

Chapter 2

Distributed Database Management Systems: Architectural Design Choices for the Cloud

Joarder Mohammad Mustafa Kamal and Manzur Murshed

Abstract Cloud computing has changed the way we used to exploit software and systems. The two decades' practice of architecting solutions and services over the Internet has just revolved within the past few years. End users are now relying more on paying for what they use instead of purchasing a full-phase license. System owners are also in rapid hunt for business profits by deploying their services in the Cloud and thus maximising global outreach and minimising overall management costs. However, deploying and scaling Cloud applications regionally and globally are highly challenging. In this context, distributed data management systems in the Cloud promise rapid elasticity and horizontal scalability so that Cloud applications can sustain enormous growth in data volume, velocity, and value. Besides, distributed data replication and rapid partitioning are the two fundamental hammers to nail down these challenges. While replication ensures database read scalability and geo-reachability, data partitioning favours database write scalability and system-level load balance. System architects and administrators often face difficulties in managing a multi-tenant distributed database system in Cloud scale as the underlying workload characteristics change frequently. In this chapter, the inherent challenges of such phenomena are discussed in detail alongside their historical backgrounds. Finally, potential way outs to overcome such architectural barriers are presented under the light of recent research and development in this area.

Keywords Cloud computing · Distributed database · ACID · CAP · Replication · Partitioning · BASE · Consistency · Trade-offs

J. M. M. Kamal (✉)

Gippsland School of Faculty of Information Technology, Monash University, Clayton, VIC, Australia
e-mail: Joarder.Kamal@monash.edu

M. Murshed

Faculty of Science and Technology, Federation University, Churchill, VIC, Australia
e-mail: Manzur.Murshed@federation.edu.au

2.1 Introduction

In recent years, with the widespread use of Cloud computing based platform and virtual infrastructure services, each and every user-facing Web application is thrusting to achieve both ‘high availability’ and ‘high scalability’ at the same time. Data replication techniques are long being used as a key way forward to achieve fault-tolerance (i.e., high availability) and improving performance (i.e., maintaining system throughput and response time for an increasing number of users) in both distributed systems and database implementations [29]. The primary challenges for replication strategies include: (1) replica control mechanisms—‘where’ and ‘when’ to update replicated copies, (2) replication architecture—‘where’ replication logic should be implemented and finally (3) ‘how’ to ensure both the ‘consistency’ and the ‘reliability’ requirements for the target application. These challenges fundamentally depend on the typical workload patterns that the target application will be going to handle as well as the particular business goals it will try to meet.

Even in the absence of failure, some degree of replication is needed to guarantee both ‘high availability’ and ‘high scalability’ simultaneously. And, to achieve the highest level of these two properties, data should be replicated over wide area networks. Thus, the replicated system inherently imposes design trade-offs between consistency, availability, responsiveness and scalability. And, this is true for deployments either within a single data centre over local area network (LAN) or in multiple data centres over wide area network (WAN).

A high-level Cloud system block diagram is portrayed in Fig. 2.1, where a typical layout of a multi-tier Cloud application has been shown in a layered approach.

According to Fig. 2.1, end-users’ requests originate from the typical client-side applications such as browsers and desktop/mobile apps through HTTP (which is a request/reply based protocol) interactions. Database name server (DNS), Web and content delivery network (CDN) servers are the typical first-tier Cloud services (typically stateless) to accept and handle these client requests. If it is a read-only request, then clients can be served immediately using cached data, otherwise update (i.e., insert, update, delete) requests need to be forwarded to the second-tier services.

Application servers, on the other hand, process these forwarded requests based on the coded logic and process the operation using in-memory data objects (if available) or fetch the required data from the underlying database-tier. Model view controller (MVC) pattern-based logic implementation can be considered as an example. In an MVC application, user requests (typically URLs) are mapped into ‘controller’ actions which then fetch data from appropriate ‘model’ representation and finally set variables and eventually render the ‘view’. If in-memory representation of the model data is not available then the model component needs to initiate a transactional operation (like using ActiveRecord or DataMapper patterns) in the inner-tier database services. Otherwise, in-memory update can take place and updated information can be later pushed into the database.

Note that, application servers are typically ‘stateful’ and may need to store state values (e.g., login information) as session objects into another highly scalable key-value store. While in the inner-tier, database can be partitioned (i.e., Shards)

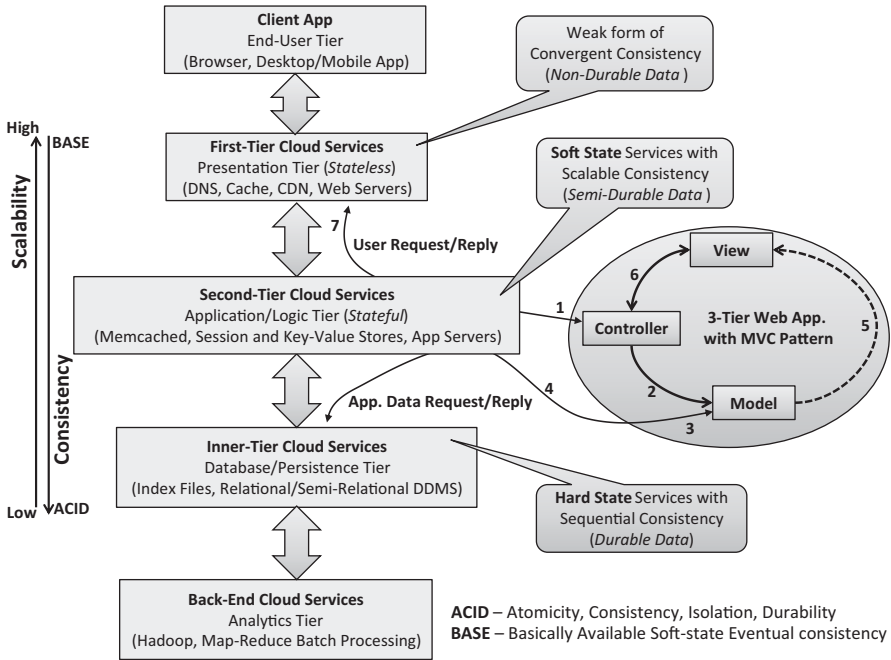


Fig. 2.1 Different service tiers of a typical 3-tier Web application and their interactions within the Cloud computing model. *DNS* database name server, *CDN* content delivery network, *DDMS* distributed database management systems

as well as replicated based on application functionality and requirements. Based on the replica control and placement policies, data can be fetched (if read-only) or updated accordingly and ultimately reply back to the model component in the MVC implementation at the upper-tier.

Our curiosity is to investigate how this end-to-end request-reply procedure access and utilise these durable and consistent data objects into different tiers of a typical Cloud system. And, gradually this will also clarify the system-design trade-offs for different components in a large-scale distributed systems. Read-only user requests for static information (and some form of dynamic information) can be directly served by first-tier Cloud servers based on the data staleness bound. As durability is not guaranteed in this stateless tier, stored information can be lost due to failures. Again, high availability (by means of rapid responsiveness) and high scalability are needed to handle client requests with a typically converging consistency requirement, which also depend on cache expiration and freshness policies.

For read requests which cannot be served due to expiry now can be fetched from the in-memory data objects that reside in the application tier. Update and scan requests typically routed to the second-tier services and mapped accordingly as explained earlier. In this tier, application logics are typically executed using the in-memory data representations which offer scalable consistency with semi-durability. Based on the implementation mechanism of this second-tier services,

consistency guarantees reside in the development of soft-state services with reconstructible data pieces. If the required data are not available, then the application logic initiates transactional operations into the inner-tier databases. And they usually offer strong consistency (via atomicity, consistency, isolation and durability (ACID) properties) and durable data (via replication services). However, scalability is hard to achieve in this tier as stronger form of consistency comes with the price of responsiveness.

2.1.1 Why ACID Properties Are Hard to Scale

It is well known that scale-out and utilisation are far more cost-effective using thousands of commodity hardware than through high-end server machines [3]. However, deploying user facing Web applications with typical transactional workload in such shared nothing architecture [41] is not trivial. Again, the underlying database system itself needs to be replicated and/or partitioned to provide required read/write scalability for the end users. The problem resides in the fact that if a transaction needs to access data which span over multiple machines, it is pretty complex to guarantee ACID properties. At the same time, managing distributed transaction and executing them in parallel into a number of replicas to ensure atomic success or abort is also challenging.

Atomicity property (in ACID) requires a distributed commit protocol such as ‘2-phase commit’ (2PC) to run across multiple machines involved in a particular transaction. In the meanwhile, the isolation property insists that the transactions should acquire all of its necessary locks for the total duration of the run of a 2PC. Thus, each transaction (whether it is a simple or complex one) requires a considerable amount of time to complete a 2PC round while performing several round trips in a typical failure-free case. While in case of failure of 2PC coordinator, the total system blocks and a near-success transaction can be aborted due to a single suddenly failed replica.

