Table of Contents

Preface

Introduction

In this paper, we will describe a way to implement a fully non-blocking but still strongly consistent and distributed datastore.

It:

- Provides strong, ANSI SQL-compatible consistency guarantees;
- Is distributable across multiple computers, datacentres and even geographical locations;
- Can have a resilient, fault-tolerant architecture, where even failures of multiple different datacentres can allow the operation of the datastore to continue seamlessly, depending on the configuration.

This is presented through the idea that sets of resulting data from multiple computers do not necessarily have to be consistent immediately, but that they can be filtered before the results are presented to the client. When looked at this way, the problem can be reduced to making sure that the filtering algorithm references an atomic event which can be deterministically compared to others, and it will result in the same answer anywhere in the system.

The document outlines:

- A fully optimistic, lockless conflict resolution mechanism and write process, which allows the results of write operations (potentially writing to multiple computers) to be exposed atomically to readers.
- An architecture where the internal state of each computer can be reconstructed from the same ordered set of inputs. This allows the whole system (which is the sum of these computers) to be reliably distributed, persisted and generally be made more resilient.

This is different from current systems, where writes either use some form of two-phase commit algorithm (which are inherently blocking) or lack transaction ordering semantics and thus are unable to provide strong consistency guarantees.

In other words: in current database systems, the internal state is protected by certain mechanisms, so that writes and reads can be finished according to their required consistency guarantees. In this system, the internal state of a computer is considered to be the result of an ordered list of input-messages. This means that two computers will always hold the same internal state when this list is consumed by both.

The above means that there can be no race conditions or other sources of ambiguity anywhere in the system, which mandates that computers only communicate with each other using asynchronous methods (since race conditions are inherently non-deterministic).

This document does not aim to be a documentation of requirements for implementing such a system; it only aims to show that this is possible. Some optimisations are suggested, others hinted at, while others again are ignored. This is because the goal of the document is to provide the first proof of concept that such a system could be built and could be maintained, even considering the stringent requirements of the software industry. Opportunities for optimisation will be mentioned in later chapters, the appendix and in footnotes.

Goals

In the following, we describe a theoretical framework which can be used to implement a distributed, linearly scalable and strongly consistent, SQL-compatible datastore. We define these terms as:

Distributed: The system maintains its information on multiple computers.

Linearly scalable: If additional computers are introduced into the system, its capacity will roughly increase by the ratio of the total capacity of the new computers compared to the total sum of all capacity of the computers utilised in the system before they were added. *For example, if the system used 100 computers previously and 10 new computers are introduced, capacity will grow by about 10%.*

Strongly consistent: A global ordering exists between each piece of information. The order between these is referred to as “earlier,” “later” or “at the same time.” The information implementing this ordering is called time[1]. The system will only return results which contain all relevant information introduced earlier or at the same time compared to a chosen point in time, and does not contain any information introduced later than this time.

Implementation

To achieve the goals described above, the following components are specified:

1. A number of clocks ordered into a hierarchical tree structure, advancing independently from each other. On this structure, an algorithm exists which defines global timestamps. Any two timestamps can be compared to each other to decide which happened:
    1. Later,
    2. Earlier or
    3. At the same time.
2. An algorithm which guarantees that each piece of information will be readable and can be evaluated using a certain timestamp and that information introduced later cannot be a part of the results.
3. A solution which guarantees that if there are conflicting modifications, there is a system-wide consensus regarding which modifications (if any) have been accepted and which were rejected. More formally: if *version* is defined to be the value of a piece of information at a given time, then each version can only have at most one successive version.

We also give technical proposals for a possible implementing infrastructure that enables:

[1] This definition of time differs from its general meaning in English

- Fail-safe, redundant communication between computers in the distributed system,
- Error-correction,
- Persistence,
- Replication
- And latency mitigation.

Comparison with SQL

The system described here allows for the implementation of an SQL-compatible database.

The SQL standard allows for:

1. A transaction to be refused for any reason, at the discretion of the server;
2. The database to escalate the defined isolation level without asking for permission from the client.

The system described here guarantees that it follows the industry's ACID properties, which are:

1. Transactions are introduced into the datastore atomically and the other transactions will also acquire any information introduced by it atomically.
2. The redundant constraint-information introduced by a transaction will only become available at the same time as all the other information entered by the transaction and there can never be two simultaneous parallel transactions committed which would together violate the constraint. *Thus, it is always possible for the system to evaluate the validity of a redundant constraint-record either at the time the information is introduced or at commit.*
3. Transactions cannot read or write each other's half-finished states.
4. Information already accepted by the system cannot be lost at a later time.

As implied, this system does not solve the existing issues posed by the SQL-standard, such as phantom reads, phantom writes or the necessity of using SELECT FOR UPDATE statements for signalling write-protection of a set of information when that is needed.

The term **statement** means roughly the same in this document as it does in the SQL-standard, so a read and/or write instruction from the client, which might have resulted in some information being presented back to it.

Distributed hierarchical clocks

The following covers the first point of the components listed in the Implementation section.

Introduction

A potentially globally distributable hierarchical clock is described in this chapter. Its goal is to allow computers whose data are closely correlated to have a fast and efficient information exchange mechanism and still allow for a system which could potentially be grown to a global scale in which

even remote computers could participate in transactions with strong consistency guarantees and relatively quick response times.

The way these “clocks” are arranged is indifferent from the perspective of the specification. Each clock could be hosted on a specific computer or the clocks could be distributed even further. The essence of this chapter is that a strong ordering can be drawn up between values assumed by a set of distributed clocks by establishing an algorithm which is capable of doing so.

Groups and local clocks

Consider a number of computers which send packets of information called **clock token-messages** between each other in a circle. A set of computers such as this is called a **group**. The message contains a natural number which is incremented by one by each computer before passing it on to the next in the chain. In such a setup each computer (as long as it knows its position in the chain) knows which numbers it will receive and which numbers it will send on. The information which is unknown to the computers is the time which will elapse between two token-messages. Since the clock itself is composed of all the participating computers, the **group clock** is essentially synonymous with the group. The only distinction is a fine one, in that when the group clock is mentioned, a specific ability of the group to create versions is being discussed.

The time window between processing two token-messages is called a **period**.

- Periods which have finished before the latest token-message was received (i.e., each one except for the last one) are called **closed periods**.
- The latest period, which will be closed by processing the next token-message, is called an **open period**.

*For example, in a group which contains three computers, the first computer is going to assume the following values: 0 = 3*0, 3 = 3*1, 6 = 3*2 ... m=3*k where m is the assumed value, k is the number of the period numbered from the start of the message-exchange mechanism. Since each computer knows the latest number it also knows the next one. For example, a period on this computer is the time elapsed between processing the messages containing number 3 and 6; or in other words, the time it takes for all the computers in the chain to complete a whole round of message-passing.*

The system never needs to look ahead of the currently open period in any group. This means that if the set of computers needs to change within a group, the new number of computers in the group could potentially be passed along in the message, so that the newly introduced or removed computer can start/stop participating in the message-exchange once the token-messages have gone full circle and reached the node that initiated the start/stop process.

A computer’s local clock can be described with the following two attributes:

1. What was the value of the last message received?
2. How many computers will be present in the group in this round?

The period last closed by the receipt of a message is called the **group clock value**. This is therefore not a global value. To identify it, one needs to know which group is being discussed.

group clock values are also referred to as **versions**. Since each period identifies a group clock value, the terms open and closed versions are also used to mean the group clock value identified by the open or closed period.

Hierarchy

Given a group of computers, we can associate new groups of computers to each. The computer in the original group will be referred to as the **parent** or **parent computer**, the new group of computers associated to it will be called the **child** or **child group**. These child groups will have a clock of their own, which will advance independently from the parent's clock. This establishes a tree hierarchy of both computers and groups (as parent computers are also members of groups).

We define the **top** of the hierarchy to mean the set of computers which have no parent computers and the **bottom** or **leaf** as those which have no children.

It is assumed that communication between computers on the bottom of the hierarchy is orders of magnitude faster than it is between the ones at the top of it.

An **ancestor computer** is either a parent of a computer or a group, or an ancestor of a parent of a computer or a group. An **ancestor group** is a group an ancestor computer belongs to. A **common ancestor group** is a group which is an ancestor group to both of the two specified computers or groups. The **lowest common ancestor group** or **lowest common group clock** is the first common ancestor group of two specified computers or groups when searching from bottom to the top (or more formally: the group, included in both sets, which has the highest number of ancestors). The **root group** is the only group which has no parent and is therefore the top of the hierarchy.

