying such transactions.

Regarding scope, CGSI guarantees global isolation for updating transactions. Read-only transactions are also guaranteed to execute under SI at the replicas. However, delays caused by the log merger may cause transaction inversions. ConfluxDB solves this by offering session guarantees [6]. A session holds the timestamp of the most recently committed transaction that belongs to the session. Whenever a read-only starts executing, a replica site makes sure that it is updated with respect to the session's timestamp. This is how ConfluxDB is able to provide global scope guarantees. Finally, by using the log at master sites to replicate transactions, ConfluxDB uses a deferred update propagation strategy.

3.2 Datablitz

Datablitz [4] is a multi-master replicated database that utilizes information about placement of replicated items to update them while ensuring a completeness level of consistency. It does not support distributed transactions and works exclusively under serializability isolation. The main challenge that Datablitz is trying to overcome is to guarantee completeness consistency in a shared-nothing environment and without a centralizing component that can determine global timestamps for transactions. Figure 5 shows an example of a consistency problem that can arise under such architectures. An inversion of propagated transactions T_1 and T_2 can happen between sites S_2 and S_3 because master sites are unable to determine a global order for each transaction.

Datablitz proposes three different replication protocols that avoid inversion of propagated transactions. All three protocols require knowledge about the topology of the data distribution in order to work properly. A directed graph is used to represent such topology. Each vertex represents a site, and an edge from site S_i to site S_j is used to represent that S_j holds replicated items from S_i. If the resulting graph is acyclic (a directed acyclic graph, or DAG), two replication protocols can be used:

DAG(WT): DAG without timestamps (WT) propagates transactions along the edges of a tree. This tree is a transformation of the DAG and has the property that each site can only have one parent. The parent is the only site that is allowed to send propagated transactions to its children. This prevents transaction inversions because the parent determines an order of updating and propagating transactions, which later is

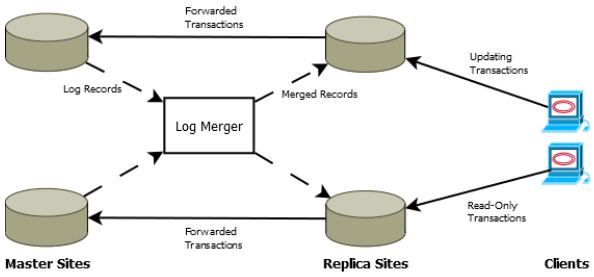


Figure 4: ConfluxDB's multi-master architecture. Primary sites send log records to a log merger component, which establishes a total ordering of transactions and merges records of distributed transactions into a single transaction. This is because each secondary site holds replicas of all partitions.

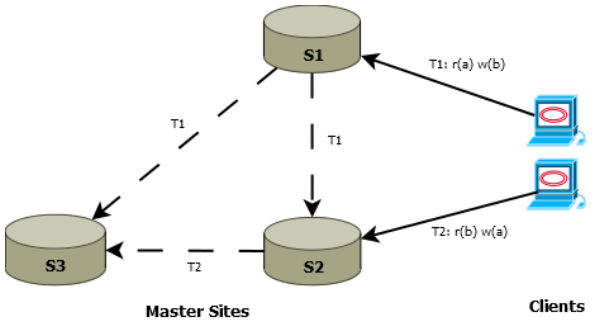


Figure 5: An example of an anomaly that Datablitz prevents. Site S_1 is the master of item b, site S_2 is master of item a and replica of item b and site S_3 holds replicas for both a and b. Once the updating transactions T_1 and T_2 commit at the master sites, they have to be propagated to all replicas. The problem arises when sites S_2 and S_3 are not coordinated and allow T_1 and T_2 to execute in a different order. This can potentially lead to replicas not reaching mutual consistency.

communicated to the children.

DAG(T): DAG with timestamps (T) employs local timestamps to impose an order on propagating transactions. Whenever a site S_i executes a propagated transaction, it stores relevant information in a vector. Later, when a transaction locally commits at S_i and needs to be propagated, the vector is passed along to share state among replicas. The combined state of all master sites that propagate towards the same replica is used to determine the order of transactions.