Again, having data replication schemes in action, to achieve strong system-wise consistency (e.g., possibly via synchronous update) requires to make trade-off with the system response-time (as well as transactional throughput). Finally, in a shared-nothing system with failing hardware ensuring durable transactional operation in the face of strong consistency is far away from reality and practice. As mentioned earlier, real system designers have to make diverse set of trade-offs to ensure different levels of consistency, availability and latency requirements in face of scalable ACID semantics.

2.1.2 CAP Confusion

Current Cloud solutions support a very restricted level of consistency guarantees for systems which require high assurance and security. The issue develops from the

misunderstanding of the design space and principle like consistency, availability and partition (CAP) devised by Eric Brewer [10], and later proved by Gilbert and Nancy [16]. According to the CAP principle, the system designer must choose between consistency and availability in the face of network partition. And, this trade-off comes from the fact that to ensure ‘high availability’ in case of failure (i.e., crash-recovery, partition, Byzantine, etc.) data should be replicated across physical machines.

In recent years, due to the need for higher system throughput in the face of increased workload and high scalability, distributed database systems (DDBS) have drawn the utmost attention in the computing industry. However, building DDBSs are difficult and complex. Thus, understanding of the design space alongside with the application requirement is always helpful for the system designers. Indeed, the CAP theorem has been widely in use to understand the trade-offs between the important system properties—the CAP tolerance.

Unfortunately, today’s development trend indicates that many system designers have misapplied CAP to build somewhat restrictive models of DDBSs. The narrower set of definitions presented in the proof of CAP theorem [16] may be one of the reasons. In their proof, Gilbert and Nancy considered ‘atomic/linear consistency’ which is more difficult to achieve in a DDBS while being at fault and partition tolerant. However, Brewer actually considered a more relaxed definition of the ‘Consistency’ property referring to the case considered in the first-tier of a typical Cloud application as shown in Fig. 2.1.

In reality, the probability of partition in today’s highly reliable data centre is rare although short-lived partitions are common in WANs. So, according to CAP theorem, DDBSs should provide both ‘availability’ and ‘consistency’, while there are no ‘partitions’. Still, due to extreme workload or sudden failure, it might be the case that the responsiveness of inner-tier services is lagging behind comparing to the requirements for the first-tier and second-tiers services. In such a situation, it would be better to value quick responses to the end users using cached data to be remaining act as available. The goal is to have a scalable Cloud system that remains available and responsive to the users even at the cost of tolerable inconsistency, which can be deliberately engineered in the application logic to hide the effects.

In his recent article [11], Eric Brewer has revisited the CAP trade-offs and mentioned the unavoidable relationship between latency, availability and partition. He argued that a partition is just time bounded on communication. It means that failing to achieve consistency in a time-bound frame, i.e. facing P, leads to a choice between C and A. Thus, to achieve strong ACID consistency in cases either there is a partition or not, a system should both compensate responsiveness (by means of latency) and availability. On the other hand, a system can achieve rapid responsiveness and high availability within the same conditions while tolerating acceptable inconsistency.

To this end, it is fair enough to suggest that design decisions should be made based on specific business requirements and application goals. If an application strives for consistent and durable data, all time scalability will be limited, and high availability will not be visible (due to low responsiveness). Otherwise, if the target is to achieve scalability and high availability, the application should be able to live with acceptable level of inconsistency.

In Sect. 2.2, important components and concepts of distributed databases, i.e., transactional properties, are discussed. Strategies to update replicated data and different replication architectures, partitioning schemes and architectures along with classifications based on update processing overhead and in context of multi-tier Web application have been elaborated in Sect. 2.3. In Sect. 2.4, the evolution of modern distributed database systems has been explored in parallel with the architectural design choices and innovative management of replicated and partitioned databases in details. Finally, Sect. 2.5 concludes with the remarks on the important characteristics (i.e., data replication and partitioning) of modern distributed database systems which have been shaped the Cloud paradigm over the past years and thus provided the opportunity to build Internet-scale applications and services with high availability and scalability guarantees.

2.2 Background of Distributed Database Concepts

In the following sub-subsections, the building blocks of a modern distributed database management system is discussed, which will eventually help the reader to understand the ACID properties and their implications in great extent.

2.2.1 *Transaction and ACID Properties*

A transaction T_i is a sequence of read operation $r_i(x)$ and write operation $w_i(x)$ on data items within a database. Since, a database system usually provides ACID properties within the lifetime of a transaction, these properties can be defined as shown below:

- *Atomicity*—guarantees that a transaction executes entirely and commits, or aborts and does not leave any effects in the database.
- *Consistency*—assuming the database is in a consistent state before a transaction starts, it guarantees that the database will again be in a consistent state when the transaction ends.
- *Isolation*—guarantees that concurrent transactions will be isolated from each other to maintain the consistency.
- *Durability*—guarantees that committed transactions are not lost even in the case of failures or partitions.

In contrast to a stand-alone database system, a replicated database is a distributed database in which multiple copies of same data items are stored at multiple sites. And, replicated database systems should be acted as a ‘1-copy equivalence’ of a non-replicated system providing ACID guarantees. Thus, within a replicated environment the ACID properties can be redefined as below:

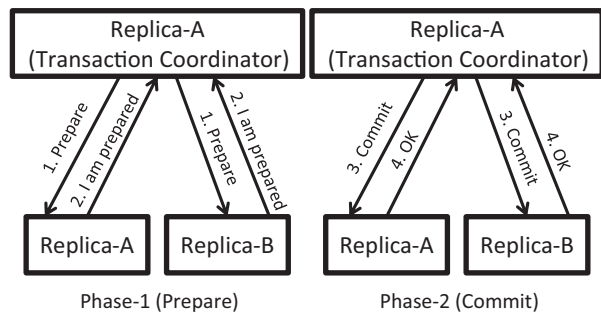
- *1-copy atomicity*—guarantees that a transaction should have the same decision of either all (commit) or nothing (abort) at every replicas which it performs the operation. Thus, some form of ‘agreement protocol’ is necessary to run among the replicas which should force this guarantee.
- *1-copy consistency*—guarantees that a consistent database state should be maintained across all replicas in such a way that the restrictions imposed by the ‘integrity constraints’ (e.g., primary/foreign key) while executing a transaction, are not violated after it ends.
- *1-copy isolation*—guarantees that concurrent executions of a set of transactions across multiple replicas to be equivalent to a serial execution (i.e., order) of this set (as if the set of transactions are running serially in a non-replicated system). Also defined as the ‘1-copy-serialisability’ (1SR) property.
- *1-copy durability*—guarantees that when a replica fails then later recovers, it does not only require to redo the transactions that had been committed locally but also make itself up-to-date with the changes that committed globally during the downtime.

2.2.2 Distributed Transactions and Atomic Commit

When a transaction attempts to update data on two or more replicas, 1-copy-atomicity property needs to be ensured which also influences consistency and durability properties of the data item. To guarantee this, 2PC protocol [17] is typically used. As shown in Fig. 2.2, initially 2PC is originated from the local replica and the scheme includes all the other remote replicas that hold a copy of the data items that are accessed by the executing transaction.

At phase-1, the local replica sends a ‘prepare-to-commit’ message to all participants. Upon receiving this message, the remote replica, if it is willing to commit, replies with a ‘prepared’ message, otherwise sends back an ‘abort’ message. The remote replicas also write a copy of the result in its persistent log which can be used to perform the ‘commit’ in case of failure recovery. While the coordinating local replica receive ‘prepared’ messages from all of the participants (means all remote replicas have persistently written the result into log), only then it enters into phase-2.

Fig. 2.2 The 2-phase commit protocol



The second round message from the coordinator tells the replicas to actually ‘commit’ the transaction. 2PC aims to handle every possible failure and recovery scenarios (like in case of the coordinator fails); thus, transactions are often ‘blocked’ for an unbounded amount of time. ‘3-phase commit’ [40] protocol was proposed lately which is non-blocking. However, it requires more costly implementation in real system as well as only assumes fail-stop-failure model. Thus, in face of network partition, the protocol simply fails to progress. A more elaborate description of distributed transaction processing can be found in [8].

Note that, both 2PC and 3PC protocols are within the solution family of Consensus [50] problems. More recently, Paxos [27, 51], which is another family of protocols (more resilient to failures) to solve the consensus problems, has received much attention in both academia and industry.

