Where does ACID go to?

Xing Qu
East China Normal University
51164500235@stu.ecnu.edu.cn

ABSTRACT

As the four basic attributes of the OLTP transactions, *ACID* occupies a very important position and a long history in database field. The use of ACID simplifies the complexity of the transaction to ensure the accuracy and easy to understand. With the NoSQL movement, however, ACID status has been challenged, many of the recent decade proposed database have claimed not to guarantee ACID properties. In this paper, we examine a number of NoSQL and NewSQL data stores to prove that ACID is still important and to explain how each property is implemented.

To do this, we first discuss the the history of ACID and the challenges that ACID suffers and give us reasons why ACID is still important. We then introduce the ACID properties of the implementation in modern database or prototype, respectively.

Keywords

ACID; NoSQL; NewSQL; Transaction;

1. INTRODUCTION

ACID is the four basic properties of a transaction proposed by Jim Gray in the 1970s.That is atomicity, consistency, isolation and durability.

- **Atomicity**: All operations in the entire transaction, or all completed, or all not completed, it is impossible to stop in the middle of a state. An error occurred during the execution of a transaction, which will be rolled back (Rollback) to the state before the transaction begins, as if the transaction had never been executed.

- **Consistency**: The database integrity constraint has not been broken before the transaction is started and after the transaction has ended. It is means that database transactions cannot destroy the integrity of relational data and the consistency of business logic.

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

- **Isolation**: When multiple transactions are executed concurrently, transactions are isolated, and a transaction should not affect the effect of other transactions.

- **Durability**: Once a transaction is committed, it should be persisted and not be lost due to conflicts with other operations or other reasons such as loss of power.

All the time, the database has been designed to follow the golden rules of ACID from System R to MySQL. In recent years, however, a number of new databases forgo the rules of ACID. They claim that traditional relational data is out of date and unable to meet the needs of modern applications. They can not tolerate the database can only be scale up, not scale out. They have a common name *NoSQL* that stands for "Not Only SQL" and "Not Relation Model"[19]. NoSQL systems use *key-value*, *document* and *extensible record* stores instead of relation stores. Most these systems do not supply SQL but use TCP protocol or API. By this way, these systems achieve high availability and scalability. With Web 2.0 booming, these systems have achieved extremely success with high performance, low latency and scalability over traditional relational databases. But this success does not obscure the fact that they can only be used in simple web applications. In complex large-scale business scenarios, such as ordering, banking, They have never been accepted due to not supporting ACID. Ironically, Orace, MySQL and Microsoft SQL Server have been ranked in the top three. On the contrary, even the best performance of the NoSQL system – MongoDB[5], is only ranked fourth in 2016 DB-Enginnes.

ACID is very important.Foremost it simplifies the complexity of data manipulation, programmers can focus attention to the business logic, do not worry about concurrent operations lead to errors.Furthermore, the database is better suited to manipulate data than the application. ACID has been criticized for the great loss of performance. So there are attempts to use the *BASE*[46] instead of ACID. Unfortunately,the BASE transcation is so hard that "Designing applications to cope with concurrency anomalies in their data is very error-prone, time-consuming, and ultimately not worth the performance gains"[48]. In addition, some people who support NoSQL use CAP[17] theory to attack ACID. They argue that CAP theory dictates that ACID systems can not achieve high availability and scalability.In fact,the CAP theory was abused. The author of the CAP theory, Eric Brewer, explain that "the choice between C and A can occur many times within the same system at very fine granularity. not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved"[16]. So

"2 of 3" is missleading. The aim of the theory that designers can achieve some trade-off of all three rather than completely abandoning one. Surprisingly, Not all NoSQL systems abandon ACID.Some NoSQL systems also support ACID transcations,such as ScaleDB[8],Clustrix[3].

In order to distinguish from NoSQL systems, a new data store system called *NewSQL* systems that they can scale out and scale up without sacrificing the ACID properties. These new systems are divided into three categories by Andrew Pavlo and Matthew Aslett[43].That are novel systems that base on new architecture, such as In-Memory and NVM, middleware and database-as-a-service. We will only discuss first category for lack of space in next section. In fact,In-Memory database is not entirely something fresh because almost all key-value stores (they are NoSQL systems) are also based on memory, such as Redis[6],Riak[7].But these systemes try to provide the same scalablity of NoSQL while still maintaining ACID guarantees for transactions. The purpose of this paper is to summarize how these systems (NoSQL and NewSQL) implement ACID guarantees.

The rest of the paper proceeds as follows.Section 2∼5 shed more light on these implementations of the ACID properties,respectively. And the advantages and disadvantages of these implementations are compared. Section 6 concludes the paper.

2. ATOMICITY

To some extent, the atomicity of the transaction is the first piece of the fallen dominoes. A lot of problems arise because of the need to guarantee atomicity. The difficulty of atomicity is to ensure that a series of operations of data are done or not done. To be honest, this is a difficult task, especially in a distributed environment. However, the guarantee of atomicity itself is not hard. The most common way to achieve this is *logging*. Logging can be guaranteed atomicity by *redo* and *undo* operations. At the same time, the log is also used for the guarantee of durablity. This is described in detail in section 5.1. Through redo to ensure that all operations will be done in the event of an accident. In the circumstance that a transaction can not be committed successfully, undo ensures that all operations are undone. It is necessary to declare that both redo and undo are not necessarily required. For example, C-store[49] delete undo log for the guarantee of storage model. In the NVM-based databases[10], redo log is sometimes unnecessary. However, due to the persistence of different media, the problem of the guarantee of *atomicity of writes* is still very complex in NVM-based systems[54, 59].