Continuing with the example showed in Figure 5, DAG(WT) builds the tree $S_1 \rightarrow S_2 \rightarrow S_3$ and propagation needs to follow this order. Instead, DAG(T) does not restrict communication between sites. It augments the information being shared by sites and helps site S_3 to execute transactions in the same order as S_2. The main difference between both protocols is the number of messages passed between sites. DAG(WT) incurs in a larger amount because of the restriction of having only one parent per site. A propagated transaction will need to go through several intermediary sites to reach its destination. Sites working with DAG(T), on the contrary, are aware of the location of all replica sites. Sending propagating transactions requires one message to each replica.

A third protocol is also proposed by Datablitz which imposes no restrictions about data placement, i.e. the graph can have cycles. It is called the BackEdge protocol. It is able to identify a set of edges, called backedges, that can be removed from the graph in order to break cycles. The protocol, however, does not remove backedges. Instead, they are used to identify sites that can cause transaction inversions regardless of the lazy replication protocol used. These sites have to eagerly replicate transactions in order to avoid inversions. All other edges can use either DAG(T) or DAG(WT) to propagate transactions.

All protocols yield the same result: a completeness level of consistency under serializable isolation. Every state seen on master sites is succesfully propagated to replicas, using the same serializable schedule of transactions. However, serializability is only guaranteed at a local level. Updating transactions are allowed to read stale versions of replicated items at a site and commit. Finally, the BackEdge protocol utilizes a hybrid strategy for propagating updates. An immediate strategy is used along backedges and deferred is used for all others.

3.3 KuaFu

KuaFu [15] is a single master system based on MySQL that improves data freshness at replica sites by improving on traditional log-shipping replication techniques. Log-shipping replication involves participating sites to exchange portions of transaction logs, which are later replayed at replicas. It is a common feature provided by popular DBMSs, like MySQL, PostgreSQL, Oracle and SQL Server. It is also relatively easy to implement, as DBMSs already have capabilities to manage logs for recovery purposes. Replica sites will usually replay the log serially to ensure data is correctly installed.

The problem identified by KuaFu is that of a *parallelism gap* happening between master and replica sites. This gap refers to the possibility of replicas not being able to keep up with the master's pace because of the high degree of parallelism on the master, while replicas are serially replaying the log. Having a large parallelism gap between a master and its replicas can result in an increased risk of data loss in case of a failure on the master [15].

KuaFu's basic functionality and architecture is depicted in Figure 6. Updating transactions are routed to the master site and written to a Write Set (WS) log, which records exactly what tuples were modified by a transaction and stores before and after images of each. Instead of describing the actions that transactions made at the master, WS log records only their effects. The replica, upon receiving the log, can concurrently replay the log by identifying dependencies among transactions. A transaction will depend on a previous one if the intersection of the set of tuples that each transaction modifies is not empty. The replica is exclusively dealing with write-write conflicts between transactions, since reads are never reflected on the WS log. KuaFu then proposes to concurrently execute the transactions which have no dependencies or whose dependencies have all finished executing.

KuaFu can work at three different consistency levels: completeness, strong and weak. By re-ordering transactions at the replica, it provides weak consistency guarantees. It guarantees that the final states at the master and replica are the same by serially executing dependent transactions. However, it can be configured to provide stronger consistency guarantees. It introduces the concept of a barrier, that can only be used on MVCC systems. A barrier is a grouping of transactions that, once all have committed, enables a snapshot on which read-only transactions can now execute. A barrier configuration of 10, for example, allows KuaFu to apply its dependency detection on the next 10 pending transactions on the log and execute them concurrently. Read-only transactions execute on a database snapshot that either sees all of the 10 transactions, or sees none. A barrier configuration of 1 ensures that all transactions are serialized at the replica (no concurrency). Consequently, each consistency level depends on the configuration of the barrier:

If no barrier: The strongest consistency guaranteed is weak consistency because of the re-ordering of transaction at the replica site.

If barrier = 1: The strongest consistency guaranteed is completeness because every transaction seen at the log is applied and immediately seen by read-only transactions.