The lowest common ancestor group should always be computable by any computer based on two known versions and the references to their group clocks.[2]

The root group is a common ancestor group to any two groups in the system.

From the child group, a message called **value query message** containing:

1. The name of the sending computer and
2. A group clock value created by it (which can be either closed or open at this point)

can be sent to the parent computer.

When the parent processes the received message:

- If no value of the parent's clock has been associated with the received one, it returns the currently open period. In the same atomic operation, it also assigns the open period's value to all child-group clock values which are higher than the highest value already queried, up until and including the value present in the message.
- If the received group clock value already has an associated parent-clock value, it returns the associated group clock value.

[2] For example, this can be achieved by it knowing the hierarchy of the groups or by choosing a group-naming strategy that allows this.

We refer to this relation as a child group clock version being **associated with** the parent-clock version. Associations of versions are bidirectional. Every child version is associated with exactly one parent version however one parent version can be associated with multiple child versions. The association-relation orders child- and group-versions into a tree. We also use the term **transitively associated** when describing a chain of association (potentially involving any number of elements) in a sequence of parent-child relationships.

As an example, take a child group from which we send value query messages with the values 21, 22 and 23 to the parent. Suppose that there are 3 computers in the parent's group, of which the parent computer is the first one. Let's say the parent receives the values 22 and 21, in this order. When it received 22 in the message, its current open period was 3, so it will associate the child's clock values of 21 and 22 with its own clock's value 3. Once it receives the second message, it will immediately answer 3, since that is a value that has already been associated. Let us also assume that by the time it receives the message containing 23, it is on its next open period (and so 3 is now a closed period), so it will associate 6 with the value 23.

Weak and strong ordering

The association described above can be used to implement a global ordering between any two versions. In the following chapter, we'll describe two strategies to decide if a version comes before, after or at the same time as another one.

We define the **at or before** operator recursively as the following: if x and y are two versions in the clock hierarchy and X is the group clock of x and Y is the group clock of y, x is at or before y if

- x and y are version on the same group clock (X=Y) and y is the same as or earlier than x (y<=x),
- X is an ancestor group of Y and there is a y' transitively associated with x and y <= y'

This algorithm defines a top-down search for everything that is either equal to or lower than a chosen version. This is called a weak ordering because no strict order can be established using this algorithm between the elements within the set. This method is also referred to as the **weak ordering strategy**.

Using this, we can draw up a set of versions which must have been created at the same time, or before the specified version, without having to go deeper into the hierarchy to explore their detailed order.

The **strong ordering strategy** is:

1. If two versions are from the same group, their values are directly compared.
2. If the two versions are from different groups, their associated value is compared on their lowest common group clock.[3]

[3] They are guaranteed to have an order on a common ancestor if they originated from two groups where neither is an ancestor of the other, since they can't have the same parent-computer.

3. If on the lowest common group clock, one version is transitively associated with the other and therefore no order could be established this way, the one which is lower in the hierarchy (i.e.: the one which has more ancestors) is the lower one.

The strong ordering is stricter then the weak ordering by the 3) point. These guarantee a deterministic order between any two versions.

Atomicity of associations between clocks

Continuing the example in Hierarchy: If we were to ask which values are at or before the parent's group clock's value of 3, we know that the answer is 21 and 22. If, however we were to ask which ones fulfil the same requirement for 6, our answer cannot be complete as that period is still open and therefore can still accept higher values.

Since there is no guarantee that the parent computer will receive the child group's clock values in order, the answers coming from the parent computer can only be consistent if it does not only consider its current open period, but it also takes its earlier replies into account.

The association between clocks is therefore atomic, since it is defined by when the parent-computer first received a higher value than the highest one stored previously.

Continuing the previous example, if:

- *The last value received on the parent from the child clock was 22,*
- *The next one was 27,*
- *The current open period's value is 6,*

Then the value 6 on the parent's group clock will be associated with every natural number between 23 and 27.

For a suggested optimisation strategy, see Clocks.

Example

Imagine a group which includes 3 computers, A, B and C. Each of them has an associated child group named after the parent computers, so group A, group B and group C, respectively.

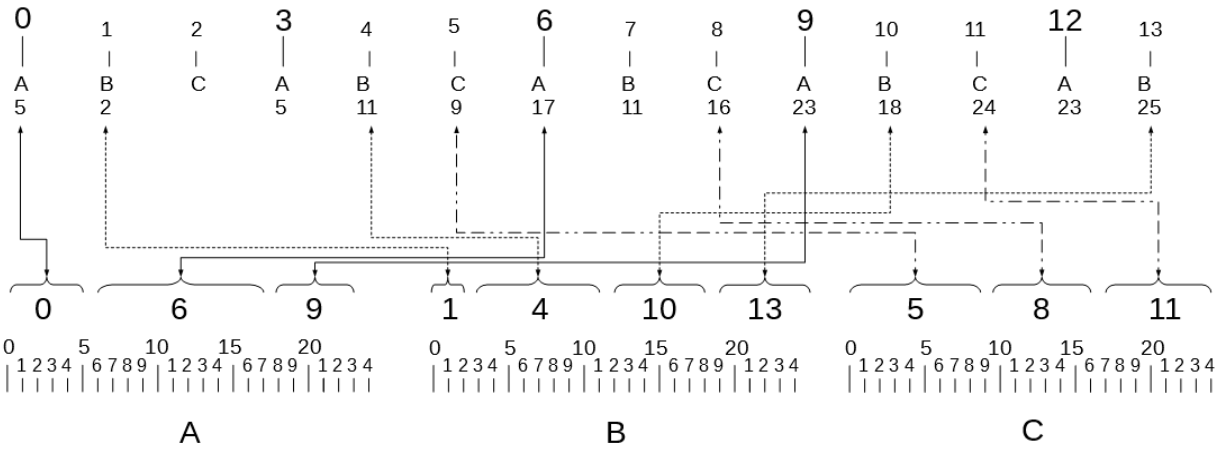


Figure 1: A possible association of version nodes between the described groups

The illustration here shows a possible result of the group clock's independent execution and the value query messages which associate the versions of the 4 independent clocks.

In the upper half, in the first column, from top to bottom:

- The number in the top row is the value of the group clock. So, the first value is 0. The values received by A are shown with a larger number to show at which points the cycle started again.
- The A underneath 0 shows that the value is associated with a value of the group clock of group A.
- The number 5 under the A shows that the parent group's value of 0 is associated with version 5 of group A.

The second column is computer B receiving its first message from computer A, the third one is computer C receiving its first message from computer B and so on.

The lower half shows the three independent group clocks. These are denoted by their parent computer's names, so they are labelled as A, B and C in the bottom row.

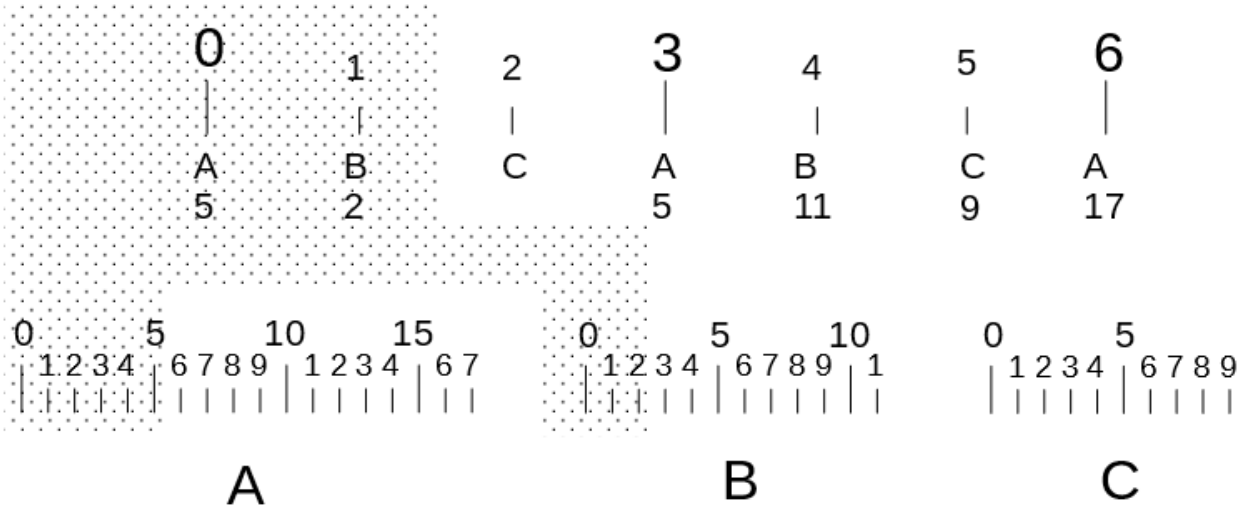
The leftmost curly bracket above the segment 0 and 5 shows the values of group A associated with computer A's 0 period. The arrows show an association-message that was sent to the parent computer from its child group. The leftmost arrow is the first one, where the parent was A, and the message contained group A's clock value of 5. To help differentiate between them, each group has a different style for the line of the arrow. Group A has a solid line, B has a dotted line and C has a double-dotted-triple-dashed line.