The logging to ensure atomicity seemed nice enough. But its drawback is that it will seriously sacrifice the performance of the database under the current hardware conditions. Therefore, many NoSQL systems such as key-value system(e.g. Redis[6], Memcached[4]) usually only provide atomicity guarantees to single operation on the single object. And only provide a simple API interface such as *get* and *put*. In this way, a lot of things become very simple, while achieving high performance. A transaction that constrains the operation of data is called a *single-row* transaction. This type of system includes bigtable[21] and Megastore[14]. However, this magic is not really the way to solve the problem. The burden of ensuring atomicity of multiple operations was thrown to the programmer.

The implementation of atomicity usually requires an agreement protocol to ensure that, such as 2PC, Paxos. In order to achieve scalability and high performance, the NewSQL system avoid such an agreement protocol by such as only supporting for local transactions. While Calvin[51] supports scalability by introducing a deterministic locking mechanism. The heart of mechanism is that once the transaction is acquired all the lock transactions will be executed, regardless of any situation. This is very similar to Salt[56], which the first sub transaction is accepted then the entire BASE transaction will be submitted.

3. CONSISTENCY

3.1 Consistency in ACID and CAP

Before discussing the consistency of ACID, we first clarify some concepts. Due to ACID and CAP have a "C", it is easy to cause misunderstanding. Although the names are the same, the two consistency semantics are different. The consistency of ACID refer to data integrity and *business consistency* in database, while the consistency of CAP is to distribute data among multiple servers in order to obtain the same data, which is a kind of distributed domain *data consistency* concept. Or more simply to understand the consistency of the transaction is to ensure the consistency of the overall relationship between different objects, such as the total amount of the two is unchanged after the transfer. The consistency of data is guaranteed to be consistent between replicas. However, they are not completely irrelevant. The consistency of ACID implementation must ensure the realization of the consistency of CAP. Due to the application of the driver, the database needs to be scalable must be distributed. Therefore, the problem of data consistency in distributed environment is presented[23]. And in a distributed environment, transaction consistency is also a very difficult problem.

3.2 2-PC

For a stand-alone system, the consistency of transactions can be guaranteed by atomicity, isolation and persistence mechanisms. In distributed environment, however, the database is usually introduced into the *two-phase commit protocol*[27, 40] (2PC) to ensure the consistency of the transaction. In the two phase commit protocol, the system generally consists of two types of servers (or nodes): one for the coordinator, usually in a system with only one; the other is transaction participants (or workers), usually contains more than one. The two-phase commit protocol consists of two phases that performed as follows:

- Phase 1 (commit-request phase): The coordinator will notify participants prepared to submit or cancel the transaction. Then the participants will inform the coordinator of their decision: agree (transaction participant local task execution success) or cancel (local task execution failure).

- Phase 2 (commit phase): The coordinator will make decisions based on the results of the first phase of the vote: submit or cancel. If and only if all the participants agree to submit the transaction coordinator to inform the participants to commit the transaction, otherwise the coordinator will notify all participants to cancel the transaction. The participants in the re-

ceived message sent to the coordinator will perform the operation in response to.

In fact, there are some variants of the two phase commit,such as tree 2PC protocol[55], D2PC[47]. But we do not introduce them due to the limited space. The disadvantage of 2PC is that the execution of the protocol is blocked. When participants occupy public resources, the other third party access to public resources have to be in a state of blocking. And once the coordinator fails, the participants would have been blocked down. Especially in the second phase, the coordinator fails, then all of the participants are in a state of affairs locked resources, unable to continue to complete the transaction. A more serious situation is presented in the two stage of the second stage, when participants send commit request to the coordinator, the local network anomaly in or send commit request process coordinator fails. This results in only a fraction of the participants receiving the commit request. In this part of the participants received the commit request will be executed after the commit operation. However, other parts of the participants that did not receive the commit request could not commit the transaction. So the whole distributed system has the phenomenon of inconsistency.

In order to solve the above problems, it was proposed that the *three phase commit* (3PC)[31]. 3PC introduced the timeout mechanism in the coordinator and participants. And inserting a preparation phase in the first and second stages. It ensures that the states of the participating nodes are consistent before the final commit phase.

3.3 Data Consistency

Strictly speaking, data consistency does not belong to transaction consistency. But the modern database, especially the NoSQL systems use the *eventual consistency*[50, 44, 37, 13, 53] to replace 2PC. So, let's briefly introduce the data consistency in this subsection. Data consistency can be divided into strong consistency and weak consistency. The eventual consistency is a kind of weak consistency.

Strong consistency is achieved by *consistency by writes*, *consistency by reads* and *consistency by Quorum*(R+W>N). (1)Consistency by writes: Writing and copying the data to other servers, read any one can get new write data. (2)Consistency by reads: Once the server is written, it is no longer copied, but the version is used to coordinate replication (e.g., the vector clock algorithm) in reading, which simplifies the write operation and increases the burden of the read operation. (3)Consistency by Quorum: Copying on write data to other 2/3 servers and reading from the 2/3 servers, reading determine which is the latest update results, which share the burden in reading and writing. It is not enough to have a strong consistency in a distributed system. *Linearizable consistency* is a stronger consistency, more similar with the consistency of transactions. That is concurrent execution of transactions, no matter how concurrent, as if they are executed independently, then the final result is always the same. There are two ways to realize this kind of distributed linearizable consistency consistency: 2PC and Paxos[35]. The pursuit of strong consistency of the database is represented by Spanner[22] and F1[48].