If barrier > 1: The strongest consistency guaranteed is strong consistency because the database snapshot is only advanced once all transactions within a barrier are committed.

KuaFu can work at any isolation level. This is a consequence of dealing with the WS log. This log provides an abstraction from isolation anomalies within conflicting transactions because it only records the effects that the transactions had on data items. As mentioned before, this translates into a log that captures only write-write conflicts. The only phenomena that can happen when dealing with no reads is a dirty write [2]. All of the isolation levels considered for this survey guarantee that no dirty writes happen. The propagated transactions executing at the replica can do so at any isolation level and not affect the outcome of replaying the log. Regardless of the isolation level, both the master and replica sites are guaranteed to converge.

Finally, log shipping replication is a deferred update propagation strategy. Regarding the scope, KuaFu is only able to provide global update guarantees for a given isolation level since read-only transactions are allowed to read stale data.

3.4 Ganymed

Ganymed [19] is a single master system that makes use of a middleware scheduler to manage consistency among replica sites. It proposes a scheduling algorithm called Replicated Snapshot Isolation with Primary Copy (RSI-PC). The algorithm works on a set of stand-alone MVCC databases to schedule transactions that execute under different scopes of isolation guarantees. RSI-PC separates read-only and updating transactions and schedules them to different sites. Figure 7 shows, at a high level, how RSI-PC works: updating transactions are sent to the master site, while read-only transactions are sent to any of the replicas.

Once the updating transactions are committed at the master site, the scheduler extracts the transaction's Write Set (WS) and global version number. As discussed in Section 3.3, WS extraction is a deferred propagation strategy that is used to describe the effects, and not the actions, that transactions had on data items. This allows replication protocols to propagate transactions lazily under different isolation levels. The scheduler has one thread per each replica, which is in charge of applying each of the actions described in the WS as soon as they arrive from the master. Consequently, it is possible that replicas reflect a state in which a transaction is only partially installed. Therefore, Ganymed guarantees a convergent consistency level when working under this setting.

There is, however, a feature that allows Ganymed to provide a stronger consistency level. As mentioned before, the scheduler is aware of the transaction's global version number once it is committed at the master. This number is used to determine the ordering of transactions that needs to happen at replicas. It also allows the scheduler to be aware of how up-to-date is each of the replicas. A read-only transaction can establish whether it wants to work with the latest version of the data, or if it can work within a certain threshold of data staleness. The scheduler will then execute the read-only transaction at an appropiate replica. Two consequences arise from this feature:

Support for both global and global update sscopes: A read-only transaction that requires up-to-date replicas works under a global isolation scope because no transaction inversions can occur. On the contrary, transactions that execute on stale data work under a

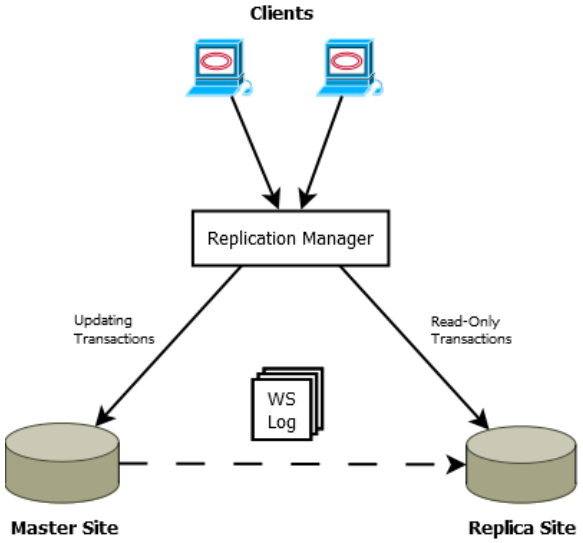


Figure 6: KuaFu's single master architecture. Updating transactions are sent to the master site. The transaction's Write Set (WS) is extracted and written to a log. The replica periodically reads log files from master and updates itself.