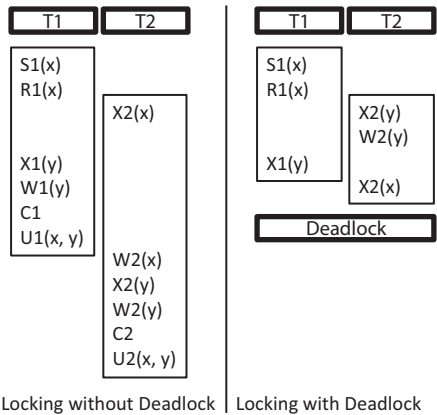
2.2.3 *Distributed Concurrency Control*

Concurrency control mechanism [8] in a database system maintains an impression that concurrent transactions are executing in isolation. There are two families of concurrency control protocols that exist: ‘pessimistic’ and ‘optimistic’. Pessimistic approach is typically implemented using ‘locking’. A ‘shared lock’ is acquired by a transaction to get read-access in the database record (typically the whole ‘row’ in a database ‘table’) and an ‘exclusive lock’ is acquired to have write-access. If a lock cannot be granted by the concurrency control manager, then the involving transaction is blocked in waiting until conflicting locks are released. A shared lock can be granted if there are at most other shared locks currently held on to a record.

On the other hand, an exclusive-lock can only be granted if there are no other locks currently on hold. Thus, read operations are permitted to execute concurrently while write operations must go through serially. Also note that read-only operations may also ‘block’ during a period of exclusive-lock holds by another transaction. Alternatively, a write operation may also ‘block’ during a period of shared-lock holds by another transaction. In order to ensure strict serialisability, all acquired locks are typically released only after the transaction commit or abort. This total mechanism can be implemented through either using ‘2-phase locking (2PL)’ or ‘strong strict 2-phase locking (SS2PL)’ protocol. In phase-1, all required locks are requested and acquired step-by-step from the beginning of a transaction towards its execution. In phase-2, all locks are released in one step based upon commit/abort decision.

As shown in Fig. 2.3, deadlocks can be created by due concurrent transactions racing to acquire locks. In such situations, the concurrency control manager should be able to detect such deadlocks. 2PL/SS2PL can still be used to guarantee 1-copy serialisability; however, it pays the costly penalty in system throughput and latency, i.e., responsiveness. One of the conflicting transactions has to be aborted in all replicas to release its locks, which allow the other transaction to proceed and complete its operations. Sometimes, locking may create unwanted delays through blocking, while the transactional operations could be serialisable.

Fig. 2.3 Deadlocks with pessimistic concurrency control using 2PL



Alternatively, simple ‘atomic commitment protocol’ could be used where all the transactional executions are done within an atomic operation in the participating replicas. Optimistic approach on the other hand, allows concurrent transactions to proceed in parallel. A transaction can create its local copy and perform all the necessary update operations in it. At the end of transaction, a validation phase takes place and checks whether the read-sets of the considered transaction overlaps with the write-set of any transaction that has already successfully validated. If true, it has to be aborted, otherwise it can be committed successfully via writing its changes persistently back to the database.

In DDBS with replication mechanism enabled, a distributed lock manager is required which will try to detect and resolve distributed deadlocks among conflicting replicas in a pessimistic approach. Atomic commit protocols like 2PC/3PC could still be used along with 2PL/SS2PL. One such approach is to achieve global serialisation order instead of distributed locking by using 2PC atomic commit globally while locally applying 2PL/SS2PL. However, achieving global serialisation order is costly and pays the price with restricted system performance. On the other hand, an optimistic approach would try to perform distributed or centralised conflict detection and resolution procedure to rescue. Whichever the case is, the bottom line is implementing distributed concurrency control through locking always creates ‘race condition’ locally which may lead to deadlocks or alternatively require costly conflict and serialisation order management schemes globally.

Cursor stability (CS) is another kind of concurrency control mechanism which uses short ‘read’ locks. A read lock on a data item x is acquired and released directly after the read operation is executed. In situations when a data item is accessed by a read-only operation simultaneously and a write operation is blocked for an unbounded amount of time CS can be used in rescue. Short ‘read’ locks gradually upgraded to exclusive write locks to prioritise the blocked write operations to complete. However, inconsistencies may occur due to ‘lost update’ from another transaction in progress.

2.2.4 Multi-Version Concurrency Control and Snapshot Isolation

In multi-version concurrency control (MCC or MVCC) approach, a database system always performs update operation by creating a new version of the old data item instead of overwriting it. MVCC typically utilises timestamps or transaction IDs in increasing order to implement and identify new data version copies. The benefit of using MVCC is reads will be never blocked by write operations. Read-only access in the database will always retrieve a committed version of the data item. Obviously, the cost incurs in the storing of multiple versions of the same data items. Database that supports MVCC implementation typically adopts snapshot isolation (SI) [8] which performs better with low overhead working with such multiple data versions. However, SI is less restrictive in nature than serialisability thus may allow non-serialisable operations leading to anomalies. In practice, commercial systems also provide lower level of isolation as it is always hard to scale with increasing number of concurrent transactions with serialisability.

SI assumes whenever a transaction writes a data item x , it creates a new version of x ; and when the transaction commits, the version is installed. Formally, if transaction T_i and T_j both write data item x , then T_i commits before T_j and if no other transaction commits in between T_i and T_j and writes x , then T_i 's version is directly ordered before T_j 's version of x . SI adopts two important properties:

- *Snapshot reads*—provides each transaction a snapshot of the database as of the time it starts, i.e., last installed version. It guarantees high transaction concurrency for read-only operations and reads never interfere with writes.
- *Snapshot writes*—writes that occur after the transaction are not visible. It disallows two concurrent transactions (neither commits before the other starts) to update the same data item. It avoids well-known anomalies that can occur in the use of lower-level isolation guarantee.

2.2.5 Isolation Anomalies

Based on the above discussion on different concurrency control mechanism and isolation levels, it would be better to introduce few isolation anomalies which are typically used to appear in the system [21, 8]:

- *Dirty read*—reading an uncommitted version of a data item. For example, a transaction T_j reads an uncommitted version a data tuple x which has been updated by another transaction T_i . However, if T_i later aborts due to any reason, this will also force T_j to abort as well. This is called ‘cascading aborts effect’.
- *Lost update*—overwriting updates by concurrent transactions. For example, T_j writes (i.e., overwrites) x based upon its own read without considering the new version of x created by T_i . T_i 's update will be lost.
- *Non-repeatable read*—reading two different versions of a data item during a transaction execution period.

- *Read skew*—if MVCC is allowed, then it might be possible that by reading different versions of multiple data items which are casually dependent on any applied constraint, is violated.
- *Write skew*—similar to read skew, constraints between casually dependent data items may be violated due to two concurrent writes.

2.3 Replication and Partitioning Mechanisms

2.3.1 Replica Control Strategies

Replica control strategies can be categorised based on two primary dimensions: *where* updates will be taken place and *when* these updates will be propagated to remote replicas. Considering these criteria, the classification based on [14] is shown in Table 2.1. Considering the ‘when’ dimension, there can be two classes of replica control mechanisms. One is the ‘*eager*’ *replication* that is a proactive approach, where tentative conflicts between concurrent transactions are detected before they commit while synchronously propagate updates among replicas. Thus, data consistency can be preserved while in the cost of high communication overhead which increases the latency. It is also called the *active replication*. The second is the *lazy replication* which is a reactive approach which allows concurrent transactions to execute in parallel and make changes in their individual local copies. Therefore, inconsistency between replicas may arise as update propagations are delayed by performing asynchronously after the local transaction commits. It is also called as *passive replication*.

Again, based on the ‘where’ dimension, both ‘eager’ and ‘lazy’ replication scheme can be further divided into two categories. One is the *primary copy update* which restricts data items to be updated in a centralised fashion. All transactions have to perform its operations in the primary copy first which then can be propagated either synchronously or asynchronously to other replicas. This scheme is benefited from a simplified concurrency control approach and reduces the number of concurrent updates in different replicas. However, the single *primary copy* itself may be a single point of failure and potentially create bottleneck in the system. On

Table 2.1 Typical classification of replica control strategies [18]

Propagation vs. ownership	Eager	Lazy	Remark
Primary copy	1 transaction 1 owner	N transactions 1 owner	Single owner (can be potential bottleneck)
Update anywhere	1 transaction N owners	N transactions N owners	Multiple owner (harder to achieve consistency)
	Synchronous update (converging consistency)	Asynchronous update (diverging consistency)	

the other hand, the second category of *update anywhere* approach allows transactional operations to be executed at any replicas in a distributed fashion. Coordination between different replicas is required which may lead to high communication cost while using *eager update* propagation. While using *lazy propagation* potentially leads to potential inconsistencies which require expansive conflict detection and reconciliation procedure to resolve.