Strong consistency usually makes the data throughput is not high. Therefore, many databases use weak consistency, such as Dynamo[25]. There are many types of weak consistency, such as *causal consistency*, *session consistency*, *sequential consistency*, etc. In order to save space, more consistency(e.g. RedBlue consistency[36]) can refer to this review[53]. The advantage of using weak consistency is that you can quickly read and write a local copy. However, different copies may have write conflicts. As a matter of fact, the eventual consistency is somewhat unsafe[11]. But in many scenarios the system does not depend on whether it is safe, so the eventual consistency is appropriate.

3.4 Summary

Consistency is a very big topic, especially in Distributed Systems. In most of the time, the consistency of the database refers to the consistency of the data. Therefore, we briefly put forward the contents of the data consistency in this section. However, we need to note that there is a certain relationship between the consistency of the transaction and the consistency of the data. In order to better scalability and performance, data consistency in the NoSQL systems is usually weakened. This weakening is appropriate in some situations. But that does not mean consistency and performance can't be together. In fact, we can have a lot of external means to achieve high performance without losing consistency. Therefore, it is still necessary to ensure consistency of the database.

4. ISOLATION

4.1 Isolation Levels

Isolation is the visibility of concurrent transactions. The most widely used isolation levels are defined by ANSI/ISO, including (1) READ UNCOMMITTED, (2) READ COMMITTED, (3) REPEATABLE READ, (4) SERIALIZABLE. However, these isolation levels all rely on locking to define. In fact, the isolation levels defination also can be rely on degree of consistency,such as *snapshot isolation*[15]. Besides, many databases have their own defined isolation levels, such as *salt isolation*[56]. In simple terms, isolation is guaranteed by the *concurrency control*[20] mechanism of the database. From the point of view, the concurrency control mechanism can be divided into pessimistic and optimistic concurrency control(OCC)[32]. Pessimistic concurrency strategy is actually based on locks, while optimistic concurrency control without using locks. It is important to note that there is an *optimistic lock* based on *multi-version concurrency control* (MVCC)[39].

The performance constraints on the execution of transactions by four standard isolation levels is gradually serious from read uncommitted to serialization. Therefore, not all databases support serialization in a database that supports ACID[12]. In fact, the default isolation level for most databases is read commited,such as Oracle. It seems that the use of weak isolation levels is an obvious trend. But this is only for the performance of the choice of the helpless way.

It must be noted that isolation levels can coexist simultaneously. Taking MySQL as an example, different sessions connected to the same database can be set to different isolation levels. The session A is set to read commited, while the session B is set to read uncommitted. The session A uncommitted transactions can be read by the session B. However, the session A cannot read uncommitted transactions of the session B. Indeed, they guarantee the semantics of the isolation level of corresponding session. There is still a write lock on the same level. Writing the same row of the two

read uncommitted, the Later transaction is blocked before the previous one has been submitted. After submitting one of the transactions, the transaction of seession A can read the submitted written, but not submitted to write, still can not see.

4.1.1 Isolation of NoSQL

The isolation of the NoSQL systems is simpler than the SQL systems. Taking mongoDB as an example, it only suport "serialization". For a single instance of mongod, the write operation that takes place on the same document is serialized, and read is serialized. Therefore, serialization of the NoSQL systems is not the serialization of the SQL system. It is just the isolation of a single object, but can not isolate multiple objects. Isolation of multiple objects can be described as read uncommitted. Reasonably,the isolation of the NoSQL systems is rooted in its atomicity. In the isolation of individual objects, there are other risks in In-Memory databases. For example, read operation is possible to see those data has not been submitted to the disk. Due to the failure of replica primary in set, failover or the entire cluster failure (loss of power), the read operation previously seen in the data may be "roll back", that is, data will no longer exist; Semantic equivalent "not repeatable read" in the transaction.

4.1.2 Salt Isolation

Salt isolation is the two kind of transaction isolation guarantee in Salt[56]. Salt according to the 80/20 principle, BASE-ify some critical ACID transactions to achieve close to the high throughput of the BASE system while programming as easy as the ACID system. Salt isolation can ensure that ACID transactions and BASE transactions coexist in the Salt and are not affected each other. In other words,the ACID transaction believes that the BASE transaction is a ACID transaction. Meanwhile a BASE transcation can expose the intermediate state to another. This is very useful for salt to obtain high throughput because it can get better concurrency. In order to realize salt isolation, salt introduces three special locks: ACID lock, alkaline lock and saline lock. Table 1 shows the mutex matrix of the three locks. The ACID lock is a lock that the ACID transaction adds to the operating object. the alkaline lock is a lock that the substransaction of BASE transaction adds to the operating object. The saline lock is a downgrade of the alkaline lock. Its role is to prevent ACID transactions from accessing to the intermediate state of BASE transactions. The saline lock is not released until the entire BASE transaction is committed.

Table 1: Conflict table for Salt's locks

	ACID Lock	Alkaline Lock	Saline Lock
ACID Lock	X	X	X
Alkaline Lock	X	X	O
Saline Lock	X	O	O

4.2 Concurrency Control Based on Locking

In this subsection, we discuss the implementation of the standard isolation levels based on locking. The concurrency control Base on Locking is most prevalent implementation choice,including Oracle, MySQL, MongoDB.

Declare : α and β are both transactions. A is a table.

4.2.1 Read Uncommited

Locking:

1. Transaction does not lock the currently read data;
2. Transactions in the update of a data moment must first *row-level shared* lock, until the end of the transaction before the release.

Result:

1. When α read a line of record, the β can also read and update this line of records;
2. When β updates the record, α reads the record again and reads a modified version of β for that record, even if the modification has not yet been committed;
3. When α updates a row, β can not update this row until α ends.