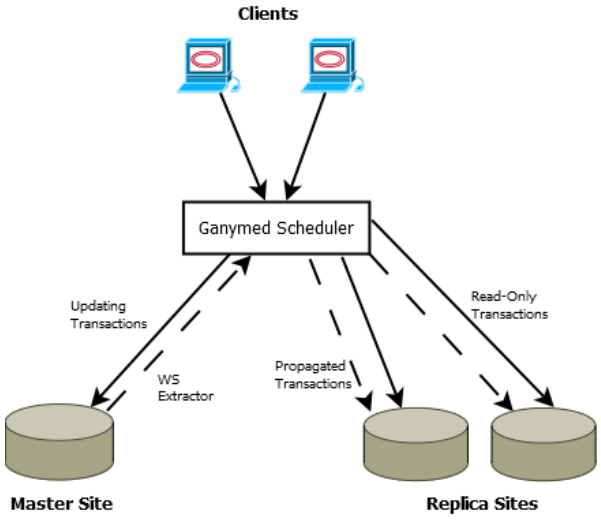


Figure 7: Ganymed's single master architecture. Transactions are sent to the scheduler, which then routes updating transactions to the master site. The master extracts the transaction's Write Set (WS) and sends it back to the scheduler so it can apply propagated transactions into replicas.

global update scope.

Guarantees for completeness consistency: Transactions working on up-to-date replicas will only see valid states of the data. These states are installed using the same schedule that was used at the master, thanks to the global version number. Furthermore, the underlying MVCC database guarantees that once the transaction starts executing, the snapshot remains unchanged regardless of concurrent updating transactions happening at the master. On the contrary, transactions that execute on stale data see a convergent level of consistency.

Finally, Ganymed is able to provide support for two different isolation levels: serializability and read committed. SI is not supported because clients connect to Ganymed through JDBC, a database connector standard used for database-independent interactions between applications[4]. JDBC, as it currently stands, uses the ANSI SQL standard for isolation levels. Read-only transactions are always run at a serializable level (no explanation is given for this). Therefore, these transactions are the only ones that can execute at replicas and see different scope and consistency levels. Read committed transactions are always globally scoped (because they can only update) and they see a completeness level of consistency.

3.5 Walter

Walter [24] is a key-value data store in a shared-nothing environment that provides transactional support. It works under Parallel Snapshot Isolation (PSI), a distributed isolation level that can be utilized by multi-master geo-replicated systems. PSI allows individual master sites to execute local transactions under Snapshot Isolation and distributed transactions are coordinated by a modified version of the 2PC protocol, as seen on Figure 8. This modification allows participant sites to share information about the snapshot that the transaction is trying to update. Each site is then able to determine if the snapshot is valid or not by applying local SI.

Unlike ConfluxDB, PSI is not able to guarantee global SI. This is because 2PC will only be enforced on updating transactions. Read-only transactions can be answered locally at each site, because the site is able to host replicated partitions, i.e. partitions for which it is not a master of. This has the consequence that data might not be up-to-date, causing transaction inversions. A variation of PSI, called PSI with counting-set objects (cset PSI), is also introduced. Csets are objects that can only be modified by commutative operators. Hence, these objects are able to commit locally any updating transaction that modifies them. The main advantage is that the operation that modifies the object is later replicated to all copies of the item and eventually all converge to the same value.

Walter implements both variations of PSI. It relies on a data structure called vector timestamp to share state between sites participating in a distributed transaction. Each master site S_i holds a vector timestamp vs_i which maps every other

[4]http://www.oracle.com/technetwork/java/javase/jdbc/index.html

master site S_j to a (monotonic) timestamp. This timestamp belongs to the last replicated transaction executed at site S_i. If $vs_i[j]$= 100, as an example, it means that site i is successfully up-to-date with respect to the 100th transaction from site j. Whenever a distributed transaction starts its execution at site S_i, this site creates a vector timestamp of all the other sites involved in the transaction and sends it over as part of the augmented 2PC protocol. Each site then corroborates that the items being modified by the transaction have not been updated since the timestamp indicated by the vector or that there are no concurrent transactions that conflict with the distributed one. If this is the case, then the distributed transaction can safely commit.