Granularity

The figures in this chapter are an elaboration of the example given above. The values in them can be correlated back to that.

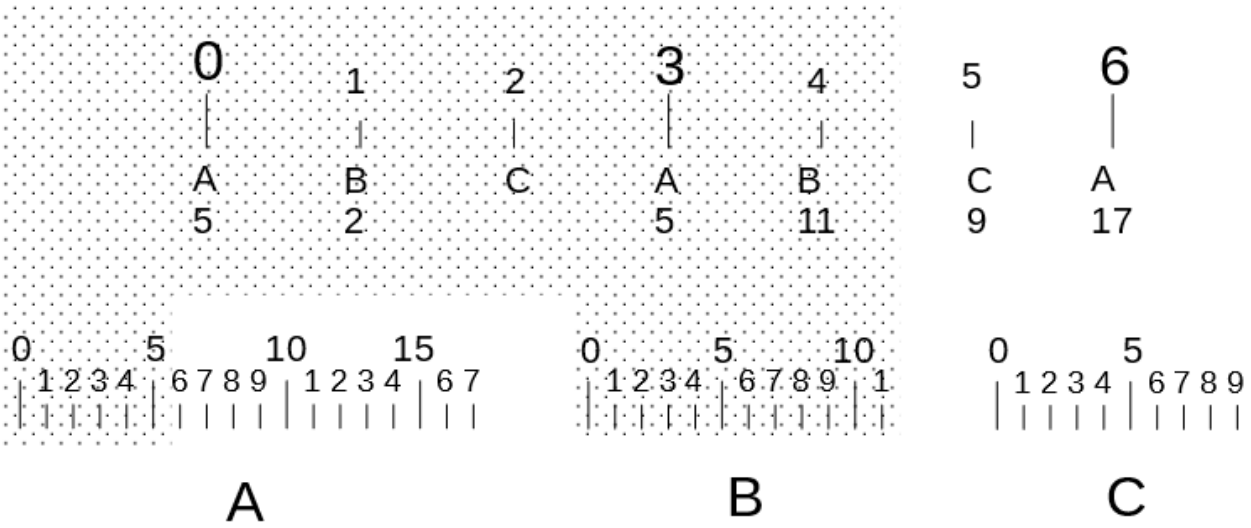
Assume that a client's purpose for using the clock is that it should always be able to decide which clock values happened at or before (or inversely, after) a specific time. First, say that the client uses the root group clock, so when it poses this question, the read operation's reference value (against which all others are compared) belongs to the root group clock.

The next figure shows which values will be at or before if the reference value is 1.



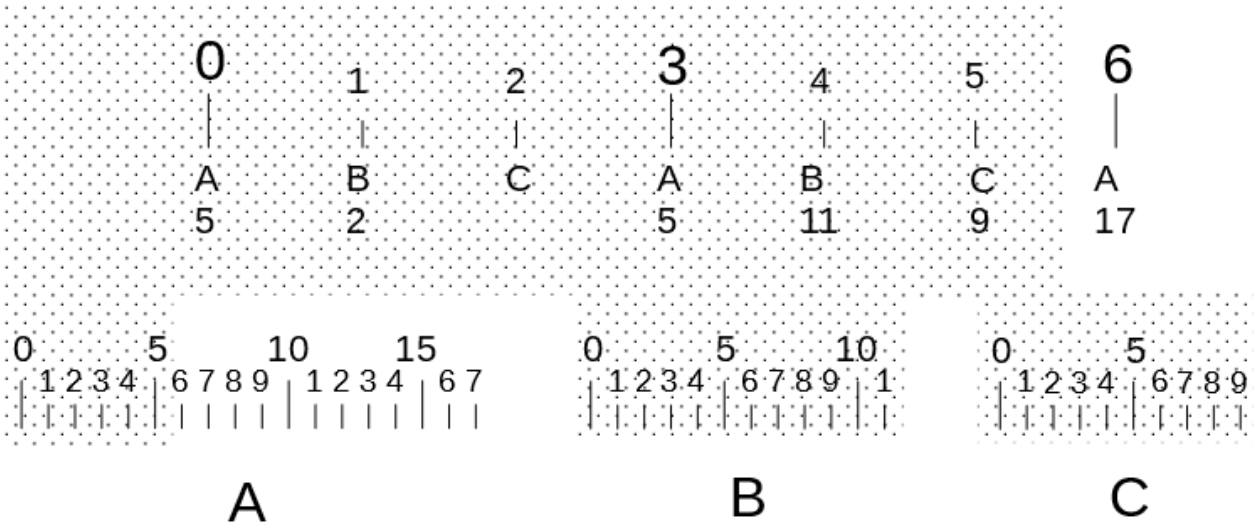
The dotted area represents the clock values which are at or before the root group clock's value of 1. The values included from group A are the ones between 0 and 5, for group B, between 0 and 2. Nothing is included from group C. The reason for this is that even though the child groups' clocks advance independently from their parent clocks, their advancing determine which values will be assigned from the child groups to the parent group.

In the figure above, group B's group clock's value is 3. The values between 0, 1 and 2 were created before computer B received its first token-message. For a client looking in from the outside, these values represented by group B will first be visible after the computer B processed its first token-message, even if the computer B has processed the values sent by group B earlier. Thus, the client will receive the values represented by the dotted area, as the ones which are at or before value 1 on the root group clock.



The figure above illustrates the state of all group clocks after the next round (of passing the token messages) has finished. Since in the previous figure the root group clock assumed the position 1 and since there are 3 computers in the root group, the clock's value is 4.

If we compared the dotted area in this figure with the one above, one can tell that neither group A nor group C provided new values during last round. Group B, however, has extended its set of values since the previous cycle, and now includes values between 3 and 11. This illustrates that group clocks do not assume that child groups associate new values in each round to their parents.



This last figure shows the state after the next value was assumed on the root group clock. Compared to the previous figure, the only change is that the next computer in line (computer C) has processed the token-message it received from child group C. Based on this, one can tell that while group clock assumed values 2 and 5, the highest value computer C received from child group C was 9.

Comparison

An important goal for the hierarchical clock is that it should allow for the comparison of two versions (which are potentially remote to each other) without necessarily needing to exchange messages to distant computers with high network latencies. It is assumed that the network distance between two computers at the top of the hierarchy is some orders of magnitude higher than between those lower down. The other aspect of this is that (assuming optimal geographical distribution) the closest member of the root group is orders of magnitude quicker to reach than the others. Therefore, if the information used to decide whether a certain version is “at or before” another version can be gathered without having to move very far, the operation becomes a lot quicker.

Transitive closure

Since querying the clock hierarchy for open periods is allowed and since open periods are closed independently from each other in each group, it is possible that a parent’s closed period is associated with open periods in its child group. So, even if the parent closed its period and even though this version no longer accepts new associated versions from its children, there is no guarantee that its children will not accept transitively associated versions from their own children. This is possible for an open version on the child group which was already associated with a version on the parent.

This guarantee would only hold if, by the time the parent clock’s period was closed, the periods the child sent earlier were also guaranteed to have been closed. Ensuring this would mean that the parent clock has to wait before closing its currently open period until it receives a signal for at least the highest period from the child computer associated with it. This is not only complicated, but would also slow down the closing process, which would affect the throughput of the whole system.

There are two versions, x and y on two computers. If, on the lowest common group the associated version are x’ and y’ respectively, we call x and y being transitively closed if all the versions in the x → x’ association chain and all the versions in the y → y’ association chain are closed.

We can use this information to know that each version that is associated (directly or transitively) with the specified versions in all the groups between the two versions have already been closed and so cannot include more information than what it already has. This does not guarantee that all versions transitively associated with these versions have also been closed.