4.2.2 Read Commited

Locking:

1. The transaction adds a *row-sharing* lock to the currently-read data (locks when read) and releases the *row-level shared* lock as soon as the row is read;
2. The transaction in the update of a data at the moment (that is, the moment of the update), must first *row-level exclusive* lock, until the end of the transaction before the release.

Result:

1. When α read a line of record, β can also read and update this line of records;
2. When β updates the record, α reads the record again, reading only the version of β prior to its update, or the version of β after commit.
3. When α updates a row, β can not update this row until α ends.

4.2.3 Repeatable Read

Locking:

1. The transaction in the moment of reading a data (that is, the moment to start reading), must first add its *row-level shared* lock, until the end of the transaction before the release;
2. The transaction in the update of a data at the moment (that is, the moment of the update), must first *row-level exclusive* lock, until the end of the transaction before the release.

Result:

1. When α reads a row, β can also read and update the record;
2. When β updates the record, α reads the record again and reads the first read take that version.
3. When α updates a row, βcan not update this row until αends.

4.2.4 Serialible

Locking:

1. The transaction in the read data must first add a *table-level shared* lock, until the end of the transaction before the release;
2. The transaction in the update data must first add a *table-level exclusive lock*, until the end of the transaction before the release.

Result:

1. When α is reading the records in A, β can read A, but can not update, add, or delete A until α ends.
2. When α is updating the records in A, then β can not read any records in A, and it is impossible to update, add, and delete A until α ends.

4.3 Optimistic Concurrency Control

The optimistic concurrency control[32] (OCC) is a concurrency control strategy without lock. OCC backup transactions in advance, when the conflict happened on between the commit of transactions, the rollback to ensure transaction isolation. Of course, OCC most want is that there is no conflict between transactions. Optimistic concurrency control divides the transaction into three phases: read, verify, and write. In the reading stage, writing the data to a local transaction buffer, there will not be any check operation. In the process of checking, the system will synchronize checking on all matters. In the writing stage, the data will be submitted to the final. After the completion of the read phase, the system will assign a timestamp to each transaction.

The optimistic concurrency control is usually used in some NoSQL systems,such as Redis, Silo[52], TicToc[58]. In Redis, the use of *check-and-set* operation to achieve optimistic lock. the check-and-set operation specific command is *WATCH* in Redis. Some keys are monitored by the watch command and the command detects whether the monitored keys are changed before the transaction is executed. If at least one of the monitored keys has been modified before execution, the entire transaction is canceled, and the return transaction to the client has failed. When the failure occurs, Redis will continue to repeat this transaction until success. The retry policy is acceptable. Because most of the time, different clients will have access to different keys, and the collision is usually very small, so there is no need to retry.

Silo is a single node in-memory OLTP database. It use *serializable OCC* with parallel backward validation. Its key technology lies in batched timestamp allocation using *epochs*. In Silo, the time is divided into fixed epochs. All transactions that start at the same epoch are submitted together after the end of a epoch. Due to the use of batch processing technology, Silo has a very high performance in the case of less cross partition transactions. Instead of using the global timestamp, TicToc uses the timestamp from records. Record timestamps include *write timestamp* and *read timestamp*. In fact, the timestamp is that the logical timestamp of the last transaction that read or write the record. In the verification phase, TicToc also contains three steps: (1)Lock Write Set, (2)Compute Commit timestamp, and (3)Validate Read Set. The logical timestamp of the read transaction is updated before it is committed, and the transaction is rolled back. Otherwise, the read timestamp is the latest version or update occurs after the transaction is committed, so the transaction can be submitted.

4.4 Multi-Version Concurrency Control

Multi-Version Concurrency Control[39] (MVCC) is a very useful concurrency control method, which can prevent the mutual blocking between read and write transactions. For read-only transactions, the use of MVCC can be used to avoid locking records. Since there is no need for locking, the degree of concurrency can be greatly improved. As a result, MVCC almost has become the standard configuration of many mature databases, such as Oracle, Microsoft SQL Server, MySQL, SAP HANA. In each database, MVCC can be implemented in different ways, but it can be divided into two ways: locking and timestamp.

4.4.1 MVCC Base on Timestamp

MySQL's InnoDB transaction engine implementation of MVCC is based on timestamp. In InnoDB, each record has two additional values, one of which is a record of when the data is created, and the other is a record of when the data is expired (or deleted). However, InnoDB does not store the actual time when these events occur, instead it only stores the system version number at which the events occurred. This is a growing number with the creation of transactions. Each transaction records its own system version number at the beginning of the transaction.

An example is given to illustrate the mechanism of MVCC. For *select* operation, the result of a query meet that a row version must be at least as old as the transaction Version (i.e., its version number is not greater than the transaction version number) and the deleted version of this row must be undefined or larger than the transaction version. The above two conditions ensure that the data is present before the transaction is started, or when the transaction is created, or when the data is modified. It will record the current system version number when inserting a line. Similarly, the system version number is recorded when a row is deleted. For update operation, it will write a new copy of the data, which is the current version number. It also writes the version number to the deleted version of the old line. The result of this extra record is that there is no need to obtain a lock for most queries. They simply read the data as quickly as possible to ensure that only the rows that meet the criteria are selected. The drawback of this solution is that the storage engine has to store more data for each line, do more checks, and deal with the aftermath.

There is also a need to point out that MVCC can not work at all isolation levels, it can only work at REPEAT READ and READ COMMIT isolation level. READ UNCOMMITED and MVCC are incompatible because the query cannot find a row version that is appropriate for their transactional version. They can only read the latest version at a time. Nevertheless, the MVCC based on timestamp is still very popular, such as SAP HANA[26].