Walter can guarantee two different consistency levels: weak consistency and convergence. If Walter is configured to work with cset objects it can only guarantee convergent consistency. This is because it is possible for a cset object to be in completely different states at each site. As stated before, all sites eventually converge into a single value. Without csets, Walter is weakly consistent because the vector timestamps are used to guarantee an ordering of transactions per site. A replicated transaction T_r originated at site S_i will only commit at site S_j once $vs_i[j]$= T_r.timestamp - 1. This guarantees the same ordering of transactions at the master and replica sites. However, it does not guarantee the same global ordering of transactions because distributed transactions can be partially applied at replica sites.

Finally, Walter purposely uses a deferred propagation strategy in order to better perform at a geo-replication scale. Walter is designed to be highly available in case of network partitioning.

3.6 Tashkent

Tashkent [13] is a multi-master system that is able to implement group commits at stand-alone sites in order to make better use of persistent storage. Group commit is a known technique used to synchronously write to disk the commit records for several transactions with just one I/O operation [12]. It is a feature regularly included in modern DBMSs. However, replication protocols that provide higher levels of consistency guarantess will often overlook this feature because propagated transactions need to be written to disk serially.

The architecture for Tashkent can be seen in Figure 9. It works under a distributed isolation level called Generalized Isolation Level (GSI). GSI provides global SI for updating transactions (global update scope) and allows read-only transactions to execute without blocking. It makes use of a middleware component, called the certifier, that is in charge of ordering transactions and managing propagation to replicas. Each master site contains a copy of the entire database and transactions can execute anywhere. Sites keep track of a global version number, which is incremented by the certifier everytime an updating transaction commits. An updating transaction executes locally at a site and sends its write set and version number to the certifier to decide if it can commit or not. When the certifier determines a result for the transaction, it communicates it back to the originating site, along with the latest global version number and a batch of propagated transactions that need to be installed.

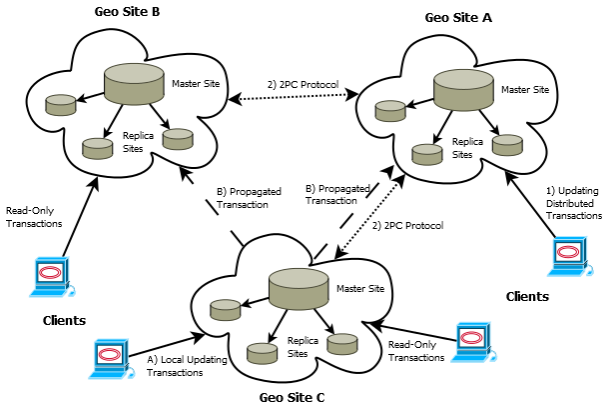


Figure 8: Walter's multi-master geo-replicated architecture. Each geo-site has master and replica sites, with replication managed by a distributed file system. Between geo-sites, replication is done by lazy replicated transactions, as the ones originating from Geo Site C (Steps A & B). A distributed transaction uses 2PC protocol to guarantee that the global snapshot that the transaction sees is valid (Steps 1 & 2).

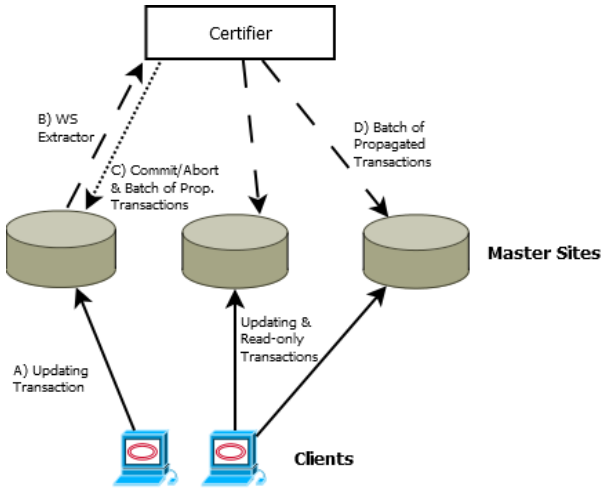


Figure 9: Tashkent's processing of updating transactions. A clients sends an updating transaction to a master site, which processes it locally (Step A). Before committing, the transaction's Write Set is extracted and sent to the certifier (Step B). The certifier determines whether the transaction commits or aborts by using a global version number. The result is sent back to the site, along with a batch of transaction that need to be propagated (Step C). Periodically, the certifier will propagate a batch of transactions to each site (Step D).