Each computer holds an association list of all its ancestor computers and its highest version known by it to have been transitively closed. This associated list is called the list of **known transitive closures**.

Transitive closure messages are communicated from top to bottom. The message contains the list of versions and their originating computers (which is a chain of transitively associated versions). The first element of the list is the highest, triggering computer. The versions in the chain are added to the end of the list once they were also found to be closed.

Each time a parent computer closes a period, it checks if that version was associated with another one on its child group. If it was, it lists the associated versions for each computer in the child group and sends a transitive closure message to each, containing the version it just closed as the first element of the list.

When a child computer processes the message:

1. It replaces the highest transitively closed version information for the ancestors listed in the message in its known transitive closures list if the newly received version information is a higher number than the previous one stored for the ancestor.
2. If the local version associated with the parent’s version is still open, it waits for the next token-message to arrive to continue with step 3,
3. Adds the local version associated with the parent’s version to the end of the transitive close message.
4. List the associated child periods for this version for each computer in its child group and sends the transitive closure message to each, or terminates if it is not a parent computer.

This way, each computer has a list of which versions are stable for their ancestor computers, i.e., the highest known version to which no new values will be associated (either directly or transitively) on its ancestors.

Transactionality

Deletes, modifications and creation of records can be thought of as incremental new information within the system. When defined this way, the whole state of the system can be described by a constantly growing set of information, where new information is based on earlier ones. If a version is assigned to these bits of information, one can tell the ordering of any two pieces of information (for details, see Weak and strong ordering).

If the system is to be consistent, the observer must be able to deterministically tell whether a piece of information is a part of the whole system for each finalised version, and no previously entered information can be lost.

The system stores information, upon which read/write operations are executed. These processes and the modifications created by them are called **transactions**. Transactions can be cancelled, in which case the transaction won't affect the information (i.e., the information readable by clients) in the system. Thus, the transaction defines both a set of information and the read/write process. Transactions are always associated with (and managed by) a single computer. **Records** are atomic pieces of information which are viewable by the client. A new version of a record (created by a transaction updating it) is called a **variant**. Based on the above, when serving a client request, at most one variant for a record can be chosen at any time.

The system supports the atomic finalisation or cancellation of variants created by a single transaction. This means that no proper subset of information written by a transaction can be read by another transaction. Since the client can use any closed period to query the system and since information is distributed across multiple computers, some form of synchronization is needed to achieve this. Current database servers solve this problem by blocking the execution of all but one of a set of conflicting transactions until a decision is made whether the executing transaction is committed or cancelled. This solution involves blocking transactions, so this is not an acceptable strategy here.

If an operation modifies an existing record, the system guarantees that the variant it introduces will be the next one compared to the record it read during its read-phase (this implies that records have to be read by the process before they can be modified) or that the whole transaction will be cancelled.

The **reference version** is a specific version chosen for the statement, to which all encountered variants' versions will be compared. A read operation will only include a variant in its results if the variant's version is at or before the reference version.

Steps of the transaction process:

1. The computer associated with the transaction generates a **transaction identifier**, which also identifies the associated computer.
2. Creates a **transaction record** on the computer associated with the transaction.
3. Executes the read/write operations and identifies the reference version to use for its read operations which depends on the isolation level.
4. Starts the commit process, which will be detailed later, in the Transaction commit algorithm chapter
5. If the commit process...
    - finishes successfully, the transaction is assigned a final version, which will also become the final version of each piece of information it created.
    - fails, the transaction is cancelled, and it has no effect on the information in the system.

Write process

While the transaction is executing its processes (and so can read or write data on multiple computers) it does not have a final version. Therefore, variants created by it (potentially on other computers) cannot have a version either.

Until the final version is assigned to it, the transaction uses its globally unique transaction identifier in place of a version.

A transaction executes the following steps when updating or creating a set of records:

1. Collects the variants and keys of the records it will need to update by executing the read operation (skipped if it is a new record, not a modification).
2. Based on the key of each record, it identifies which computers hold them (the **target computers**).
3. For each record, it sends a write message to the target computer, which contains:
    1. the new value of the record,
    2. the transaction identifier,
    3. if this is a modification: the record's version read earlier.
4. The target computers process the message (in an atomic operation):
    1. If this is a modification: checks if the variant already has a finalised version that is later than the version read in step 1. If it does, then the whole transaction is cancelled.[4]
    2. Creates a new variant which holds the transaction identifier as a temporary (non-finalised) version.
5. The target computers send an acknowledgement for each write operation.

This algorithm is extended (by further information being communicated between the affected computers) later in the chapter Transaction commit algorithm. For the time being, it is assumed that variants created by finalised transactions never overlap. In other words: step 4 on the computer managing the record is the atomic event from which point onward no two parallel transactions can be committed which have both inserted variants for the same record.

If a transaction updates the same record multiple times, the same variant will be updated with the new values of the record, but the original read version of the variant will not be changed. This is safe to do, because (even if the implementing database chooses a relaxed isolation strategy and allows for an earlier version in a later read statement than the one it used before) a transaction will always find its own changes of a record first.

After the commit process finishes successfully, the finalised version is written into the transaction's record. The assignment of the final version to the transaction is the atomic event of the transaction's commit. The final version will be the open version of the group clock on the computer associated with the transaction.

[4] This means that the variant was finalised between the closing of the reference version used for the read and this message being processed.

See An optimised write process below for a proposed optimisation strategy.

Read process

To be able to serve read requests in a non-blocking fashion, the following is proposed:

1. The read process first compiles the necessary data from the computers holding the required keys.
2. It computes the reference version according to the isolation level used by the read process.
3. Before applying the data transformations required by the client, it filters out variants which are “after” the calculated version (see at or before relation).
4. For each record, the variant with the highest version is chosen.

Thus, since variants (which are part of a single transaction) are entered into the system atomically, they will either all be a part of the results, or will all be filtered.

Variants introduced by a transaction will be observable by another read process if the version of the transaction and the reference version of the read process are transitively closed.

An aim of the system is to provide Oracle- and Postgres-like “statement-level repeatable read” isolation-level for each execution. To this end, a single **reference version** is identified for each read process.

Since the transaction is assigned a version in an atomic event, it is possible that the variant found by the read process will not yet hold the transaction’s final version when the query finds it, even though this assignment might have already happened on the remote computer managing the transaction. In these cases, (when the read process encounters a variant with a transaction id during its execution) it has to query the computer managing that transaction about the current state of the transaction. If the transaction has been finalised in the meantime, the variant will be included in the results. In any other case, the variant will be filtered out, either if it has no version or if the version was found to be later than the reference version, unless its transaction-identifier is that of the read process. This is safe to do from a consistency perspective, since no transaction which finished after the read operation started can be included in the results (due to the transitive closure values chosen as a reference version see Isolation level section).

When reading a record, a transaction takes the following steps:

1. Based on the key of the record, it identifies which computer holds the record; this is the **target computer.**
2. It sends a query message to the target computer containing the reference version and the key of the record.
3. The target computer sends a response consisting of:
    1. all the variants of the record, and
    2. the list of its known transitively closed group clock values.

4. Once the response is received on the transaction's computer, if transaction identifiers were found:
    1. Computers associated with these transactions are queried for their versions.
    2. The received versions replace the transaction identifiers on their variants.
    3. The variants marked with a transaction identifier equal to the executing transaction's own transaction identifier are also added to the results.
    4. Those variants which have no finalised version are filtered out.
5. The reference version is calculated according to the isolation level and the remaining variants using the algorithm described below in the SQL Isolation levels support chapter.
6. The chosen variant is:
    1. The one with the same transaction identifier as the transaction running the statement, or if that does not exist,
    2. The one with the highest version using the strong ordering strategy.
7. The results are returned to the client after formatted according to the statement’s requirements.

For a discussion on possible optimisation strategies, see An optimised write process.

SQL Isolation levels support

The SQL standard defines the following read phenomena:

Dirty Read: When a transaction is allowed to read data from a row that was modified by another transaction but not yet committed.

Non-repeatable Read: When a record is selected twice during the execution of a transaction and its value changes between the two.

Phantom Read: When a new record is inserted that fulfils the criteria of a repeated search and it was not present the first time, but it is present the second time.