4.4.2 MVCC Base on Locking

In the early the MVCC base in locking[39], multiple versions are stored in the stack, proposing 2 version of the algorithm, the 3 version of the algorithm and N version of the algorithm. However, this method is limited by the stack space leading to the presence of blocking. The implementation of the modern of the MVCC base on locking has been

greatly different.

the MVCC base on locking is introduced using the Oracle as an example. In Oracle[33], the start of the transaction to modify the data before, it must obtain a transaction slot and allocation of space in the rollback segment. And creating the mirror of the data, and then the same transaction information is recorded in a ITL, then modify the transaction to continue. Oracle, in order to, ensure that the transaction is to back off. When a transaction commits, the transaction is marked as a commit in the rollback segment. At this time the corresponding version of the SCN created in the submission of data. When multiple transactions successfully modify the same data, there are rows in a different version of the SCN in the rollback segment. When a read operation reads a block of data, it is found that the data block has a modified trace (the lock flag points to ITL). It will need to be checked by SCN to read the current data or modify the data before, to ensure that all the data read at the beginning of the transaction data to ensure the consistency of the data. This is the consistency of the Oracle statement level read, if you want to get the transaction level read consistency to prevent non-repeatable read and phantom read must use *Serialable* or *Read Only* isolation level.

In fact, most databases implement only snapshot isolation (SI) instead of full serializability by MVCC. Serialization implementation in most databases is still done through the standard two-phase locking mechanism. Even many popular databases do not provide serialization isolation levels[12],such as SAP HANA, MemSQL. Recently, however, the research has been implemented on MVCC by *Serializable Snapshot Isolation*[18, 45] (SSI) technology. The basic principle of SSI is similar to the serial graph testing. Through the establishment of dependency graph between transactions and detecting whether there is a conflict to achieve serialization. And a special *SIREAD lock* is introduced in order to track read dependencies.

Another way[41] to achieve the MVCC serialization isolation level is to use an *update-in-place* version mechanism and *precision locking*[28]. Incremental data is stored as a linked list in *Version Vector*. In fact, incremental data is the data of the different version that is generated in the order of transaction start time. In addition, in order to achieve high scanning performance, a range is maintained to record the start and end positions of the rows modified by the transaction.

4.5 Modular Concurrency Control

MCC, the *Modular Concurrency Control*, is a kind concurrency control strategy based on modular in Callas[57]. The main idea of MCC is to divide the transaction into multiple groups as much as possible. Each group adopts its own isolation property or concurrency control strategy. Through this refinement of the transaction concurrency control strategy to obtain better throughput. The MCC does not restrict the type of transaction, nor does it limit whether it is a predefined transaction. For non-predefined transactions (interactive transactions), will be placed in the same group, the use of standard concurrency control mechanism. MCC aims to allow different types of transactions can choose the appropriate concurrency control mechanisms to achieve better performance. MCC to solve the problem is to meet the original ACID transaction isolation properties and correctness. In fact,the MCC achieves correctness by ensuring the isolation of transactions within and between groups.

4.6 Summary

Isolation is an important concept in the database, which ensures the correctness of the transaction. The implementation of isolation is closely related to concurrency control in the database. In this section, we introduce the basic principles of concurrency control based on locking and unlocking. The implementation based on locking is the most widely used and mature way. But the higher the isolation level, the higher the cost. The optimistic concurrency control can achieve very high concurrency due to the avoidance of locking and the cost of holding a lock for a long time. However, it is only suitable for the situation where the competition rate is very low. Therefore, in order to obtain high concurrency, database researchers introduce multi-version concurrency control mechanism. The MVCC is not the only way to achieve, so the final effect is also different. In order to achieve the true serialible isolation level, we introduce the implementation of two kinds of Serialible isolation level on MVCC. Finally, a novel way that allow multiple concurrency control mechanisms to coexist with the modular concurrency control mechanism is also presented. We believe that this approach is worth encouraging, because *"learn from others strong points and close the gap"* is a mature database should have quality. Relying solely on a concurrency control mechanism is clearly becoming less and less able to meet the increasing richness of the types of applications.

5. DURABILITY

Durablity is vary fundamental guaranty. But it alway is a place that has plenty of detractors. As we all know, the disk speed is slowest than memory and CPU. However disk is the basic persistent equipment for database. As a result, the disk speed is attacked as the bottleneck of the database.

5.1 Logging

Fristly, *logging* is is the basic means of database persistence. Usually, it is too expensive to operate the data that read and write directly from the disk.

5.1.1 Write Ahead Logging

Accordingly, almost all databases use *write-ahead logging* (WAL)[39] technology, such as MySQL, PostgreSQL, Oracle. Logging to disk before the transaction is committed. By this way, the transaction already be persisted. When the database is restored, it will *redo* or *undo* log to maintain the correct state before the server downtime. Further more, supporting durable updates, WAL has three implement way: in-place updates, copy-on-write updates, and log-structured updates[10].

1. in-place updates: The first update method is a direct way to update columns on the original tuple. This approach has the advantage of high efficiency, no need to copy the tuple, only to update the necessary columns. VoltDB is using this method. But the main problem with this approach is that cause a lot of data backups in memery. Because each modification will record the contents of the modified tuple to WAL. And taking into account the cost of scattered random write too high, it will be a group of ways to regularly flush the

contents of the WAL to disk. Group commit, however, results in an increase in the average transaction response delay.