Tashkent makes use of the certifier to implement group commits. It proposes two different configurations: Tashkent Middleware and Tashkent API. The middleware configuration does the group commits at the certifier and propagates the write sets of all the committed transactions to the master sites as a single propagated transaction. The master sites act as main-memory databases, no interaction with disk is needed. This configuration re-utilizes the persistent log that GSI's certifier already implements for recovery purposes and uses it to determine a grouping of transactions.

On the other hand, Tashkent API modifies the underlying DBMS at the master site so it can concurrently execute propagated transactions, while imposing an ordering of commits. Basically, it extends the database's COMMIT command so it accepts an additional parameter: the order in which the transaction should commit. Similarly to Tashkent Middleware, the certifier groups several write sets of different transactions into a single propagated transaction. However, the API configuration can concurrently execute several propagated transactions and they can all be committed as a group by the local DBMS. The local group commit enforces the ordering of transactions given by the extended COMMIT command.

Both API and Middleware configurations yield the same result: a strong level of consistency guarantees. Also, both of them use a deferred propagation strategy. GSI allows read-only transactions to read stale versions of data because these transactions are never forwarded to the certifier. Hence, Tashkent has a scope of global updates. Finally, evaluations performed on Tashkent showed that the middleware configuration had lower latencies and higher throughput than the API configuration. This is because the API has to perform an additional validation to make sure that there are no write-write conflicts between the propagated transactions. Because propagated transactions are a grouping of updating transactions, it is possible that a conflict that never ocurred at the certifier now occurs at the replicas.

4. CONCLUSIONS

Space for conclusions.

5. REFERENCES

[1] ISO/IEC 9075-1:2011. *Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*, 2011.

[2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.

[3] Philip Bernstein and Eric Newcomer. *Principles of Transaction Processing: For the Systems Professional*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[4] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S. Seshadri, and Avi Silberschatz. Update propagation protocols for replicated databates. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 97–108, New York, NY, USA, 1999. ACM.

[5] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4):20:1–20:42, December 2009.

[6] Prima Chairunnanda. Multi-master replication for snapshot isolation databases. Master's thesis, University of Waterloo, 2013.

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

[8] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[9] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, September 2010.

[10] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 715–726. VLDB Endowment, 2006.

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

[12] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 1–8, New York, NY, USA, 1984. ACM.

[13] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. *SIGOPS Oper. Syst. Rev.*, 40(4):117–130, April 2006.

[14] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, January 2012.

[15] Chuntao Hong, Dong Zhou, Mao Yang, Carbo Kuo, Lintao Zhang, and Lidong Zhou. Kuafu: Closing the parallelism gap in database replication. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1186–1195, April 2013.

[16] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[17] Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Pearson Education, Inc., 3rd edition, 2011.

[18] E. Pacitti, E. Simon, and R. Melo. Improving data freshness in lazy master schemes. In *Distributed*

Computing Systems, 1998. Proceedings. 18th International Conference on, pages 164–171, May 1998.

[19] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '04, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[20] Load Balancing PostgreSQL 9.4: High Availability and Replication. http://www.postgresql.org/docs/9.4/static/warm-standby.html. Accessed: 2015-07-01.

[21] MySQL 5.7: Semisynchronous Replication. https://dev.mysql.com/doc/refman/5.7/en/replication-semisync.html. Accessed: 2015-07-01.

[22] SQL Server 2016: Transactional Replication. https://msdn.microsoft.com/en-us/library/ms151176.aspx. Accessed: 2015-07-01.

[23] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, August 2013.

[24] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[25] LinkedIn: Project Voldemort. http://www.project-voldemort.com/voldemort/. Accessed: 2015-07-01.

[26] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. Consistency algorithms for multi-source warehouse view maintenance. *Distributed Parallel Databases*, 6(1):7–40, January 1998.