Isolation levels are defined by whether these phenomena are allowed or disallowed from happening during the course of their execution. The following table connects the read phenomena to the isolation level definitions:

Isolation Level	*Dirty Read*	*Non-repeatable Read*	*Phantom Read*
READ UNCOMMITTED	Allowed	Allowed	Allowed
READ COMMITTED	Disallowed	Allowed	Allowed
REPEATABLE READ	Disallowed	Disallowed	Allowed
SERIALIZABLE	Disallowed	Disallowed	Disallowed

Isolation levels are allowed to be elevated, allowing escalation from the more lenient toward the stricter ones. For example, the system is allowed to serve READ COMMITTED transactions to the client if it requested a READ UNCOMMITTED one, but not the other way around.

Our system only allows the READ COMMITTED and the SERIALIZABLE isolation levels. This is due to the different architecture of the data store, which is designed to be non-blocking by nature.

The following algorithms are suggestions. Implementations are free to give different solutions, as long as they fulfil some understanding of the definition of the isolation level descriptions in the SQL standard.

READ UNCOMMITTED

Read uncommitted would allow clients to access data which has not yet been committed by other transactions. This would be impractical in the described framework and since the spirit of this isolation level is to allow the ability to provide non-blocking queries to the user and this is already fulfilled, the framework automatically escalates this level to READ COMMITTED.

Since there is no obvious way to identify which of the non-final variants will be the next one for a record, one cannot implement a READ UNCOMMITTED isolation level, in the sense the SQL standard specifies it, in our system.

READ COMMITTED

The following requirements can be eased by relaxing the statement-level consistency guarantees.

If there is a guarantee that information which has been seen by a client can still be accessed later on (an assumption made in this proposal based on the durability requirement of ACID), the value which has been established as a reference value for a statement will have to be the minimum reference value for subsequent statements. Since moving up on the group clock hierarchy means that the period above it might be open, in some cases the subsequent statement may have to wait for a group clock to include the earlier reference value.

The computer executing the read process and the computers which serve information for the process are together called the **participating computers**.

Any reference version is fit for statements running on a READ COMMITTED isolation level, as long as it fulfils the following criteria:

1. The reference version must be at or before the version produced by the following algorithm:
    1. Take the lowest common group clock of the participating computers.
    2. Take lowest value stored in the lists of known transitive closures from the participating computers.
2. The reference version of each statement (which is non-first in a transaction) will either have to be equal to or be higher than the reference version of the previous statement. The reference version described this way is called the **historical reference version** and is stored on the database record. If the reference version identified by the strategy is lower than the historical reference version, the process has to wait until the identified computers receive notifications that the historical reference version (or a version associated with it) has been recursively locked for all of the participating computers. Once it has received this notification from each computer, it can retry executing the statement.

For an optimisation strategy on choosing which ancestor-version can be chosen, please see Comparing versions.

Since transitive closure takes some time (even after a transaction has received its final version), there is a trade-off between:

- newer information being present in the results—in this case, there is a potential delay a client might have to suffer while executing a statement; and
- newer information missing from the results—in this case, the client does not have to wait for the statement to return with the results.

Below, we present two ends of the spectrum and a mid-way solution.

1. In solution 1, the recently finalised variants will not be found by the statement, but we will only have to wait for a consistent state in very rare circumstances.
2. In solution 2, the statement may have to wait in certain cases, but it is guaranteed to find the latest possible state that is guaranteed to be consistent.
3. Solution 3 has a mix of both, to show a possible compromise between solutions 1 and 2.

Solution 1: Non-waiting

The statement uses the versions which are at or before the lowest recursively closed version of the lowest common group clock of participating computers.

The data set for this reference version is guaranteed to be consistent without having to wait for anything, as long as the identified version is higher or equal to the historical reference version stored in the database record.

Solution 2: Up to date

The statement identifies the last closed period on the computer processing the statement as its reference version. When the process identifies a computer that needs to be queried for information, it waits until the computer has received the recursively locked message for at least the lowest common group clock's associated value. The resulting reference version is the lowest common group clock's version which is associated with the version the statement identified at the start.

Solution 3: Middle ground

The statement identifies the last closed period of the parent computer of the computer executing the statement, thus making sure that all the computers which belong to the same group as itself will have a consistent data set. If the statement needs to query computers outside its own group, it uses the method described in the "Up to date" strategy.

Which point on this spectrum is targeted by the implementing database should be determined by the use case of the database and it can be a configuration setting.

This strategy works for SELECT FOR UPDATE statements or conditional UPDATE statements also. In these cases, the statement has to do two passes:

- On the first pass, it identifies the variants it will need to update, in the same way as if it was doing a SELECT.
- On the second pass, it inserts its own variants by executing the write process.

Since modifications are considered to be additional variants in the system and not direct changes to the record, finalised variants are immutable. Since the write process will use the variants it identified in this step, the optimistic conflict resolution algorithm (described in chapter Transaction commit algorithm) will deal with any potential parallel modifications from this point on.

Once the reference version has been identified, it replaces the currently stored historical reference version with the new one.

REPEATABLE READ

The framework automatically escalates this level to SERIALIZABLE, since new variants are versioned using the same method as updated or deleted ones, and therefore we cannot make a distinction between the two.

SERIALIZABLE

Clients look for information which has a lower or equal version to the last known transitively closed period of the root group clock, or if that is found not to be transitively closed on a participating computer, the lowest one of the transitively closed versions (based on the lowest common group clock) found on the participating computers.

It is possible that a version which is transitively closed on one computer is not closed on another one. In these cases, even choosing the lowest version in a common ancestor group will only guarantee that this fulfils the requirements for SERIALIZABLE for the statement, not the transaction, since there is no guarantee that the transitively closed information already reached all the computers which hold information for later statements in the same transaction. Theoretically, it is possible for a subsequent statement to choose an even lower version when applying the above rules. Technically, the only way to be sure that there can be no need to wait would be if the computer managing the transaction knew what versions are known to be transitively closed to all the other computers in the system.

This however, is a moot point. Realistically, the time it takes to send the transitive closure message (which must have already been sent for the previous one compared to the one chosen based on the rule above) is much shorter than the time it takes for a remote message to be received, so in these extreme cases waiting before answering the message should cause only an insignificant delay.

Using this version means that the referenced set of information will not change. This also means that there are more opportunities for transactions to conflict than on the READ COMMITTED level.

Constraints and Indices

Since constraints and indices are redundancies of the original transaction’s data set, they will all either be chosen to be the next version or will be rejected together. The SQL standard only requires these values to be correct after commit, so the SQL standard still holds even with the optimistic

conflict resolution model, even if this introduces a change for developers when working with the database, compared to current database models.

Transaction commit algorithm

Model

Because the set of variants left by a single transaction can only be finalised atomically, the event of the finalisation of these variants is the same event as the finalisation of the transaction itself and vice versa.

Each variant holds:

- The version of the variant that was modified
- The finalised version of the transaction
- The new value of the record.

Since the version of the transaction is either missing or (if finalised) necessarily later than the version of the variant which was modified by it, a period between these two versions can be identified. We will call this the **contended time window**. The purpose of the algorithm in this chapter is to guarantee that there can never be two finalised variants for the same record with overlapping contended time windows.

In other words, the observer must always be able to tell what the state of a record was at any given time. In a distributed environment, this is not only an important but also a difficult question, since different computers can each have different pieces of information. The classic solution to this problem is to make overlapping write processes wait for each other's termination, so that if both transactions are trying to modify the same record, only one of them can progress, and the other one will have to wait until the first one has finished all its operations.

Since transactions cannot wait for each other in this system (this is a requirement of linear scalability), a different solution is needed.

Variants in the system always have a guaranteed order between them (i.e., *they cannot be equal to each other, or they would have been written by overlapping transactions).*

In the next examples there are two transactions: A and B, which are both modifying information X held on computer 0. The information of the transactions is held on computers 0_A and 0_B, respectively. The variant for X written by A is called X_A, the one written by B is referred to as X_B.

Each transaction leaves a mark of its transaction identifier on the variant created by it, as explained in the Write process chapter. If transaction B's write operation creating X_B came after the creation of X_A, we say that **B is known to be clashing with A on X**, since transaction B can (during its execution of the write operation) see A's transaction identifier on the variant X_A. This is not necessarily a symmetric relationship. For it to be so, both transactions would need to know about the variant left by the other one. To achieve this, A would need to look at X at a later stage again, after X_B had been created. Therefore, this relationship can be symmetric, but it is not guaranteed to be.

As explained in the Read process chapter, a read process can only read the variants either created by its own transaction or those with a finalised version at or before the reference version of the read process. Here, the discussion is simplified so that a process can only read variants which were already finalised when the read process started.