A trade-off is typically considered where high performance can be achieved by sacrificing consistency via using ‘lazy’ replication schemes. Alternatively, one can get consistency in the price of performance and scalability via using ‘eager’ replication scheme. Further classification of replica control mechanisms can be deduced in this regard. One of the popular replication technique is to implement read-one-write-all (ROWA) solution where read operations acquire local locks while write operations need distributed locks among replicas.

The correctness of the scheme can be satisfied with ‘1SR’. 2PC and SS2PL are also required to ensure atomic transactional commits. An improved version of this approach is read-one-write-all-available (ROWAA) which improves the concurrency control performance in the face of failure. Quorum-based replication solutions are also an alternative choice which typically reduces the replication overhead through only allowing a subset of replicas to be updated in each transaction. However, quorum systems also do not scale well in situations where update rates are high. An excellent analytical comparison can be found at [21] regarding this analogy.

In [18], Jim Gray was the first to explore the inherent dangers of replication in these schemes when scalability matters. Gray pointed out that as the number of replicas increase, it also exponentially increases the number of conflicting operations, response time and deadlock probabilities.

For ‘eager’ schemes, the probability of deadlocks increased by the power of three of the number of replicas in the system. Again, disconnected and failed nodes also cannot use this approach. In the ‘lazy’ scheme, the reconciliation rates (in *update anywhere*) and the number of deadlocks (in *primary copy*) sharply rise with the increase of the number of replicas.

Alternatively, Gray [18] proposed the *convergence property* instead of strict serialisability provided by the ACID semantics. It considers that if there are no updates within a sufficient amount of time, then all participating replicas will gradually converge to a consistent state after exchanging ongoing update results. He coined the examples of Lotus Notes, Microsoft Access and Oracle 7 which were typically proving such kind of convergence property at that time.

Commercial implementation of replica control schemes also followed the ‘lazy’ approaches and offered different options for appropriate reconciliation procedure for a long time. Research efforts were also engaged in solving and optimising the inconsistencies that arise from ‘lazy’ approaches like weak consistency models, epidemic strategies, restrictive node placement, using ‘lazy’ primary approach and different kinds of hybrid solutions. However, maintaining consistency over the impacts of inconsistency is much simpler to implement, but hard to optimise for scalability.

To meet this challenge, Postgres-R [22] was developed which provides replication through an ‘eager’ approach using *group communication* primitives, thus totally

avoids the cost of distributed locking and deadlocks. The Postgres-R approach uses a ‘shadow copy’ of the local data item to perform updates, check integrity constraints, identify read-write conflicts and fire triggers. The changes that are made into a shadow copy propagate to the remote replicas at commit time, thus vastly decreases the message/synchronisation overhead in the system. Read operations are always performed locally as following a ROWA/ROWAA approach.

Thus, there are no overheads for read operations in the system. Update (i.e., write) operations of a transaction are bundled together into a write-set message and multicast in total order to all replicas (including itself) to determine the serialisation orders of the running transactions. Each replica uses this order to acquire all locks required by that transaction in a single atomic step. The total order is used to serialise the read/write conflicts at all replicas at the same time. Thus, by acquiring locks in the order in which the transactions arrive, all replicas are performing the conflicting operations in the same order. As a plus point, there will be no chance for deadlocks. In case of read/write conflicts, reads are typically aborted as a straightforward solution while different optimisations can also be possible. After completion/abortion of the write operations in the local replica, the decision is propagated to the remote replicas.

Performance results from [22] indicate that Postgres-R can scale well with increasing workloads and at the same time boost system throughput by reducing communication overheads and by eliminating the possibility of deadlocks. A more detail of this work can be found at [23]. However, replica control, i.e., coordination is still a challenging task in practical systems and two essential properties always need to ensure: (1) Agreement—every non-faulty replicas receive every intended request and (2) order—every non-faulty replica processes the request it receives in the same order. Interested readers can find an elaborate discussion in [51] on how we can maintain these properties, thus understand how state machine replication works using consensus protocol like Paxos [27] and what determinism in database replication really means.

2.3.2 Replication Architectures

One of the most crucial choices is ‘where’ to implement the replication logic. It might be implemented tightly with the database in its kernel. Alternative approach might be using a middleware to separate the replication logic from the concurrency control logic implemented in the database. Based on these choices, replication logic can be implemented in the following ways (see Fig. 2.4):

- *Kernel-based*—replication logic is implemented in the database kernel and therefore has the full access to database internals. The benefit is that clients can directly communicate with the database. On the other hand, any change in the database internals (e.g., concurrency control module) will directly impact the functionalities of replica control module. Again, refactoring database source code is cumbersome and the implementation is always vendor specific. Also called as ‘white-box replication’.

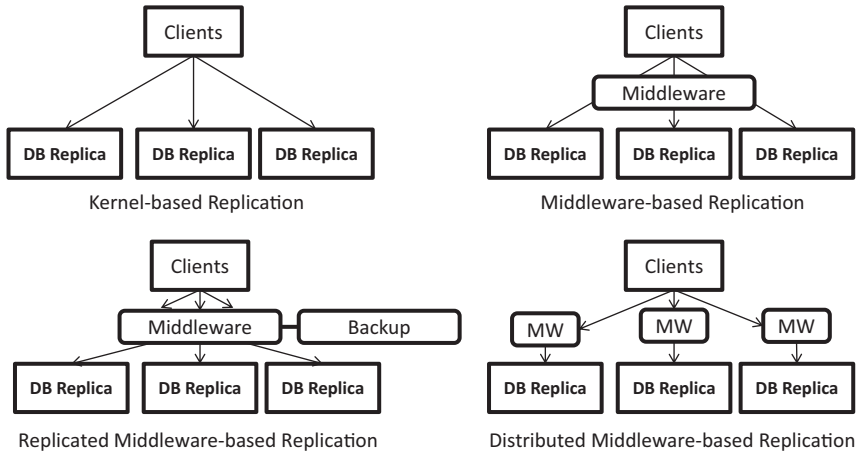


Fig. 2.4 Different replication architectures

- *Centralised middleware-based*—replication logic can be separately implemented into a middleware layer. It provides much flexibility and independence to integrate with any database. However, the functionalities of concurrency control module have to be re-implemented. It is also called as ‘black-box replication’. A modified version of this scheme can be called ‘gray-box replication’ where the database itself should expose the required concurrency control functionalities through specific interface for the middleware to utilise in replica control scheme.
- *Replicated centralised middleware-based*—to avoid single point of failure and bottlenecks, backup middleware can be introduced. However, failover mechanisms are hard to implement to support hot-swap for running transactions and coordinating with the application layer modules.
- *Distributed middleware-based*—every database replica is coupled with a middleware instance and act as a single unit of replication. In case of failover, the total unit can be swapped. Again, the approach is more suitable in WANs reducing the overhead of clients to communicate with the centralised middleware each time it wants to initiate transactional operations.

2.3.3 Partitioning Architecture

It is obvious that replicating data to an extent will increase the read capacity of the system. However, after a certain replication factor, it might be difficult to maintain consistency even if ‘eager’ replication and synchronous update processing are used. On the other hand, write capacity can be scaled through partial replication where only subsets of nodes are holding a particular portion of the database. Thus,

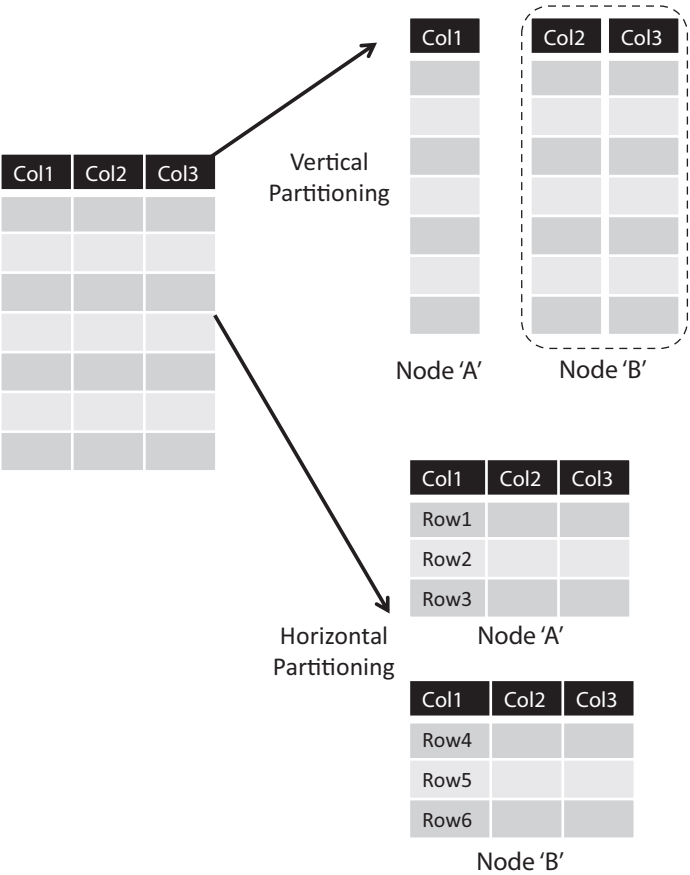


Fig. 2.5 Database partitioning techniques—vertically and horizontally

write operations can be localised and the overheads of concurrent update processing can be reduced. Sharding is a technique to split data into multiple partitions (i.e., Shards). There are two basic ways of partitioning data as shown in Fig. 2.5:

- *Vertical partitioning*—by splitting the table attributes (i.e., columns) and thus creating tables with small number of attributes. It only offers limited scalability in spite of the ease of deployment. The main idea is to map different functional areas of an application into different partitions. Both the datasets and workload scalability are driven by different functional aspects of an application. Thus, it is necessary to pick up the right tables and column(s) to create the correct partition, because the ‘join’ operations in a relational database will now need to be performed within the application code. Hence, the underlying database will no longer support relational schema, and apparently the application scalability is restricts to its hosting node’s resource capacity.