2. copy-on-write updates: In fact, this method does not use WAL. Copy-on-write creates a copy of the tuple when modified, then modifies it on the backup. Because copy-on-write does not modify the data that was already committed, so it does not require a WAL for recovery. Correspondingly, it provides a different directory to access different versions of the tuple in the database. This method is called the hidden page (shadow paging) in System R. The current directory are used to access the contents of the recently committed transaction, while the dirty directory refers to the modification of the uncommitted transaction. In order to isolate the uncommitted transactions, the storage engine maintain a master record that always points to the current directory. When the writing modified content, dirty directory point to the new dirty tuples. When the transaction is committed, the master record directory point to dirty directory. copy-on-write isn't need undo operate. When the dirty directory is not commited, it will not affect the current directory. And by the garbage collection mechanism, it can be destroyed after the dirty directory after reboot. As it is in accordance with the block operation, each time it will be flush the block into the disk, there is a serious IO overhead.

3. log-structured updates: This is a very popular practice. This approach is used in many database such as C-stroe[49],bigTable[21]. Obviously, this approach originated from *log-structured merge* (LSM) trees[42]. Log-structured updates different from in-place updates in that it has similar data structure to organize data in memery and disk. Separation of read and write is an obvious feature of modern database. For example, BigTable's MemTable and SSTable accept write and read requests, respectively. Through the regular memory of the MemTable flush into disk merge with SSTable to complete the persistence. Of course, it also need to log of MemTable to maintain the correct MemTable. Log-structured updates can reduce the number of update for database in disk. An obvious drawback is that it is optimized to write, but not friendly to the read operation. Because read operations must read data from Memtable and SSTable.

5.1.2 Log in NoSqL Systems

For the key-value systems, log is also required. For example, MongoDB's log mechanism is called *journal* and *write concern*. First, update of the data in MongoDB written to the Journal Buffer and then update the memory data, and then return to the application side. Journal will be 100ms interval batch brush to the disk. Write concern is a unique feature of MongoDB. This is a trade-off between performance and reliability. The following levels of write concern: Unacknowledged, Acknowledged, Journaled and Write to most nodes.

1. Unacknowledged: Unacknowledged refers to each write operation, MongoDB does not return a successful state. This level is the best performance, but also the most unsafe level.

2. Acknowledged: Acknowledged means that every write MongoDB will confirm the completion of the operation, regardless of success or failure. Observingly, this confirmation is based only on the memory of the primary node,not disk.

3. Journaled: The use of Journaled means that each write operation will be not return util journal flushed into disk. MongoDB will not immediately brush for each operation, but will wait up to 30ms, the 30ms of write operations together to use sequential addition of the way into the disk. Thus the overall response time for a single operation will be extended, but for highly concurrent scenarios, the average throughput and response time will not be much affected.

4. Majority: Majority means most nodes. With this write security level, MongoDB will return to the client only if the data has been copied to the majority of the nodes. It is clear that this approach is the most secure but the worst performance due to the presence of multiple nodes' communication with each other.

For the In-Memory databases, log is one of persistent means, such as Redis[6], VoltDB[9]. In addition, they used snapshot technology. This technique is discussed in the next subsection. Redis'AOF records all write operations performed by the client and restores the dataset by re-executing these commands at server startup. Using AOF persistence can make Redis very durable. It can set different fsync policies, such as no fsync, fsync per second, or each time the command of write to fsync. Redis can rewrite AOF automatically in the background when the AOF file size becomes too large. The rewritten new AOF file contains the minimum set of commands required to restore the current data set. Generally speaking, the memory database does not contain *undo log* due to volatile of memory.

5.1.3 Recovery

The log is used to recovery databases. However, traditional log recovery methods are very inefficient. Early database running on expensive hardware, in case of a hardware failure, need to wait for a hardware reset, loading the operating system and database, and then recover to the final state of database through the redo and undo log. The whole process need to wait for a few minutes to an hour or more. The main reason for the slow recovery of the database is that the log is usually appended to the file in sequence, resulting in replaying each a log only in sequential. In order to handle this issue, SiloR[60] use a new way to fast durability and recovery through multicore parallelism. SiloR use value logging rather than operation logging. Value logging contain complete record, not operation and parameters. Thus, SiloR can replay in parallel due to only replay operation of last modified same tuple. Apparently,it has some disadvantages that include more the cost of space and IO, slower transcation execution. In siloR, a series of workers and loggers concurrent generation logs. Recovery requires all log entries. The recovery process includes the recovery of checkpoints and logs. On the contrary, another recovery technique is called *command logging*[38]. By using a coarse granularity log, it sacrifices the recovery time while obtaining greater availability.

5.2 Snapshot

The snapshot is similar to the checkpoint. They are all able to give the log a range of bounds for recovery. However,the snapshot contains the state of the entire database before a certain period of time. For example, Redis's RDB can generate a point-in-time snapshot of a dataset at a specified time interval. Snapshot is very suitable for disaster recovery. It is only one file, and the contents are very compact. It is easy to copy and send other datacentor. But it also has limition of database size. If the size of the data is too large, such as dozens, hundreds of terabytes or even petabytes of data, the cost of a full database snapshot is too expensive. Therefore, an optional method is an incremental snapshot. Only tuples that are different from the previous snapshot are saved at a time.