Continuing the example, if B is known to be clashing with A, this also means that B has information that both A and B cannot be committed. So B will only get an opportunity to be committed, if A is cancelled. The situation is different from A's point of view: since it arrived at X before B's write operation did, it does not have this information. In other words, A does not know that it conflicts with B.

Thus, there is asymmetrical information between the transactions, which affects their end-state calculations. However, the dependency itself is mutual. So, if A is committed by the system, then B has to be cancelled and vice versa. As mentioned earlier, the commit processes can also be run on different computers at the same time and they are not allowed to block each other from finishing.

The solution is an algorithm that guarantees that (based on the information available to them) each transaction can calculate what the final state is for overlapping transactions. In other words, which ones were committed and which ones were cancelled in the system.

If the final state of a transaction (the information whether it was committed or cancelled) is purely a function of its dependencies on other transactions held in the system (and their state), the final state itself is only a redundancy, which can be reliably calculated again by looking up the necessary information.

So, the goal is an algorithm which guarantees that:

1. Two clashing (conflicting, overlapping) transactions will never both be committed.
2. Each running transaction will reach the same conclusion about the final state of each overlapping transaction.
3. The algorithm can run in parallel on multiple computers with potentially high latencies between them and each one will reach the same conclusion about the end state of each transaction it needs to know about.

Algorithm

When a transaction's commit process is started, the following must already exists:

1. The set of transactions which the transaction knows to be clashing with (as it has seen their variants during its execution) and
2. The set of transactions which have seen at least one variant created by the transaction for a record they also modified, and have notified the transaction about this at the start of their own commit process, before the transaction's commit process was started.

Let us call the first set the **discovered clashing transactions** (or **discovered** for short), and the second set the **registered clashing transactions** (or **registered** for short). The elements in either set will be called the **detected transactions**.

Every transaction's intent to commit either has to be registered on the record of the other transaction, or (failing that) calculate what the end state of the other transactions are. In other words: whether a transaction A can register its intent to commit on transaction B is the same as asking the questions:

1. Will A be finalised and force B to be cancelled?
2. Will B be finalised and force A to be cancelled?
3. Will both of them be cancelled?

The goal of the algorithm is that at any time a transaction reads the data of another transaction, it can make an unequivocal decision.

When transactions are committed, they need to notify the discovered transactions about this and must stop them from being committed before those transactions can stop it.

When transaction B has finished writing all its variants and collected all the information about the transactions it clashes with (call this transaction A), the client can ask the system to commit its changes. We can look at the transactions known to be clashing with this transaction in two ways:

- B must stop A from committing or
- if A's commit process has already started, B must find out what A's final state is and determine its own final state based on this information.

So, if transaction A's commit process was started earlier on 0_A and B discovered A, then:

1. It inspects and finalises the registered set on 0_B. After this, the registered set is immutable.
2. Finds the transactions which were:
    1. not in its registered set
    2. but are present in its discovered set
3. Send a message to the computers managing these transactions, notifying them that B is clashing with them and that its commit process was started. In our example, this is A managed by 0_A.

After this atomic step, B does not accept any more transactions which could stop it from committing. Therefore, transactions which clash with B on any record and arrive later, are blocked by B, so that they can only commit if B is cancelled.

Using the above, the potential circles are eliminated from the analysis, as the essence of such a loop is that an event in a set affects (directly or indirectly) an element which was already present in it. This is impossible, since as soon as the commit process is started, a transaction record's state is no longer mutable.

Since the discovered set is final (as the transaction's execution has finished) and the registered set is no longer mutable after this event either, any state that could influence the decision on A's end-state is immutable as far as the transaction record present is concerned.

So, if (when the commit process is started) A finds that B is listed in its registered set, the following is known:

1. B's commit process started before A's.
2. B's registered set is no longer mutable.

So it is known already whether B is an obstacle to A committing or vice versa.

If B is not in A's registered set, A will assume that B has not started yet, so B's end-state doesn't have to be calculated by A when its commit process decides whether it can be committed. In this case the transactions which have to be inspected are the ones which are in the registered set at the time of finalisation.

A transaction will be committed if and only if:

- either its registered set is empty,
- or every transaction in its registered set has been cancelled.

If we assume that every transaction's state can be explored in the system (and since there can be no circles in the registered sets), it can be shown that an end state can be reliably calculated for each transaction.

This would be a sufficient solution if each commit ran in a global order. However, since multiple computers participate in the system, commit processes can start in parallel and they can finalise their registered sets simultaneously. Therefore, it is possible that transaction B discovered by A is not yet present in its registered set, but by the time the message (notifying B) arrived, it already started its own commit process and thus finalised its registered set. In this case A is cancelled. Since this relationship is asymmetrical (so B does not necessarily know about its overlap with A) it is possible, but not certain, that both transactions are cancelled.

If transaction A has successfully registered itself in the registered set of each of its discovered transactions, we say the finalisation process had a **successful start**. Conversely, if A encounters at least one finalised registered set while trying to register itself, we say that it had a **failed start**.

A successful start is a necessary but not sufficient condition for a transaction's successful commit.

To decide whether a transaction had a successful start, it is necessary to:

- Receive each message on the computer holding the record of the discovered transaction,
- For the message to be processed,
- The resulting message to be sent,
- For the original computer to process the resulting messages.

Thus, the information is not available at the start of the commit process. This process of sending out the messages and waiting for their results is called the **start state aggregation**. The start state aggregation process is considered closed when either the first unsuccessful registration message was processed or when all registration messages have been processed successfully.

Therefore, a transaction A can be considered to have committed if and only if:

1. its registered set is empty or each transaction in it was found to have been cancelled; and
2. for each transaction B which is not present in A's registered set, but is present in its discovered set, transaction A is present in B’s registered set (in other words, if A managed to register itself into each discovered transaction's registered set in time).

And it is considered to be cancelled otherwise. For a discussion on proposed optimisation strategies, see the chapter Commit algorithm.

Implementation

Even though the theoretical framework does not require any of the following implementation-details to be followed, these are included to show how a practical solution can be assembled from the pieces presented above. Since these are only recommendations and since performance is an important aspect of feasibility, optimisations are also discussed in the chapter.

Clean-up

In order to get rid of information which can no longer be of use in the system, it needs to reach a global consensus on what versions are still in use. The result of this consensus is that anything before this version can either be removed (if it was technical information used for the commit process for example) or made permanent (if there were multiple versions for a record for example) for information where versions no longer matter (as it is now visible to all present and future read processes). Since each computer executing a process must participate in this consensus, its value must eventually be communicated as a value of the clock of the root node-group, as this is the only ancestor-group to everyone.

A possible solution to this is to implement a hierarchical message-passing sequence between the computers executing these processes. Each round-trip could yield the lowest version still in use within the specific group of computers, after which this version could be translated to the parent-computer’s level by passing it upwards in the hierarchy. After the root-group completes its own message-passing round, the result is the lowest global version still in use by any of the currently executing processes.

This mechanism could manage timeouts, cancels and information left over from transaction commits, for example.

For performance reasons, it is suggested that transactions (which have been assigned a version) should update the information they inserted by converting their own version values to the clock of the computer holding it. Although no transitively associated value can exist between most computers, a local range can still be specified, which would say something like “version between X and Y” (where $X < Y$ and are local values).

As an added measure it is also suggested to introduce a reserved version number, which is lower than any other version in the system. This is useful when a version of an information is known to be lower than the lowest version still used by the system. In these cases, the information’s version would no longer serve any purpose but would still need to be considered when running queries.

Messaging, Redundancy, Replication and Persistence

Introduction

Even though a technical detail, redundancy, latency- and failover-management are critical aspects of distributed systems. In the following, we present a possible way to configure the system so that it can recover from the simultaneous failure of multiple nodes, provide error-correction, persistence and a latency mitigation strategy.

Description

The system communicates using asynchronous messages only. These messages are published to- and consumed from **queues**. There is no requirement for all messages in the system to provide a response for every message-type (i.e.: some message types require a response, some do not), and if it does, there's no requirement for the response to be sent immediately after the original message's processing has finished (although the spirit of the framework is that it should).

It is required that each node in the system must:

- Be able to list the versions used by its executing operations according to the requirements of the implementing system. *No obvious solution can be suggested here to determine this version, since the way isolation-levels are implemented in a system or the way caching strategies are employed can influence what versions are still used here.*
- Not rely on any state specific to the host-system (e.g.: computer's internal clock, IP- or MAC-address or any of the other local settings).
- Must only use deterministic code, like repeatable randomization using a known seed.
- Be able to provide the index of the last message it processed from its input-queue when requested.
- Have a set number of running replicas.