- *Horizontal partitioning*—by splitting the tuples (i.e., rows) across different tables. It allows scaling into any number of partitions. The tuples are partitioned based on a key which can be hash based, range based or directory based. Join operations are similarly discouraged to avoid cross-partition queries. The performance of write operations mostly depends on the appropriate choice of shard key. If sharding is done properly, then the application controller can route the write operations towards the right server.

The bottom line is that sharding a database results in partitioned datasets spread over single-to-multiple data centres, thus forcing the beauty of relational model to reduce. In recent years, NoSQL communities have picked up the trend to abandon relational properties and SQL in favour of high-scalability by only supporting key-value type accesses in their data stores. However, many researchers have already pointed out that abandoning SQL and its feature has nothing to do scalability. Alternatively, many have also indicated ways where careful system and application design can lead to the desired level of scalability [39]. There has been a debate going on in the recent years between these two communities and interested readers may head towards [42, 44, 28] to get a glimpse of it.

2.3.4 *Classification Based on Update Processing Overheads*

Replication architecture also depends on ‘how’ data is actually replicated. Depending on the overheads incurred by the update processing operations, data items can be replicated into all nodes participating in the system or into a subset of nodes. The former one is called *full replication* while the later one is called *partial replication*. It is to be noted here that the primary overhead in replication resides in the update processing operations for the local and remote submissions.

There are two basic choices: *symmetric update processing* and *asymmetric update processing*. The former choice requires a substantial amount of resources (i.e., CPU, I/O in the remote replicas); it may also initiate divergence consistency for non-deterministic database operations (like updating a value with current time). Alternatively, in the asymmetric update processing, the operations are first performed locally and only the changes (along with corresponding primary identifiers and after-image values) are bundled together in the write sets, then forwarded to the remote replicas in a single message. This approach of processing still holds even if the system is using ‘eager’/‘active’ replication scheme.

Depending on the update processing approaches, we can consider the trade-offs between using the *full replication* and *partial replication* schemes. Full replication technique requires an exact snapshot of the local database into every other remote replicas, which may face high-system overheads in the face of increased update workloads. Both symmetric and asymmetric update processing introduce a level of overhead as data needs to be updated into every replicas. However, by using partial replication scheme, one can reduce this overhead and localise the update processing based on their origination.

Surprisingly, partial replication also comes with its own challenges. There are several variants of the partial replication, e.g., (1) *pure partial replication*—where each node has only copies of a subset of the data items, but no node contains a full copy of the total database and (2) *hybrid partial replication*—where a set of nodes contain a full set of the data items, while another set of nodes are partial replicas containing only a fraction of the data sets.

Now, depending on the transaction, it might want to access data items on different replicas in a pure partial replication scheme. It is non-trivial to know which operation will access which data items in the partial replicas. Thus, flexibility is somehow reduced by typical SQL transactions which often need to perform ‘join’ operations between two tables. However, if the database schema can be partitioned accordingly and workload pattern is not changing frequently, then the benefits of localising of update processing can be revealed.

Considering the case of hybrid partial replication, update operations need to be applied fully in the replicas which contain the full set of database. With the increase in the number of transactions, these nodes might create hotspots and bottlenecks. The beauty of the hybrid approach is that while read operations can be centralised to provide more consistent snapshots of data items, the write operations can be distributed among partial replicas to reduce writing overheads. The bottom line is that it has been always challenging to know the transactional properties (like which data items need to access) and apply partial replication accordingly. However, if the application requirements are understood properly and workload patterns are more or less static, then partial replication can exploit the scalability goals.

2.3.5 Classification Based on Multi-Tier Web Architecture

Recalling the example drawn in Fig. 2.1, real-life Web applications are typically deployed in multi-tier Cloud platforms. Each tier is responsible to perform specific functionalities and coordination between these tiers and is necessary to provide the expected services to the end users. Hence, replicating a single tier always restricts scalability and availability limits. Again, apart from being read-only or update operations, workloads can be compute intensive (require more resource and scalability at the application/logic tier) or data intensive (require more ability in the inner database tier).

Again, considering failure conditions, replication logic should work in such ways that the interdependencies between multiple tiers should not lead to multiple workload execution both in the database and application servers [24]. For example, despite failure, ‘exactly-one’ update transaction should be taken place in the corresponding database tier and its entire replica for a single transactional request forwarded from the application tier. Based on this analogy, there can be two architectural patterns for replicating multi-tier platforms [20] as listed below:

- *Vertical replication pattern*—this pairs one application and one database server to create a unit of replication. Such units can be then replicated vertically to

increase the scalability of the system. The benefit of this approach is that replication logic is transparent to both application and database servers; thus, they can work seamlessly. However, challenges reside in the fact that particular application functionalities and corresponding data need to be partitioned appropriately across the whole system to get the target scalability. Much engineering cost and effort are needed for such kind of implementation; thus, in reality, these systems can be still seen very few in numbers.

- *Horizontal replication pattern*—here, each tier implements replication independently and requires some ‘replication awareness’ mechanism to run in between to make necessary coordination. In contrast to the vertical replication pattern, the beauty here is that one can scale flexibly based on the necessity across individual tier. However, without any awareness support to know whether the cooperating tier is replicated or not, it is not able to provide the utmost performance the system could achieve. In reality, this type of systems can be seen almost everywhere in the computing industry; however, they are still in lack of appropriate replication awareness mechanism which is still left as an open challenge.

To support these two categories, other architectural patterns also need to be considered like replica discovery and replication proxy, session maintenance, multi-tier coordination, etc. Several examples of real implementations based on these patterns can be found at [20, 33, 34, 35]. However, replication control via multi-tier coordination is still an open research problem both in academia and industry.

2.4 Distributed Database Systems in the Cloud

2.4.1 *BASE and Eventual Consistency*

The BASE (Basically Available, Soft state, Eventually consistent) acronym [36] captures the CAP reasoning. It devises that if a system can be partitioned functionally (by grouping data by functions and spreading functionality groups across multiple databases, i.e., shards), then one can break down sequence of operations individually and pipeline them for asynchronous update on each replicas while responding to the end user without waiting for their completion. Managing database transactions in a way that avoids locking, highly pipelined, and mostly depends on caching raise all kinds of consistency worries into surface.

While ACID can be seen as a more pessimistic approach, BASE, in contrast, envisions for a more optimistic approach. Availability in BASE systems is ensured through accepting partial partitions. Let us consider a ‘user’ table in a database which is sharded across three different physical machines by utilising user’s ‘last_name’ as a shard key which partitions the total datasets into the following shards A-H, I-P and Q-Z. Now, if one of the shards is suddenly unavailable due to failure or partition, then only 33.33% users will be affected and the rest of the system is still

operational. But, ensuring consistency in such kind of system is not trivial and not readily available like ACID systems. Thus, the consideration of relaxed consistency guarantees arises. One can consider achieving consistency individually across functional groups by decoupling the dependencies between them. As proposed in [36], a persistent pipelined system can tackle the situations where relative ordering and casual relationship is necessary to maintain or one consider de-normalised database schema design.

The ‘E’ in BASE which stands for ‘eventual consistency’ [45, 46] guarantees that in the face of inconsistency the underlying system should work in the background to catch up. The assumption is that in many cases it is hard to distinguish these inconsistent states from the end-user perspective which is usually bounded by different staleness criteria (i.e., time-bounded, value-bounded or update-based staleness). Later, Eric Brewer [11] had also argued against locking and actually favoured the use of cached data but only for ‘soft’ state service developments, while DDBSs should continue to provide strong consistency and durability guarantees. However, this implication of inconsistency requires a higher level of reconfigurability and self-repair capability of a system that tends to expansive engineering effort.