5.3 Duplicate

Hardware redundancy is acceptable as the hardware at the beginning of the database is relatively inexpensive. Therefore, in order to achieve such high-availability, the current database take the strategy that is hardware redundancy and data backup. At the same time, due to data redundancy, bringing greater throughput. Cassandra[34] is also through the backup to achieve high-availability and data persistence. Each Key-value pair in Cassandra is backed up to N nodes, where N is a configurable item. Each key is assigned to a coordinator who is responsible for the replication of the Key-value pair within its range of data. In addition to coordinator store this key corresponds to value, it will copy N-1 replicas of data to one nod of ring. And the client can choose what kind of data needs to be backed up. The choice of N-1 nodes has three backup strategies,Rack Unaware, Rack Aware and Datacenter Shard. The Rack Unaware strategy is the simplest and does not consider the actual physical location of the nodes. The Rack Aware strategy considers racks, storing N-1 datas in different racks in the data center while another data node on different data centers. The Datacenter Shard strategy selects the number of data backups held in each datacenter according to the configuration. At the same time each node will cache the coordinator within the scope of the node. Once the coordinator failure, it can choose as coordinator from N-1 successors.

5.4 NVM

In addition to the persistence of software, many databases are using flush and *non-volatile memory* (NVM) to replace the disk as a persistent medium. Non-volatile memory access speed significantly faster than the disk, slightly lower than the DARM. But it can be sustained after the power off, the capacity smaller than the disk, while much larger than the memory. But the current database architecture is not ideal to NVM that a new architecture design is necessary[24].

NVM-aware engines[10] are three implementions of storage engines that build on NVM. NVM-InP engine depend on In-Place Updates engine. When a tuple is inserted, NVM-InP engine is not copied into the WAL, but rather a pointer to the tuple is recorded. Thus, it avoid a lot of data backups. It also maintains a persistent state in the header of each slot. The durability state include unallocated, allocated but not persisted, or persisted. the slots of allocated but uncommitted become unassigned after reboot. NVM-InP's recovery is mainly to roll back the uncommitted transactions successfully, because the committed transaction is persisted immediately after the commit, but the transaction that did not commit successfully needs to be rolled back by the WAL. NVM-CoW engine depend on Copy-on-Write Updates Engine. NVM-CoW directly persistence tuples and pointers of dirty directories. It can reduce overhead of IO that flush operation block to disk. Compared with traditional methods, NVM-log can avoid redundant data in MemTable and WAL. WAL only records the modified tuple pointer. And no longer need to brush MemTable to disk with SSTable merger. Just need to mark MemTable as ImmemTable.

5.5 Cloud storage

With the rise of cloud, many database systems deployed in cloud storage systems. It become a new power that can not be ignored, such as Amazon S3[2] and EBS[1]. These databases are called CloudDB or RDS(Relation database Service). AWS S3(Simple Storage Service) provide an high-available storage service for the Internet application. S3's access way is through HTTP/S. The advantage of S3 are scalable, high availability, inexpensive. AWS EBS(Elastic Block Store) used for persisent storage. EBS volums as virtual disks. Similarly,the most advantage of EBS is that it can be scalable. At the same time, it simplifies the persistence layer of the application. However, the problem of cloud storage systems may be that the access speed is limited by the network conditions.

Apparently, a database with a reliable persistence storage is significantly different from a traditional database in architecture. SHADOW[30] system reconsider the hot standby technique in the cloud storage. Taking into account the database in the cloud storage environment, the database needs to be accessed by multiple servers. the active and standby database systems of SHADOW system share a persistent storage system. In the SHADOW system, DBMS updates the data in the local cache. And log saved in shared persistent storage system. The standby database systems updates its cache based on the log in persistent storage system. Persistent storage systems backup logs and databases to prevent data loss. The SHADOW system has the following advantages: (1)The persistence mechanism is pushed into the persistent storage system to simplify the persistence mechanism of the database. (2)No explicit synchronization between active and standby database. (3)Due to replication occurs only in persistent systems, there is less bandwidth requirements. But the SHADOW system also has shortcomings. (1)The SHADOW system is not really high available due to standby system needs time to replay history log. (2)The speed based cloud storage is difficult to guarantee. (3)The SHADOW system does not provide a distributed extension mechanism.

5.6 Summary

In this section, we discuss the implemention of durablity. Logging is the most ordinary and foundational way. We introduce three implementations of WAL. Then, we introduce the snapshot technique combined with logging. Besides, persistence based on new hardware(NVM) is also discussed. Lastly, we focus on the SHADOW system based on shared persistence system.

We believe that the direction of future persistence may be these two directions: NVM and cloud storage. The NVM system is used to store WAL logs instead of data due to the high cost and limited capacity of NVM hardware. For appli-

cations that do not care much about the speed of response, cloud storage systems may gain a dominant position.

6. CONCLUSIONS

In this paper, we discuss a very basic concept, the basic properties of transactions, namely ACID. Although many NoSQL systems do not support ACID, we believe that supporting transactions is still very important in modern databases. In this paper, we introduce the implementation of ACID in various databases and prototypes. Based on the above trends of modern database, we can draw a conclusion that the database should have more and flexible ways to deal with more and more applications. "one size fit all" is not desirable, but "one size own many" can be achieved. To refine the scene, there is a strong potential to tap in the derivation of the means for these scenarios. Modern application scenarios are becoming more and more complex, so it is difficult to obtain a larger living space for these databases that can handle only certain scenarios. This has been proved by SAP HANA[26] and HyPer[29].

7. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and what have you. In the present case, for example, the authors would like to thank Gerald Murray of ACM for his help in codifying this *Author's Guide* and the **.cls** and **.tex** files that it describes.