Each queue in the system must:

- Have a repeatable order in which they store messages
- Be able to provide these messages again, in order, starting from an index chosen by the client API
- Be persisted, so that if the system suddenly loses power, the messages which have been accepted by the queue are not lost
- Be consistent, so that messages which have been accepted by the queue are still available even if any of the computers managing the node becomes unresponsive

The above means that the human definition of time is replaced by one of message ordering and determinism. These requirements also mean that each state of each computer can be reconstructed by feeding them the ordered messages in the public queues.

Nodes

So far, we have only discussed a single type of node in the system, which handled the functional requirements of the database. In order to provide the infrastructure needed to support the non-functional requirements outlined in Introduction, we will need to introduce new kinds of nodes also.

The kind of nodes are:

- **actor nodes**, which participate in maintaining the internal state of the database and executing the operations upon it (i. e.:, the nodes mentioned previously).
- **conductor nodes**, which manage the forwarding of messages between a source to a target actor nodes. They reside on the target actor node's computer.
- **persistence nodes**, which provide durable backups of the system states.

The existence of message-channels is also implied throughout the document. We refer to these channels as **queues**.

Queues can be classified into:

- **staging queues**, into which actor nodes publish, and from which conductor nodes consume messages. They reside on the same computer as the node which was the source of the message.
- **public queues**, into which conductor nodes publish, and from which actor nodes consume messages. They reside on the consuming actor nodes' computers.

These queues can be replicated and distributed in any way the implementation sees fit. There are currently a number of available solutions to this, such as various messaging providers, Kafka and JMS.

A separate instance exists for each staging queue for each actor node instance. The nodes can be ditributed according to the specific needs of the concerned actor- and staging nodes.

The purpose of the conductor nodes is to forward messages from the staging-queues to the public-queues (when their configured conditions for forwarding have been fulfilled) and at the same time provide error-correction guarantees.

For example, a conductor node might be configured to designate a replica of an actor node to be marked as unavailable if it didn't receive the expected answer from it within the allocated time. Once it becomes available again and has caught up with all the messages in the queue, this marker can be removed and normal operation can resume.

Redundancy

Conductor nodes can provide a failover mechanism by forwarding the messages from the staging queues to a specific public queue once a set number of the same message has been received (**minimum-message number**) on its configured staging queues (see the section Network Splits for additional rules). Since each executing process will publish exactly one copy of each message, minimum-message number has to be lower than or equal to the number of replicas for the targeted node (**replication-factor**)**.** This way, even if "replication-factor – minimum-message number" nodes are lost in the system, it's still fully functional.

Conductor nodes can also be configured to have a set maximum idle time so that if at least one copy of the message is received to the configured staging queue, it can still forward it to the public queue if no other nodes have responded within the allocated time (e.g., if a VM garbage collector pause or a spike in network latency occurs).

The above implies that replicas of the public-queues must be maintained synchronously, since if they were maintained in a write-behind fashion and the nodes which are ahead were lost, some messages would not be fulfilled in the system.

Conductor nodes can have multiple instances running on multiple machines for redundancy. In this case, the framework must make sure that a message will not be processed twice by the actor node. This (for example) can be done using a “publish only if not present” mechanism, which must be atomic for the distributed queue, or by the actor nodes ignoring the message after its first occurrence in the queue.

Error-correction

Conductor-nodes provide error-correction mechanisms by comparing the messages published by all the other nodes to each other. According to the Gigabit Ethernet network standard IEEE 802.3ab, the maximum acceptable Bit Error Rate is 1 in 10^{10} bits. According to a relevant paper[i], (in real-world circumstances) somewhere between 1 in $16 * 10^6$ to 1 in 10^9 TCP packets can be corrupted without a CRC catching the error.

An error like that could be problematic for a system such as this, so an additional layer can be useful. Provided that the nodes are expected to have a uniform distribution for the bits which can potentially be corrupted in transition from one node to the queue and the distribution of error bits on any two nodes are independent, the chances of three messages having the same error, (even when considering the lower numbers, i.e.: higher chances) would be 1 packet in $(16 * 10^6)^3$, which means that it would take roughly 130 million years for the system running a million packets each second for one bit of error to happen. This is compared to the current state of the art, which is $16 * 10^6$, which would be about 16 seconds.

Persistence

The system’s backup mechanism works through **persistence nodes**. Their purpose is to consume the same input messages as all the other nodes in a write-behind approach which does not block any of the other nodes from executing and to save the resulting internal states. The persistence node can consume messages at its own time, since if the computer running it or the surrounding infrastructure does go down, it can still pick up where it stopped by fetching the next index from the distributed queue. Another requirement for the persistence nodes is that the write of the state to its local disk is transactional, so that if it reports that all messages have been consumed up to an index. This must be true even after a system failure. The persistence node is allowed to fetch messages again. This is useful if it was stopped from reaching a consistent state for some reason. This way it does not have to reach a consistent, persisted state after processing each message.

There is no limit to the number of persistence nodes the system can maintain, so these can also act as local copies of subsets of the whole system.

Latency Mitigation

Any two computers exchanging network packets over some physical medium suffer from some level of network latency. This is an often-discussed problem of distributed systems and no clean solution has been proposed to this issue.

Due to the requirement documented earlier that replicas of public queues must be kept synchronously, and because each operation must wait for all its messages to be answered when an answer is expected, at first glance, the system seems to have as many "single point of failures"[ii] as there are public queues in it.

There are, however mitigating strategies. Since nodes can have multiple synchronous replicas in different geographical locations (and are therefore very unlikely to suffer the same latency spikes as the other members of the cluster), and since modern queue-management systems like Kafka can alleviate latency problems by choosing a "leader" to define the correct state of the queue based on the availability of queue-replicas, latency problems can be isolated when they arise.

Let us define the "minimum-message number" property of a given actor node as the number N.

If we disregard local processing times, and consider that the conductor nodes are executing on the same computers as the public queues, any message's latency in the system will be equal to the highest value of the lowest N network latencies between the staging and the public queues.

Say that there are 5 replicas for a public queue, where the median latency is 5 milliseconds for each replica. Of these 5 replicas 3 are synchronous, the rest are write-behind. If one of the replicas is isolated from the others due to a network outage, the node can still work as expected and the 5 ms latency will still hold, as the missing replica will be ignored until it rejoins the cluster. In these cases the normally synchronous replica can be considered to be reclassified to a write-behind status, as it can only rejoin the other two once it has caught up with all the messages that were published since it was isolated from the others.

For write-behind, asynchronous replicas, the only requirement is that all their input-queues must have an eventual consensus about

- The messages and their contents in the queue and
- Their ordering.

Eventual consensus here means that not all queues have to receive all the messages at the same time, but if they have, the messages and their order must all match until that point and the missing message will eventually have to be delivered before the one after it is accepted.

Network Splits

A subclass of the latency problem is the network split. If the system were to naively serve clients during a network outage between two datacentres replicating the same nodes, it could end up in a situation where it allowed mutating its local states which can conflict with the local state mutated by other computers on the other side of the network split.

Because of this, nodes must only be able to serve requests if they can communicate with the plurality of its own public queue's synchronous replicas. Luckily, there's already a mechanism to

deal with this rule, the conductor node's settings. If there are no incoming messages to the public queue of the node, it won't process any operations, and will therefore be idle. So, conductor node's settings must disallow publishing messages to the public queue, unless it can see more than half of all synchronous replicas of it. This way, if a node fails to send a message to another node, it will be notified about the network outage affecting a particular region of the system and will therefore stop serving requests.

Conclusions

So as long as at least one of the persistence nodes is still available for each instance, the system can recover from crashes. Since the guide allows for multiple such persistence nodes, it should always be possible to recover from node failures.

Brewer's CAP Theorem[iii] And Its Ramifications

Definition

The CAP theorem defines three desirable properties of distributed systems:

1. Consistency (C) is the knowledge that every read receives the most recent write or an error.
2. Availability (A) is the timeliness of the operations reading from or writing to the system.
3. Tolerance towards network partition (P) is the time window allowed for network latency to interfere with the execution of the read/write operations

It states that it is impossible to satisfy all three conditions at once.