In [45], Werner Vogels from Amazon described several variations of eventual consistency which can also be combined together to provide a stronger notion while ensuring client-side consistency as follows:

- *Casual consistency*—guarantees that if there is any casual dependencies between two processes, then a committed update by one process will be seen by another process and can be superseded by another update.
- *Read-your-writes consistency*—guarantees that after an update of a data item, consecutive reads always get that updated value.
- *Session consistency*—guarantees that as long as the session exist, read-your-write consistency can be provided.
- *Monotonic read consistency*—guarantees if a process reads a particular value of an object, then any subsequent reads will not see any previously committed value.
- *Monotonic write consistency*—guarantees to serialise writes by the same process.

At the server-side consistency, Vogels [45] argues that one should look at the flow of update propagation. One can consider a quorum-based replicated DDBS [35] with N nodes where W nodes replicas are responsible to accept a write and R replicas are contacted while performing a read. Then, if $W + R > N$, then read and write sets are always overlapped, and the system provides stronger form of consistency. Again, if $W < (N + 1)/2$, then there is a definite possibility of conflicting writes as the write sets do not overlap. On the other hand, if the read and writes do not overlap as $W + R \leq N$, then a weaker form of eventual consistency is provided by the system where stale data can be read. In case of network partitions, quorum systems can still handle read and write requests separately as long as these sets can communicate with a group of clients independently. And, later reconciliation procedures can run to manage conflicting updates within replicas.

In [9], Ken Birman has effectively shown ideas that it is possible to develop scalable and consistent soft-state services for the first tier of the Cloud system if one is ready to give up durability guarantee. He argues that the ‘C’ from the CAP theorem actually relates to both ‘C’ and ‘D’ in ACID semantics. Therefore, by sacrificing durability, one can scale through first to inner-tier Cloud services while at the same time can guarantee strong consistency.

In reality, systems that utilises group communication semantics (e.g., membership management, message ordering, failure coordination, recovery, etc.) can achieve consistent replication schemes to support both high availability and high scalability. Google’s Spanner [14] is one of the most prominent examples of this kind. Although these systems can exploit the requirements for first-to-inner service tiers, the consistency guarantee usually comes with a high engineering cost and lacks generalised patterns/solutions.

Lastly, based on the current usage of Cloud systems, inconsistencies can somewhat be tolerated for improving read/write performances under increasing workloads and handling partition cases. However, the level of scalability that Cloud systems can achieve is a long cherished dream for system which prefers high assurance (i.e., both availability and consistency), reliability and security.

2.4.2 *Revisiting Architectural Design Space*

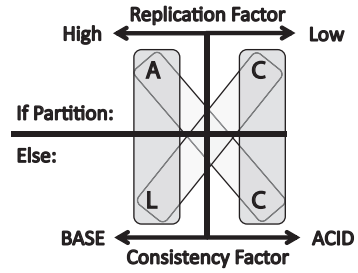
To overcome the confusion that arises from the CAP theorem, it is necessary to revisit the design space in the light of distributed replication and data partitioning techniques. This insight will also enable to clarify the relationship between the related challenges with ACID and BASE as discussed above. In [1], Daniel Abadi was the first to pinpoint the exact confusion that arises from CAP and clarifies the relationship between consistency and latency. He proposed a new acronym PACELC which he believed to be the actual representation of reality.

PACELC in a single formulation: if there is a partition (P), how does the system trade-off exist between availability and consistency (A and C); else (E) when the system is running as normal in the absence of partitions, how does the system trade-off exist between latency (L) and consistency (C)?

The PACELC formulation is shown in Fig. 2.6 under several considerations like based on replication factor, consistency level, system responsiveness and partition-tolerance level. We will explain this phenomenon with respect to PACELC classification for distributed system design. As Abadi explained in [2], there can be four possible system types as follows:

- *A-L systems*—always give up consistency in favour of availability in case of partition otherwise prefer latency during normal operating periods. Example—Apache Cassandra [4], Amazon’s DynamoDB [3] and Riak [38] (in their default settings).
- *A-C systems*—provide consistent reads/writes in the typical failure-free scenarios; however, in failure cases, consistency sacrifices (for limited period until

Fig. 2.6 Design space for large-scale distributed system development. *BASE* basically available, soft state, eventually consistent; *ACID* atomicity, consistency, isolation and durability



the failure recovers) would remain available. Example: MongoDB [31] and CouchDB [5].

- *C-L systems*—provide baseline consistency (as defined by the system, e.g., timeline consistency) for latency during normal operations, while in case of partitions it prioritises consistency over availability (or, being slow responsiveness which imposes high latency). Example: Yahoo! PNUTS [13].
- *C-C systems*—disallow to give up consistency either in the case of partition or not and thus incur availability (i.e., responsiveness), and latency costs as the trade-off. Example: BigTable [12]/HBase [6] and H-Store [19]/VoltDB [46].

This is to be noted here that, completely giving up availability is not possible at all; otherwise it will be a useless system. Availability actually spans over two dimensions: (1) resilient to failures, and (2) responsiveness in both failure and failure-free cases. Interested readers are also encouraged to read Dan Weinreb's blog entry [49] which further clarifies how availability and latency relate to each other. Similarly, completely inconsistent systems are also useless; thus, the level of consistency varies in between its weaker and stronger forms. Let us now discuss these system design choices in more detail under the light of the above mentioned considerations.

2.4.2.1 Consistency Factor

Stronger consistency models which are tightly coupled with a DBMS always ease the life of the application developer. Depending on the application requirement, giving up ACID properties in favour of BASE is also inadequate in many situations. However, stronger consistency levels can also be viable to achieve by decoupling logic from the underlying DBMS and implementing along with the replica control scheme.

Quorum-based systems are one of the possible choices in this regard where one can control the level of consistency by restricting read/write quorum requirements. Alternatively, consistency can be ensured in a much fine-granularity [37]. Ensuring entity-level or object-level consistency within a single database can also provide a notion of ACID semantics. Furthermore, entity groups can be considered as a unit of consistency and even multiple groups might act as a unit.

A-L systems which can be viewed as the BASE equivalent tend to provide different variations of eventual consistency all the time. Similar adaption is also true while the system design space gradually shifts towards C-L systems in failure cases. On the other hand, A-C and C-C systems by default tend to achieve stronger form of consistency either in the case of failure or not. However, as indicated earlier providing ACID level consistency (i.e., serialisability) is challenging and costly in DDBSs. Therefore, providing soft level of consistency guarantees like snapshot isolation or even timeline consistency (as provided in Yahoo's PNUTS [13]) seems to be more adaptable in such scenarios.

2.4.2.2 Responsiveness Factor

Responsiveness is the perceived 'delay' between when an end-user or internal system component takes an action such as clicking on a link or forwarding a request, and when the user/component perceives a response. It wraps up two other technical pieces, namely: (1) latency—initial delay to start receiving replies for a corresponding request, and (2) throughput—total time taken for all the contents of a reply to be received completely. These factors are imposed by the service level objective (SLO) goals while considering the design spaces.

One can consider the '8 second rule' [30] which still fits well to measure the responsiveness of modern Cloud applications. It states that 'if a computer system responds to a user action within 100 ms, it's perceived as 'instantaneous'; within 1 s, the user will still perceive a cause-and-effect connection between their action and the response, but will perceive the system as 'sluggish'; and after about 8 s, the user's attention drifts away from the task while waiting for a response'.

Based upon this observation, A-L systems should be chosen where strict and rapid responsiveness is the requirement. Both the A-C and C-L systems will be better on ensuring flexible responsiveness requirements in the face of failure and failure-free cases, respectively. C-C systems pay the costs to keep the system up-to-date and consistent, therefore, slow responsive will be incurred while they are overloaded.

2.4.2.3 Partition-Tolerance Factor

Partitions are not always created from network/communication outage. Sometimes, it might be the case that the system is overloaded and may not be able to respond within the timeout period. Improper network configurations in the intermediate nodes can also cause similar results. Again, the possibility of partition highly depends on whether the system is deployed in a WAN across multiple data centres or LAN within a single data centre. An interesting discussion of practical database errors which can lead to partitioned networks in DDBS can be found in [43].

Primarily based on the deployment strategies, one can consider choosing A-L or C-L system to deploy across multiple data centre distributed over WAN due to

their latency awareness during normal operation periods. On the other hand, A-C and C-C systems will be more preferred in deploying within single data centre over the LAN.

2.4.2.4 Replication Factor

The scalability of today's Cloud systems and DDBS primarily depends on how they are replicated to provide high read/write throughput, although increasing the number of replicas blindly will not make the success. It may create potential bottlenecks and unresponsiveness in the system. As discussed in [2], three types of replication strategies are popularly seen in today's deployment, viz.: (1) Data updates sent to all replicas at the same time (synchronous), (2) data updates sent to an agreed-upon location first (synchronous/asynchronous/hybrid), and (3) data updates sent to an arbitrary location first (synchronous/asynchronous).

Considering the above analogies, option-1 provides stronger consistency level in the costs of increased latency and communication overhead. Thus, it might primarily be suitable for C-C systems. Option-2 with synchronous-update propagation also ensures consistency but only limited to while deployed in LAN/single data centre. With asynchronous propagation, option-2 provides several options for distributing read and write operations. If a primary/master node is responsible for providing read replies and accepting writes, then inconsistencies can be avoided. However, it may be the source of potential bottleneck in case of failures. On the other hand, if reads are served from any node, while the primary node is only responsible for accepting writes, then read results probably reflect inconsistencies.

A combination of synchronous and asynchronous is also possible considering a quorum-based replication strategy. If $R + W > N$, then the system will provide consistent results while gradually divergent in the condition where $R + W \leq N$. Both A-L and C-L systems are well suited for the approaches mentioned above under option-2 as they are flexible and dynamic with latency-consistency trade-offs. Option-3, which is similar to option-2 apart from preferring any node to accept reads and writes, can also be used either in a synchronous or asynchronous fashion. While synchronous setting can incur increased latency, potential inconsistencies will arise using asynchronous setting. A-C and some of the C-L systems might be suitable to fit in this category.

To this end, it seems worthwhile to revisit the design choices as it broadens our mind to think beyond what the CAP theorem actually meant. It also helps to visualise how we can fit the multi-tier Cloud application within the architectural model. Although a more analytical approach to explain these trade-offs will be definitely profound. Modern software-as-a-service (SaaS) applications deployed over very large-scale distributed systems strive for the following *performance goals*: (1) Availability or uptime—what percentage of time the system is up and properly accessible, (2) responsiveness—measure of latency and throughput, and (3) scalability—as the number of users, i.e., workloads increase how to maintain the target responsiveness without increasing cost/user.

2.4.3 *Data Partitioning and Replication Management*

Typical distributed database systems (e.g., HBase [6], Cloud SQL, MongoDB [31] and MySQL Cluster [32]) which usually provide automatic partitioning and load-balancing features only support pre-configured partitioning rules. The system splits and merges the partitions based on the number of nodes (e.g., MySQL Cluster [32]), predefined data volume size (e.g., in HBase [6]), predefined key (e.g., MongoDB [31]) or even based on partitioned schema (Cloud SQL). All of these approaches are unable to adopt to dynamic workload patterns and current resource utilisation profile of the system. Again, sudden increase in workload volume, occurrences of data spikes and hotspots can also influence the change in normal workload characteristics.

However, dynamic partitioning decision making is not possible and often requires human intervention. Hence, these systems normally suffer from sudden workload spikes in any particular partition, hot-spotted partition or database table, partitioning storm and load-balancing problems. These are the potential reasons of restricted system behaviour, unresponsiveness, failures and bottlenecks. In a WAN setting, this leads to replication nightmare and inconsistency problems on top of added latency.

As Cloud systems are growing bigger and bigger day by day with the explosion of big data, automated management of these large-scale distributed systems are often desirable to maintain high scalability and elasticity. Automatic replication/partitioning management schemes are believed to stand as the solution towards these worries and opportunities. These systems can exploit the self-managerial properties (i.e., healing, optimisation, and provisioning) of a typical Cloud platform and ensure more reliability to achieve the target SLO.

Automatic management of partitioning and replication are also necessary in cases where the database is spanned in multiple data centres over WAN in a geographically distributed fashion. It can be also recognised as a classical match for the case of partial replication where individual partitions of the distributed database management systems can be distributed over WAN. The primary challenge here is to maintain rapid consistency among the replicas with an acceptable latency requirement. The trade-offs between replication and partitioning considering partitioning size as an impacting factor can be also explored in this context.

The particular emphasis is on how to find an optimal partition size for load distribution (arise from hot-spotted partitions due to workload pattern) in geo-distributed data centres. Determining an optimal partition size is essential for effective replication and data transfer between physical machines over WAN. In overall, the choice of availability, consistency, and latency play an important role in developing a scheme over WAN where network partitions occur very often and usually are not avoidable.

To understand the significance, one can be motivated by the scenarios of massively multi-player online role playing games (MMOG) and virtual worlds. Scalability in such environment is really challenging and not trivial in contrast to other

Cloud applications. Game and virtual world users are geographically distributed and can personalise the game environment as well as make interactions with other online users. Two kinds of partitioning strategies are generally seen: one is to decompose the game or virtual world based on the application design and functionality, while another possibility is to partition the system, based on the current workload pattern.

Distributing the workloads evenly among the physical servers is really tedious for both of the cases as they may spread in a WAN over several geographical locations. Again, users residing in one system partition are naturally forbidden to access or interact with other users in different partitions. Even if they wish to do so, costly replication process needs to be taken out. Games and virtual worlds like World of Warcraft, Farmville, SimCity, and Second Life are a few of the examples which have such evolving architectures and geographically distributed workload patterns over the WAN; thus, face these challenges. Jim Waldo has mentioned these challenges from a real-world point in [48] while others like the authors in [52, 25, 26] have also discussed related challenges and the significance of reliable scalability issues in MMOG.

Recent development of the Google's Big Data platform Spanner [14] also focused on a geographically distributed consistent data service platform which spans over multiple data centres in the WAN. The argument of whether existing NoSQL solutions are adequate to handle such scalability challenges effectively is still an active topic of discussion among the community [15], and it is believed that the above mentioned approach can direct an appropriate pathway towards the right vision.

2.5 Conclusion

Cloud computing backed up by modern scalable distributed databases provides significant opportunities for the start-up and established businesses as well as presents potential challenges for the system administrators. The development of distributed databases has been continuing over the past four decades, and is still emerging to adopt the Cloud paradigm. However, system designers and administrators should be well aware of the past trials and potential pitfalls. The design space should be well adopted and possible user cases need to be well studied beforehand. This is required to fit target application scenarios into the architectural design space. Although, recent developments have shown notable promises over the past years, most of the approaches are static in nature and not adaptable with dynamic workload behaviours. SaaS applications deployed within Cloud platforms also span over multiple geographical regions and thus require special attentions to adopt with distributed workload characteristics.

Designing a scalable Cloud system requires a high level understanding of the life-cycle management of a modern multi-tier Web application and characterisation of system workloads. These interpretations lead us straight to the exploration of available architectural design choices and off-the-shelf distributed databases to

support underlying high scalability and availability requirements. However, the misunderstanding of CAP theorem over the past decade, and consequent developments of hundreds of NoSQL systems providing relaxed consistency guarantees did not hold us back. In reality, all these efforts have helped the system architects to understand the actual design space for Cloud applications and thus have provided the necessary momentum to modernise the development of distributed database systems in a whole. Again, the core building blocks of a distributed database system have also helped in shaping the general ideas behind effective data replication and partitioning strategies. Eventually these apprehensions have influenced the development of high available, high scalable and partition tolerance Internet-scale Cloud applications. Nowadays, without having a clear picture of the architectural design choices in front, it is tedious to design a scalable Cloud platform. The PACELC acronym clearly identifies this challenge and helped us grasp the relationship between ACID and BASE properties. Still, automatic management of data replication and partitioning in line with workload characteristics and issues arise from multi-tenant environments that are potential challenges to deal with. With the rapid advancement in database and system research and development, it can be hoped that innovative solutions will be soon in place to rescue us from back-breaking labours of system administrations and disaster response situations.

In this chapter, a trail of modern distributed database systems has been drawn alongside the challenges which require urgent attention from the research community. The relationship between how to adopt the past to overcome the challenges at present has been also discussed in a great extent. Different data replication and partitioning techniques have been discussed in details which are essential to achieve massive scalability and elasticity for the Cloud applications. Finally, several approaches have been shown as potential way out to achieve Cloud scale modernisation of distributed database management systems in a dynamic environment for the years to come.

References

1. Abadi DJ (April 2010) Problems with CAP, and Yahoo's little known NoSQL system. <http://dbmsmusings.blogspot.com.au/2010/04/problems-with-cap-and-yahoos-little.html>. Accessed 31 Jan 2014
2. Abadi DJ (2012) Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Comput IEEE* 45(2):37–42
3. Amazon DynamoDB—NoSQL Cloud Database Service (2014) <http://aws.amazon.com/dynamodb>. Accessed 31 Jan 2014
4. Apache Cassandra Project. <http://cassandra.apache.org>. Accessed 31 Jan 2014
5. Apache CouchDB. <http://couchdb.apache.org>. Accessed 31 Jan 2014
6. Apache HBase—Apache HBase Home. <http://hbase.apache.org>. Accessed 31 Jan 2014
7. Armbrust M, Fox A, Griffith R, Joseph AD, Katz RH, Konwinski A, Lee G, Paterson DA, Rabkin A, Stoica I, Zaharia M (2009) Above the clouds: a Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley
8. Bernstein PA, Newcomer E (2009) Principles of transaction processing, 2nd edn. Morgan Kaufmann, San Francisco

9. Birman K, Freedman D, Huang Q, Dowell P (2012) Overcoming CAP with consistent soft-state replication. *Comput IEEE* 45(2):50–58
10. Brewer EA (2000) Towards robust distributed systems (abstract). In: *Proceedings of the nineteenth annual ACM symposium on principles of distributed computing* (New York, NY, USA, 2000), PODC'00, ACM, p. 7
11. Brewer E (2012) CAP twelve years later: how the “rules” have changed. *Comput IEEE* 45(2):23–29
12. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2008) BigTable: a distributed storage system for structured data. *ACM Trans Comput Syst* 26(2), 4(1–4):26
13. Cooper BF, Ramakrishnan R, Srivastava U, Silberstein A, Bohannon P, Jacobsen H.-A, Puz N, Weaver D, Yerneni R (2008) PNUTS: Yahoo!’s hosted data serving platform. *Proc VLDB Endow* 1(2):1277–1288
14. Corbett JC, Dean J, Epstein M, Fikes A, Frost C, Furman JJ, Ghemawat S, Gubarev A, Heiser C, Hochschild P, Hsieh W, Kanthak S, Kogan E, Li H, Lloyd A, Melnik S, Mwaura D, Nagle D, Quinlan S, Rao R, Rolig L, Saito Y, Szymaniak M, Taylor C, Wang R, Woodford D (2012) Spanner: google’s globally distributed database. In: *Proceedings of the 10th USENIX conference on operating systems design and implementation* (Berkeley, CA, USA) OSDI’12, USENIX Association, pp 251–264
15. Floratou A, Teletia N, Dewitt DJ, Patel JM, Zhang D (2012) Can the elephants handle the NoSQL onslaught? *Proc VLDB Endow* 5(12):1712–1723
16. Gilbert S, Lynch N (June 2002) Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33(2):51–59
17. Gray J (1978) Notes on database operating systems. In: Gray J (ed) *Operating systems, an advanced course*. Springer-Verlag, London, pp 393–481
18. Gray J, Helland P, O’Neil P, Shasha D (1996) The dangers of replication and a solution. *SIGMOD Rec* 25(2):173–182
19. H-Store: Next Generation OLTP Database Research (2014) <http://hstore.cs.brown.edu>. Accessed 31 Jan 2014
20. Jimenez-Peris R, Patino Martinez M, Kemme B, Perez-Sorrosal F, Serrano D (2009) A system of architectural patterns for scalable, consistent and highly available multi-tier service-oriented infrastructures. *Architecting dependable systems VI*. Springer-Verlag, Berlin, pp 1–23.
21. Kemme B (2000) Database replication for clusters of workstations. PhD thesis, Swiss Federal Institute of Technology, Zurich
22. Kemme B, Alonso G (2000) Don’t be lazy, be consistent: Postgres-R, a new way to implement database replication. In: *Proceedings of the 26th international conference on very large data bases* (San Francisco, CA, USA), VLDB ’00, Morgan Kaufmann Publishers Inc., pp 134–143
23. Kemme B, Alonso G (2000) A new approach to developing and implementing eager database replication protocols. *ACM Trans Database Syst* 25(3):333–379
24. Kemme B, Jimenez-Peris R, Pantino Martinez M, Salas J (2000) Exactly once interaction in a multi-tier architecture. In: *VLDB workshop on design, implementation, and deployment of database replication*
25. Kohana M, Okamoto S, Kamada M, Yonekura T (2010) Dynamic data allocation scheme for multi-server web-based MORPG system. In: *Proceedings of the 2010 IEEE 24th international conference on advanced information networking and applications workshops* (Washington, DC, USA), WAINA ’10, IEEE Computer Society pp 449–454
26. Kohana M, Okamoto S, Kamada M, Yonekura T (2012) Dynamic reallocation rules on multi-server web-based MORPG system. *Int J Grid Utility Comput* 3(2/3):136–144
27. Lamport L (1998) The part-time parliament. *ACM Trans Comput Syst* 16(2):133–169
28. Lerner RM (2010) At the forge: NoSQL? I’d prefer some SQL. *Linux J.* 2010:192. (<http://www.linuxjournal.com/article/10720>. Accessed 31 Jan 2014)
29. Lindsay BG, Selinger PG, Galtieri CA, Gray JN, Lorie R A, Price TG, Putzulo F, Traiger IL, Wade BW (July 1979) Notes on distributed databases. Research Report, IBM Research Laboratory (San Jose, California, USA) 247–284

30. Miller RB (1968) Response time in man-computer conversational transactions. In: Proceedings of the December 9–11, 1968, fall joint computer conference, part I (New York, NY, USA), AFIPS '68 (Fall, part I), ACM pp 267–277
31. MongoDB <http://www.mongodb.org>. Accessed 31 Jan 2014
32. MySQL MySQL Cluster CGE. <http://www.mysql.com/products/cluster>. Accessed 31 Jan 2014
33. Perez-Sorrosal F, Patino Martinez M, Jimenez-Peris R, Kemme B (2007) Consistent and scalable cache replication for multi-tier J2EE applications. In: Proceedings of the ACM/IFIP/USENIX 2007 international conference on Middleware (New York, NY, USA), Middleware '07, Springer-Verlag New York, Inc., pp 328–347
34. Perez-Sorrosal F, Patino Martinez M, Jimenez-Pereis R, Kemme B (2007) Consistent and scalable cache replication for multi-tier J2EE applications. In: Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware (Berlin, Heidelberg), Middleware 2007 Springer-Verlag pp 328–347
35. Perez-Sorrosal F, Patino Martinez M, Jimenez-Peris R, Kemme B (2011) Elastic SI-Cache: consistent and scalable caching in multi-tier architectures. *VLDB J* 20(6):841–865
36. Prichett D (May 2008) BASE: an ACID alternative. *Queue ACM* 6(3):48–55
37. Ramakrishnan R (2012) CAP and Cloud data management. *Computer IEEE* 45(2): 43–49
38. Riak | Basho Technologies (2014) <http://basho.com/riak>. Accessed 31 Jan 2014
39. Schram A, Anderson KM (2012) MySQL to NoSQL: data modeling challenges in supporting scalability. In: Proceedings of the 3rd annual conference on systems, programming, and applications: software for humanity (New York, NY, USA), SPLASH '12, ACM, pp 191–202
40. Skeen D, Stonebraker M (1983) A formal model of crash recovery in a distributed system. *Software engineering. IEEE Trans SE* 9(3): 219–228
41. Stonebraker M (1986) The case for shared nothing. *IEEE Database Eng Bull* 9(1):4–9
42. Stonebraker M (4 Nov 2009) The “NoSQL” discussion has nothing to do with SQL. <http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothing-to-do-with-sql/fulltext>. Accessed 31 Jan 2014
43. Stonebraker M (5 April 2010) Errors in database systems, eventual consistency, and the cap theorem. Blog, Communications of the ACM
44. Stonebraker M (2010) SQL databases v. NoSQL databases. *Commun ACM* 53(4):10–11
45. Vogels W (Oct 2008) Eventually consistent. *Queue ACM* 6(6):14–19
46. Vogels W (2009) Eventually consistent. *Communications of the ACM* 52(1):40–44
47. VoltDB <http://voldb.com>. Accessed 31 Jan 2014
48. Waldo J (2008) Scaling in games and virtual worlds. *Commun ACM* 51(8):38–44
49. Weinreb D Improving the PACELC taxonomy. <http://danweinreb.org/blog/improving-the-pacelc-taxonomy>. Accessed 27 Feb 2013
50. Wikipedia. Consensus (computer science). [http://en.wikipedia.org/wiki/Consensus_\(computer_science\)](http://en.wikipedia.org/wiki/Consensus_(computer_science)). Accessed 31 Jan 2014
51. Wikipedia. Paxos (computer science). [http://en.wikipedia.org/wiki/Paxos_\(computer_science\)](http://en.wikipedia.org/wiki/Paxos_(computer_science)). Accessed 31 Jan 2014
52. Zhang K, Kemme B, Denault A (2008) Persistence in massively multiplayer online games. In: Proceedings of the 7th ACM SIGCOMM workshop on network and system support for games (New York, NY, USA), NetGames' 08, ACM, pp 53–58



<http://www.springer.com/978-3-319-10529-1>

Cloud Computing
Challenges, Limitations and R&D Solutions
Mahmood, Z. (Ed.)
2014, XXI, 352 p. 100 illus., Hardcover
ISBN: 978-3-319-10529-1