8. REFERENCES

[1] Amazon ebs. https://aws.amazon.com/ebs/.
[2] Amazon s3. https://aws.amazon.com/s3/.
[3] Clustrix. http://www.clustrix.com/.
[4] Memcached. http://memcached.org/.
[5] Mongodb. https://www.mongodb.com/.
[6] Redis. https://redis.io/.
[7] Riak. http://docs.basho.com/riak/.
[8] Scaledb. http://www.scaledb.com/.
[9] Voltdb. https://docs.voltdb.com/.
[10] J. Arulraj and A. Pavlo. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 16(1):707–722, 2015.
[11] P. Bailis. Safety and liveness: Eventual consistency is not safe. http://www.bailis.org/blog/safety-and-liveness-eventual-consistency-is-not-safe/.
[12] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *Vldb '13*, 7(3):181–192, 2013.
[13] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Quantifying eventual consistency with PBS. *VLDB Journal*, 23(2):279–302, 2014.
[14] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 223–234, 2011.
[15] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data - SIGMOD '95*, (June):1–10, 2007.
[16] E. Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.
[17] E. A. Brewer. Towards robust distributed systems (abstract). page 7, 2000.
[18] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems*, 34(4):1–42, 2008.
[19] R. Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12, 2011.
[20] W. Cellary, E. Gelenbe, and T. Morzy. *Concurrency control in distributed database systems*. Elsevier Science Inc., 1988.
[21] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable. *ACM Transactions on Computer Systems*, 26(2):1–26, 2008.
[22] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner : Google's Globally-Distributed Database. pages 251–264, 2012.
[23] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys*, 17(3):341–370, 1985.
[24] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. B. Zdonik, and S. Dulloor. A Prolegomenon on OLTP Database Systems for Non-Volatile Memory. *Amds@Vldb*, pages 57–63, 2014.
[25] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. *Proceedings of the Symposium on Operating Systems Principles*, pages 205–220, 2007.
[26] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA Database - Data Management for Modern Business Applications. *ACM Sigmod Record*, 40(4):45–51, 2012.
[27] J. N. Gray. *Notes on data base operating systems*. Springer Berlin Heidelberg, 1978.
[28] J. R. Jordan, J. Banerjee, and R. B. Batman. Precision locks. In *ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, April 29 - May*, pages 143–147, 1981.
[29] A. Kemper and T. Neumann. HyPer : A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. *Icde*, pages 195–206, 2011.
[30] J. Kim, K. Salem, K. Daudjee, A. Aboulnaga, and X. Pan. Database high availability using SHADOW systems. *Proceedings of the Sixth ACM Symposium on Cloud Computing - SoCC '15*, pages 209–221, 2015.
[31] N. Kumar, L. Sahoo, and A. Kumar. *Design and implementation of Three Phase Commit Protocol*

(3PC) algorithm. 2014.

[32] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[33] T. Kyte. *Expert Oracle Database Architecture: Oracle Database Programming 9i, 10g, and 11g Techniques and Solutions, Second Edition*. Apress, 2010.

[34] A. Lakshman and P. Malik. Cassandra. *ACM SIGOPS Operating Systems Review*, 44(2):35, 2010.

[35] L. Lamport. Paxos Made Simple. 2001.

[36] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguic, and R. Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 265–278, 2012.

[37] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. *Proceedings of the Symposium on Operating Systems Principles*, pages 1–16, 2011.

[38] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. *Proceedings - International Conference on Data Engineering*, pages 604–615, 2014.

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

[40] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. pages 76–88, 1983.

[41] T. Neumann and A. Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 677–689, 2015.

[42] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[43] A. Pavlo and M. Aslett. What's Really New with NewSQL? *SIGMOD Record*, 45(2):45–55, 2016.

[44] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. *ACM SIGOPS Operating Systems Review*, 31(5):288–301, 1997.

[45] D. R. K. Ports and K. Grittner. Serializable Snapshot Isolation in PostgreSQL. *Proceedings of the VLDB Endowment*, 5(12):1850–1861, 2012.

[46] D. Pritchett. Base: an Acid Alternative. *Queue*, 6(3):48–55, 2008.

[47] Y. Raz. The Dynamic Two Phase Commitment (D2PC) protocol. *Proceedings of the 5th International Conference on Database Theory (ICDT '95)*, 893:162–176, 1995.

[48] J. Shute, R. Vingralek, B. Samwel, and I. Rae. F1: A distributed SQL database that scales. *Proceedings of the ...*, 6(11):1068–1079, 2013.

[49] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. *VLDB Conference*, pages 553–564, 2005.

[50] D. B. Terry, M. M. Theimer, K. Petersen, a. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *Proceedings of the fifteenth ACM symposium on Operating systems principles - SOSP '95*, 29(5):172–182, 1995.

[51] A. Thomson, T. Diamond, and S. Weng. Calvin: fast distributed transactions for partitioned database systems. *Sigmod '12*, pages 1–12, 2012.

[52] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, pages 18–32, 2013.

[53] P. Viotti and M. Vukolić. Consistency in Non-Transactional Distributed Storage Systems. *ACM Computing Surveys*, 49(1):1–34, 2016.

[54] T. Wang and R. Johnson. Scalable Logging through Emerging Non-Volatile Memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, 2014.

[55] G. Weikum and G. Vossen. Transactional information systems. *Morgan Kaufmann*, page 791âĂŞ827, 2001.

[56] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt : Combining ACID and BASE in a Distributed Database. *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, 2014.

[57] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance ACID via modular concurrency control. *Symposium on Operating Systems Principles (SOSP)*, pages 279–294, 2015.

[58] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. TicToc. *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*, pages 1629–1642, 2016.

[59] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015.

[60] W. Zheng, S. Tu, E. Kohler, and B. Liskov. SiloR: Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. *Osdi '14*, pages 465–477, 2014.