It must be noted that we define the transaction's atomic commit as the "time of writing" for point 1. We do this so that we can compare our system with the spirit of the original paper. In practice, transactions in MVCC DBMS systems very rarely ask for "the most recently committed" values, as they usually have statement-level consistency guarantees. This means that at the very least they maintain a fixed version from the time the statement was started and only fetch information created before that. Nonetheless, the problem of network latency still applies to our framework, so discussing this subject is not without merit.

A shared data system is a networked set of computers which need to communicate with each other to gather any data which is not stored on the local computer. In any such network, especially distributed ones where network latency is a fact of everyday life, if a computer has to query another (via a potentially high-latency connection) the querying computer cannot be sure about the consistency of the data it holds until the response message arrives, potentially much later.

From this problem comes the fact that if a remote computer has a piece of information which might belong to a queried transaction, the querying computer must decide to either tolerate more latency (and thus make the operation which triggered the query wait for this information) or terminate the waiting process and return with the potentially incomplete set of information.

To quote Brewer from a 2012 article[iv]:

In its classic interpretation, the CAP theorem ignores latency, although in practice, latency and partitions are deeply related. Operationally, the essence of CAP takes place during a timeout, a period when the program must make a fundamental decision-the partition decision:

- *cancel the operation and thus decrease availability, or*
- *proceed with the operation and thus risk inconsistency.*

Retrying communication to achieve consistency, for example, via paxos or a two-phase commit, just delays the decision. At some point the program must make the decision; retrying communication indefinitely is in essence choosing C over A.

Our approach

The framework described in this paper is infrastructure agnostic. In fact, since the algorithm itself is asynchronous and therefore provides considerable performance-gains by this virtue alone, it can make sense to implement the system on a single computer, which does not suffer from network latency and to which the CAP Theorem does not apply.

If the implementation is done over a networked set of computers, latency would have an impact on message-passing between the proposed queues defined in the Nodes section. Going by the definitions above, the framework prefers Consistency to Availability.

See the Latency Mitigation section for a detailed description on how we minimize network latency impact on Availability.

Appendix

Optimisations

Clocks

Since associations are originally held on the parent-computer, at first they are only available there. They can however be freely cached throughout the system, as associations are immutable, so any copy of an association information is always reliable. This opens the door to a wide array of different optimisation strategies. Considering how different these strategies can be, this is viewed as an implementation-detail.

An optimised write process

The process can also query for group clock values to be associated with the transaction's finalised version from each of the ancestor computers of the computer managing the transaction. The write process is free to also write these versions into the variants introduced earlier by the transaction, to help speed up later queries.

Since operations:

1. Do not need to wait to know which group clock value a period will be associated with.
2. Only use closed, local versions to compare (potentially open and potentially remote) versions.

As long as the information about a remote version's and the local version's lowest common ancestor-group's equal value is available locally:

1. The operation never needs to look further than the closest member of the root-group to compare two versions and
2. Never needs to wait to be able to do version-comparisons.

The first point is a result of the clock's ability to provide association information about open periods.

The second comes from the fact that reads in the system only use reference versions which have been transitively closed for each computer that participated in serving the transaction. Because this version was already closed on all ancestors at the time of looking for the data means that information associated with the versions could not have changed afterwards.

Comparing versions

As long as a set of versions belong to a single cycle within the same group and as long as they were each marked as transitively locked (on different computers), it should be safe to use the highest version in the set, as a higher one couldn't have introduced extra information to the versions represented by the lower computers. This is because other versions (i.e.: versions which are not associated with an ancestor computer) only become available to computers when the token passed for their particular ancestors.

Commit process caching

Since the information contained in the transaction record can either grow, stagnate or is immutable, an observer can always decide which of any two presented transaction records the later one is. In other words: if a transaction's record is copied and in a later stage of the process another copy was found, the process can always decide which copy holds more information.

It is also known that as far as the whole system is concerned, no new information is created for a transaction once its state was marked immutable, but since only a subset of all the information in the system is known by each process, these bits of information can often be useful for the processes executing the commits.

Assume that a local cache exists on the computer managing the transaction. The information available in this cache is the data gathered during the execution of the commit process.

Therefore, if

- The process's start was successful, and the process managed to write in each transaction's record, and
- Commit processes always communicate the most information they can about the states of the transaction records known to themselves and
- Caches store all information that was communicated to the computer while a commit process was executing

The information stored in the cache will be the state of the record of each uncommitted element of the discovered transaction when it was processed on the remote computer. In the cache on the remote computer, the immutable state of each registered transaction's record is available. This means that the only information a record should need to find to make a decision is whether the listed transactions had a successful start process or not. Once it has found this information, it is free to leave the calculated end-state in the caches of the computers it encounters during its commit phase.

Worst case scenarios

When calculating the time it takes for an operation to finish (for the sake of this analysis) we ignore the local execution time of the operations. We do this for the following reasons:

1. The time it takes to execute batched operations on an OS without context switching is expected to run in the microseconds range, well below the time it takes to pass messages between even the closest computers
2. It's a lot easier to calculate the network-distances, because these numbers are relatively stable. If a computer is congested with too many operations waiting for execution, the question of whether it is congested or not is a volatile state, i.e., it is expected that these busy states are transient, short-lived episodes.

Reads

The reference version used for the read operation is a function of the chosen isolation-level. In the cases described in the READ COMMITTED and the SERIALIZABLE chapters, some of the read operations have to be repeated after finding a later reference version and in some cases the operation has to wait until a group clock closes a certain version.

Waiting for a version

If that version does not exist yet, the process will have to wait for it to be created. However, the next version on the lowest common group clock will include both the previous reference versions and the current one, so no process has to wait longer for this version than until the closest member of the lowest common group clock receives the next token-message. In the worst possible case, the process will have to wait for an (almost whole) revolution of the root group clock and for the recursive lock messages to cascade to the lowest nodes in the hierarchy.

In the first case, the read operations have to be repeated only if the reference version calculated by the statement is found to be lower than the previous reference version used for a read operation in the same transaction.

This can only happen if the group clock of the read-reference of the last statement was lower than that of the new one (as group clocks can't move backwards). This means that the new statement is asking for some information from a computer which is further than the furthest one of the old one.

This could happen due to either:

1. a remote computer committing a remote group clock's version information to a computer in the local group and those remote versions have not yet been translated to the lowest common group clock
2. or the statement asking for some state which is held on a computer a lot farther away

An example for the first point would be a distant computer writing a variant to a local computer. An optimisation could be employed here which could list the highest common versions which are known to be lower than the version the transaction could eventually be assigned if it does commit in the end. This information could be used to exclude most variants from distant computers since they are expected to have higher versions when compared to transactions committing locally (as their contended write-windows are much longer).

Repeating reads

When an operation has to wait for a period to be recursively closed, it will also need to repeat the read operations it executed earlier. This is because it's possible that new data was entered while the operation was waiting for the period to be recursively closed, which might affect the end-result. In these cases the read will never have to wait for another period to be closed, so it follows the linear performance of the reads.

Writes

Writes need the following steps to be completed before they can finish:

1. Write all the information they need to in the affected computers
2. Become a successful commit using the method outlined in the Transaction commit algorithm chapter
3. Determine the nodegroup clock value for its version

Thus, the bottlenecks in the above steps need to be identified. The first step will have a linear correlation between the number of records to be inserted, updated or deleted and the number of messages to be sent. The commit algorithm is much more complicated, but with the caching strategies explained in the Commit process caching section, the amount of communication needed cannot be more than querying the computers held in the „overlapping transactions" set on the transaction record which is being analysed. Since these messages can be sent in parallel, the worst case scenario is the highest latency between the querying computer and the remote computer. Even on a globally distributed system, these should not take longer than ~300ms.

Commit algorithm

It is possible that a process needs to look for the details of a transaction present in a registered set.

This can happen exactly if A received its registered transaction B's state while it was waiting for its start state aggregation to finish. In these cases, since it can be known that the only reason the required information has not been received yet are technical (so not a result of having to wait for an input from the client), 0_A can wait before it answers B's query until the start state aggregation has

finished. In other words, it can make the querying process wait until the end-state of the aggregation process is known.

Queries like this only have to be run if none of the transactions calculated the end-state of any of the transactions involved, based on the information it gathered while executing its commit process, which is an unlikely scenario.

[i] https://dl.acm.org/citation.cfm?id=347561&dl=GUIDE&coll=GUIDE

[ii] Single Point of Failure: https://en.wikipedia.org/wiki/Single_point_of_failure

[iii] https://fenix.tecnico.ulisboa.pt/downloadFile/1126518382178117/10.e-CAP-3.pdf

[iv] https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed