UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

Hybrid causality

towards a model between potential and explicit causality

Dissertation presented by
Vincent DAGNELY

for obtaining the Master's degree in
Software Engineering and Artificial Intelligence

Supervisor(s)
Peter VAN ROY, Manuel BRAVO

Reader(s)
Sana IMTIAZ, Pradeeban KATHIRAVELU

Academic year 2016-2017

Acknowledgment

First, I want to thank my promoter Prof. Peter Van Roy for his help with this master thesis, but also for giving courses so passionately that he made me want to choose a master thesis in the distributed systems field.

Then, I would want to acknowledge my supervisor, Manuel Bravo, who spent a lot of time helping me, from the start to the very end and guided me through my master thesis.

Finally, I would thank also my family, for supporting me.

Abstract

Causal consistency has always been considered as the strongest consistency that achieve good availability and throughput. This consistency allows the user to not see any causal anomalies, which means that updates will always be applied after the one that they could potentially depends on.
However, new applications, like social networks, struggle with causal consistency and need a consistency that will feel the same to the user, while increasing the performance of the system. To solve this, explicit causality propose to let the application decide the dependencies it wants to use.
In this thesis, we will see the model of hybrid consistency, that can use potential or explicit causality in function of what the application require for the operation. We will also present a new system, HybridCassandra, that implement the hybrid consistency while using Cassandra as local datastore. Finally, we will see how to implement this hybrid causality with a simplified version of Facebook.
We ran simulation of our system on Grid5000 and we showed that the hybrid consistency outperforms the causal consistency in our system.

Contents

CHAPTER 1

Introduction

A distributed datastore can be evaluated with five characteristics:

- Availability: All operations issued to the data store complete successfully. No operation can block indefinitely or return an error signifying that data is unavailable.
- Low Latency: Client operations complete "quickly".
- Partition Tolerance: The data store continues to operate under network partitions.
- High Scalability: The data store scales out linearly.
- Consistency.

The four first ones, also known as the ALPS properties, measure the performances of the system. Sadly, they are incompatible with the last one. The strongest the consistency is, the less performing the system is. Usually, Systems go for causal consistency, that uses potential causalities, since it is the strongest consistency that can satisfy the ALPS property.
There are stronger consistencies, but for a user, causal consistency can be considered as strong enough. Indeed, it will prevent any causal anomalies. With a weaker consistency, we could have reference to objects that do not exist, for example a picture in an album that do not exist, and some operations might come in a wrong order. For example, one user could strengthen the privacy of an album, and then add a personal picture in it, and another user, receiving the operations in the wrong order, would be able to see the picture that he should not have the permission to watch. Those are called causal anomalies. Many distributed systems use causal consistency. We could cite COPS [12], GentleRain [7] or Saturn [6], but they are only a small part of all the distributed systems that use causal consistency.

However, causal consistency is not perfect. Recently, social networks have emerged, and they work really badly with causal consistency. Indeed, conversations have usually a really small depth. For example, in twitter, 69% of conversations only had one reply [1], and in facebook, 75% of chains (comments and likes) have depth lower to 3 [8] . This means that causal consistency generate a lot of metadata.
One solution to this problem could be the explicit causality. This causality aims at increasing the performance of the system, while keeping a consistency as strong as the causal one, by moving the responsibilities of the dependencies of the operations, from the system to the application. Since the system is made to work with every application, while the application knows the meaning and the real dependencies of the operations, this should remove the number of dependencies by

operation. However, since it is up to the application to handle the dependencies, it means that it is the responsibility of the application to prevent any causal anomalies, and we will never be able to create an explicit system that prevents causal anomalies regardless of the application.

1.1 Problem statement

In this document, we propose a new model, namely hybrid causality, that aims at solving the above-mentioned problems. The biggest problem of explicit causality is that, in some case, it is really hard to know of what depend an operation, even for the application. For example, if a user creates a new tweet, the application does not really know of what depend this tweet. To solve this problem, hybrid causality offers that when the application knows the dependencies (a reply, a like, ...), the application gives them to the system and the potential causality is used, and when the application does not know, then the application tells the system and the causal consistency is used.
This master thesis tries to address these questions:

- can hybrid causality performs significantly better than potential causality without losing transparency?
- does hybrid causality really reduces metadata?
- can an application easily know the dependencies used for the operations?

1.1.1 Contributions

This master thesis provides the following contributions:

- a literature review over distributed datastore systems with various characteristics (consistency, data model, ...).
- a new consistency model called hybrid causality that aims at increasing the performance and reducing the metadata of a system by using a conjunction of explicit and potential causalities.
- the implementation of hybrid causality in cassandra, under the name of HybridCassandra.
- sound and complete evaluation of HybridCassandra under realistic workloads in Grid5000.

1.2 Thesis overview

This document is organized as follows:
Chapter 2 provides the knowledge required to understand the following of the master thesis. It explain the potential and explicit causality, as well as explaining different distributed datastore systems related to the topic of the consistency or the metadata.
Chapter 3 defines and explains the hybrid consistency.
Chapter 4 defines and explains the system that we have implemented, HybridCassandra.
Chapter 5 discuss the evaluation of this system with the causal one.
Chapter 6 gives a summary of this master thesis and a conclusion.

CHAPTER 2

Related work

There has been a lot of work done in causal consistency, by both theoreticians and practitioners. In this chapter, we discuss the most relevant existing work. First, we differentiate the potential from the explicit causality. Second, we analyze the metadata vs performance tradeoff inherit to causally consistent implementations. Third, we define the properties by which we classify state-of-the-art solutions. Finally, we classify previous solutions based on our taxonomy.

2.1 Potential and explicit causality

Causal consistency, namely *potential causality* prevents causal anomalies. For example, let Alice have a Facebook account with a public photo album. Next, she modifies the setting of the album so that only her friends can see the pictures inside, and then upload a photo inside the album, which only friends are allowed to see. Therefore, any other non-friend user requesting to see Alice photo album should be prevented by the system to observed the latter photo. Another example is if a user Lewis post on his wall that his wife is unconscious at the hospital, then he responds that she is fine after all. Then, a friend Bob sees both posts and replies with a third message saying ´that is wonderful´. If causal consistency is not enforced, Lewis' wife (and any other user) may observe only Lewis' first post, followed by Bob's post missing Lewis' second post and completely changing Bob's intentions and probably generating a conflict among them. In contrast, as system that enforces causal consistency will never allow exposing the third post without exposing the second first.

More precisely, we say that a data store is causally consistent if, when certain update is visible to a client, all its causal dependencies are also visible. We say that two updates a and b such that b causally depends on a (denoted as $a \rightsquigarrow b$), if a and b satisfy one of the following three rules [11, 2]:

- **Execution thread:** if a are b executed by the same client, then $a \rightsquigarrow b$ if operation a happens before b.
- **Gets from:** if a is an update operation and b is a read operation that returns the value written by a, then $a \rightsquigarrow b$.
- **Transitivity:** If it exists another operation c such that $a \rightsquigarrow c$ and $c \rightsquigarrow b$, then $a \rightsquigarrow b$.

When neither $a \rightsquigarrow b$ or $b \rightsquigarrow a$, we say that these operations are concurrent.

Potential causality has the advantage of being transparent to application developers, as the order among operations is exclusively established by the order in which operations are observed by

clients and not based on the application semantics. Nevertheless, this causes potential causality to over-estimate causal dependencies to ensure that no dependency is missed. Nevertheless, most of them are irrelevant from the point of view of the application's semantics. For example, if a user of Twitter go through his news feed, and then decide to tweet, with potential causality, a lot of dependencies will be added to this tweet. But we should ask ourselves, does all the tweets he read related to the one he write (except if he retweets of course)? And for the few that might be, are they really necessary considering the number of dependencies needed?

As a reaction to this problem, *explicit causality* [4, 5]—or application-specified causal dependencies—have emerged. Under explicit causality, the application has to explicitly identify the dependencies that each update is linked to, loosing the transparency property of potential causality but reducing the among of dependencies—and therefore the amount of metadata handled. Notice that reducing the amount of dependencies to the minimum has clear advantages: (i) reducing the storage and computational overhead; and (ii) reducing visibility latencies, providing fresher data to clients. We discuss this consequences in the following subsection.

2.2 The metadata problem

Ensuring causal consistency in a geo-replicated data store requires tracking causal dependencies. In such a system, updates are propagated among datacenters and may arrive in an order that violates causality. Therefore, updates have to piggyback some piece of metadata that allows remote datacenters to install updates in causal order. Unfortunately, precisely tracking all causal dependencies may generate a very large amount of metadata, having an impact on throughput due to the computational and storage overhead [7]. A commonly embraced solution is to compress the metadata to reduce its managing overhead. Although efficient, this technique has a cost: some concurrent operations are unnecessarily serialized, creating false dependencies (or false positives). A false dependency has an impact on the visibility latency of update at remote locations. We define visibility latency as the time interval between the instant in which an update is installed in its origin datacenter and when it becomes visible in remote datacenters. Therefore, the designer of a causally consistent geo-replicated data stores is still faced today with a dilemma: either favor throughput opting for a coarse-grained tracking [7], or favor visibility latencies opting for a more fine-grained approach [3, 12].

2.3 Taxonomies

Before analyzing the most relevant systems of the state-of-the-art, we introduce a set of categories that will help us to classify them:

- **Type of causality:** We classify them based on whether attain potential or explicit causality.
- **Main mechanism:** We classify them based on which is the major technique behind their implementation of causal consistency. We consider four main categories (later describe as the systems are presented): *explicit message checking*, *serializer-based*, *stabilization-based*, and *tree dissemination*.
- **Convergence:** Since causal consistency establishes a partial order among operations, some of them are considered concurrent, and can be shown to clients in either order. This may lead clients (even the same) to observe some events in different order. A causally convergent model, namely *causal+ consistency* [12], prevents this behavior by ensuring that two concurrent operations are totally ordered and enforcing this order when exposing them to clients. There are two major trends: (i) systems that rely on the *last-writer-wins* (*lww*) rule [12, 13, 7] by which updates to a single object are totally ordered; and (ii) systems,

such as Cure [3] and SwiftCloud [17], that rely on high-level data types with rich confluent semantics called *crdts* [15, 16].

- **Metadata structure:** As shown before, the amount of metadata used to track causality has an impact on both throughput and data freshness. Therefore, we classify the systems based on the granularity of the metadata used.
- **Data model:** We consider two data models: *key-value* and *column-based*. The former assumes that data is stored in key-value pairs where key is the data item unique identifier and value it is the piece of data associated to it. The default interface only offers two single-key operations: put and get. The put operation assigns a new value to a given key. The read operation retrieves the stored value of a given key. On the other hand, a column-based data model provides a richer model. Data is stored in columns. Each column is composed by a set of name-based fields and identified by a single key. Through the interface one is capable to not only do column operations (retrieve, overwrite, or delete columns as a whole), but also field operations allowing for a more efficient—fine-grained—data interaction.
- **Read-only tx:** Some systems offer multi-key read operation. This category aims to capture this feature.
- **Write-only tx:** Some systems offer multi-key write operation. This category aims to capture this feature.

2.4 Systems

COPS [12] COPS is a distributed storage system that realizes causal+ consistency and possesses the desired ALPS properties. There exists 2 versions of COPS: The first, named simply COPS, provide causal+ consistency and the second, COPS-GT extends this model by allowing an user to get a consistent snapshot of the data store.
Each datacenter has a local COPS cluster with a complete replica of the stored data. The different clients communicate with a unique COPS datacenter though a library that keeps in memory the versions of the keys seen during the session. Each local COPS cluster is set up as a linearizable (strongly consistent) key-value store. On the other hand, replication between COPS clusters happens asynchronously to ensure low latency for client operations and availability in the face of external partitions. The data store node is a key-value store. Each key-value is associated to a version number. In addition to that COPS-GT stores for each key-value a list of dependencies (each dependence is a key with an associated version number) and maintains old versions of key-value.
Each COPS cluster maintains its own copy of the key-value store.
In COPS, by transitivity, the dependencies that must be ensure are just the nearest ones. Thus, the COPS client library stores only <key, version> entries. For a get operation, the retrieved <key, version> is added to the context. For a put operation, the library uses the current context as the nearest dependencies, clears the context, and then repopulates it with only this put. This put depends on all previous key-version pairs and thus is nearer than them.
When a client calls a put operation, the library computes the set of nearest dependencies. The library then calls the put of the COPS datastore with these dependencies, which assigns to the key a version number and returns it to the client library. The datastore ensures that value is committed to each cluster only after all of the entries in its dependency list have been written. In the client's local cluster, this property holds automatically, as the local store provides linearizability.
After a write commits locally, the datastore asynchronously replicates that write to the other datastore using a stream of put operations. A datastore that receives a put from another datastore must determine if the value's nearest dependencies have already been satisfied locally. It does so

by examining its local state to determine if the dependency value has already been written. If so, it immediately responds to the operation. If not, it blocks until the needed version has been written. If the nearest dependencies are satisfied, the datastore commits the written value.
Clients call the get library function with the appropriate context; the library in turn issues a read to the datastore for the key, to either the latest version, or a specific older one. Upon receiving a response, the client library adds the <key,version> tuple to the client context, and returns the value to the user.
There is 4 operations:

- *createContext* is called when a client connect to the client library, and it creates a new context.
- *deleteContext* is called when a client leave the client library, and it destroys the context of the client.
- *put* is called when the client want to put a value with a key in the key-value store.
- *get* is called when the client want to read the value of a key.

COPS-GT [12] A problem that can occur is that COPS only retrieve value key by key. So, for example, if one user requests the value of a key, and then requests the value of an other key, the value of the second key may have changed between the 2 requests, resulting in an inconsistent view of the datastore. A solution provided is COPS-GT, that allows retrieving a consistent snapshot of different keys. To do so, COPS-GT modifies COPS by needing all the dependencies, and not only the nearest ones, during the put operations. It also requires the datastores to store all the versions of the key, where in COPS, we only keep the latest, as well as the list of all the dependencies, for each version of each key. Then, it offers a new operation, get_trans, that return a consistent snapshot of different keys. To do so, it first gets the last value of each of the keys requested, as well as the corresponding version and the dependencies (returned as a list of <key, version>). Then, for each dependency, the client library check that, if the key is the same as one of the requested ones, the version of the requested is $\geq$ the version of the dependency list. After that, for all the keys where this is not satisfied, the client library resend a get request, but the version requested will be the newest version seen in any of these dependency lists. This give a solution to the problem, but it is very expensive in terms of storage and computation, reducing the scalability.
COPS-GT have the same operations that COPS, instead the get operation that is replaced by *get_trans* operation, which return a consistent snapshot of the keys given in argument.

Eiger [13] Eiger is a system similar to COPS, but built above cassandra, a column-family data model, instead of a key-value. Instead of explaining again COPS, we will list the main differences between COPS and eiger.

- Instead of tracking dependencies on value, eiger tracks dependencies on operations. It increase the eiger's efficiency since, with value, we would have to check for each value of the row, while often, one operation read and write many column, so one row will only need to check dependencies on few operations.
- while COPS has only a write operation, eiger has 3 write operations. The first one is *insert*, which overwrites the current value with a new column. The second is *add*, which increment a counter and the last one is *delete*, which replace the current column with a tombstone.
- eiger implement a system of counter columns, where the column can receive an *add* operation and, instead of overwriting the value, increment it. Since the result depends on all the operation that modify the value of the counter column, to reduce the number of

dependences, eiger only keep one dependency by datacenter, thanks to the linearizability of the datacenters, and only return the nearest dependencies.

The operations of eiger are:

- *batch_mutate* which takes a set of mutation (insert of delete) associated to a key and applies them to the system. If the value already exists, then insert is considered as an update.
- *atomic_mutate* is the same as *batch_mutate* but all the mutations appear as a single atomic group.
- *multiget_slice* which take a set of keys, associated to a column family name and eventually a slice predicate, and return the values associated.

Bolt-on [5] Most of the modern data store provides liveness properties (at some point, the different datastores will converge), but very few offer safety properties. The bolt-on architecture propose to have an eventually consistent data store (ECDS) with strong consistency (like causal consistency), with a bolt-on shim layer above that provide safety properties, without violating the properties of the data store, and communicates with the different clients.
The main challenge with a bolt-on shim layer is that it won't be able to see every writes. Indeed, in a ECDS, the newer versions of a key will overwrite the older versions. So the shim will be able to detect missing dependencies, but will not be able to know if the missing dependencies are needed or not.
With k=w.key the key of a write w, we define a causal cut as a set of writes C such that $\forall$writes w $\in \bigcup_{c \in C}$ c.deps, $\exists$ w' $\in$ C such that w'.key=w.key and w' $\not\rightarrow$ w. In other words, the dependencies of each write belonging to the causal cut should either

- be in the cut.
- precedes causally a write in the cut to the same key.
- be concurrent with a write in the cut to the same key.

The shim should guarantees bolt-on causal consistency: it can deliver a write w to a client only if this write, joined with the set of writes in its local store L, is a causal cut. If so, it can apply w to L. The local store, on the other hand, should always form a causal cut by itself.
When a client performs a write, it must form a causal cut with the local store, since it has only read updates from its local store so the dependencies must be present. Thus, the shim has only to add the write to his local store, and send it to its ECDS.
There is 2 strategies to read:

- causal : When a client performs a read, the shim add the key to a list of keys to check, and return the value stored in its local store. In background, an other method loops forever, and at each loop, it pops every key belonging to this list, gets his new value in the ECDS, and if it forms a causal cut with the local store, updates the local store. If not, it either requests the dependencies to the ECDS, or go to the next key (implementation choice). The problem with this implementation is that the client may experience poor visibility, or time until writes are applied.
- pessimistic : When a client performs a read, the shim will directly ask the value of the key to the ECDS. If it forms a causal cut with the local store, the shim will update the local store, and return this new value, else, the shim will simply return the value of the key stored in the local store. The problem here is the increased time required for each read operation.

The bolt-on architecture presents 2 operations to the client:

- *get* return the value of a key.
- *put* write the value of a key to the datastore after a set of selected dependencies.

SwiftCloud [17] One problem with a distributed data service is that nodes can fail. When a client is attached to a datacenter and this datacenter crash, the client can either wait for the datacenter to be up, or go to an other datacenter. The first solution is clearly problematic, because it could take a lot of time for the datacenter to be up and ideally the user should not be able to see that the datacenter crashed. The other solution solves the problem, but the new datacenter stores an other set of the data and it could present anomalies to the client. For example, the client could have read a version of a variable at the old datacenter, but the new did not receive the update yet, and when the client requests the value at the new datacenter, he would see a previous version of what he already seen.
To solve that problem, SwiftCloud propose a distributed data service where, when a client request for the value of a key, he gets the latest version of his local datacenter that is also stored in K-1 other datacenters, K being a parameter of the system. When a datacenter fails, the client will be able to connect to an other datacenter if there is one that is consistent with his past. If there is no datacenter with this property, then the client will have to wait. We can see that there is a probability that a client will be able to reconnect to a datacenter, and that this probability depends on K. A higher K increases this probability, but updates will take longer to be delivered. In the extreme case, K=N=number of datacenter, the system would not allow any update if one datacenter is unavailable. On the other hand, a lower K improves freshness, but decreases this probability.
SwiftCloud stores updates in a log and transmits them incrementally. To prevent the log to grow indefinitely, SwiftCloud replaces some updates with a checkpoint, that reduce the number of lines in the log. For example, if there is 1000 operations that increment the same value, when it is safe to do, SwiftCloud will replace these 1000 lines by one line that add 1000 to the value.

GentleRain [7] GentleRain is a distributed data services that aim to provide causal consistency with a higher throughput, thanks to clocks. It uses a key-value store, with multiversion. It means that for each key, the datacenters keep several versions, with metadata attached to each of them. There are 4 operations provided to the clients:

- *put* writes the value associated to the key to the system. If the item key didn't exist, a new item is created,. If the key existed, then a new version is attached to the key.
- *get* reads the value associated to the key.
- *get_snpashot* returns a snapshot of the data store for the given keys.
- *get_rotx* does the same thing as *get_snapshot*, but is constrained to include any values read by the client.

To implement it, the system keeps 2 clocks by client c, the dependency time DT_c and the global state time GST_c. The first one is the latest update on every items accessed by the client, and the second one is time at which the client think the system is. The system keeps also a global state time GST_n for each datacenter n.
When a client sends a *get* request, the datacenter returns the latest locally stored version of the key and modifies the clocks. When a client requests a *put* request, the datacenter writes locally, modifies the timestamp, and propagates the update to the other datacenters. When the operation is a *get_snpashot* or a *get_rotx*, then, instead of returning the latest version of the key, it returns the version at the right time, with an algorithm similar to COPS_GT.

Cure [3] Cure is a key-value distributed datastore that aims to provide causal consistent snapshots of the data in only one operation. It handles causality thanks to a causal clock. It means that if two operations are at different causal time, then the operation with the greatest causal time depends on the other operation.
Cure is composed of four elements:

- **Log**: This module stores all the updates that take place in the local datacenter.
- **Materializer**: This module, placed above the Log, generates the versions of the objects out of the updates to the Transaction Manager.
- **Transaction Manager**: This is the main module that handles the client's requests and the transactions.
- **InterDC Replication**: This module is responsible for the propagation of the local updates, found in the Log.

Cure offers 5 operations to the client:

- TxId ← *start_transaction(CausalClock)* starts a transaction at *CausalClock* time.
- Values ← *read_objects(Keys, TxtId)* returns the values associated to the key in the version of the transaction TxId.
- *update_objects(Updates, TxId)* adds a list of updates to the transaction TxId.
- *commit_transaction(TxId)* commits the transaction and make the updates visible.
- *abort_transaction(TxId)* aborts the transaction and delete the list of updates associated to it.

Saturn [6] SATURN is a service that can be used with several distributed and replicated data services to ensure causal consistency, with minimal metadata. To do so, SATURN interacts with the client thanks to a client library. Each client is associated to a datacenter. Saturn communicates to the data services by using a system of labels. A label is a tuple <*type, src, ts, target*>. *Type* is the type of label, it can be an *update*, for when a client send a write operation or a *migration* when a client want to migrate to an other datacenter. *Src* identifies the datacenter, *ts* is the timestamp and *target* can be the data item if it is an update, or the destination datacenter in case of migration. Labels define a total order (there can not be any concurrent label). Since the causal order is partial, it means that there can be several label order that respects the causal order, and thus, the different datacenters can have different labels order, while respecting all the same causal order. Saturn use this property to maximize the performance of the datacenters.
SATURN is composed of 4 elements:

- the frontends that handle the communication between SATURN and the clients.
- the gears, associated to a datacenter, that generate labels, and send data and metadata through the system.
- the label sink that gather the labels and generate the label order.
- the remote proxy sends the write operation to the datacenters in a causal order.

Clients interact with SATURN through 4 operations:

- attach : when the client wants interacting with SATURN, it first sends an *attach* operation with the latest label he has observed. SATURN choses a datacenter, and when the datacenter shares the causal past with the label, the client is attached to this datacenter.

- update : the client library transforms the write from the client to an update operation, tagged with the client label, and sends it to the gear of the local datacenter. The gear generates a new label, sends the update operation to the local datacenter, who writes it, and the label to the label sink. The new label is returned to the datacenter, and replaces the client's old label to update his causal past.
- read : the client library sends the request to the gear of the lacal datacenter, that returns the associated value and label. The value is send to the client through the client library and the label will replace the old label of the client if the new one is greater than the old one.

Occult [14] One major problem of the partitioned causal distributed datastore is the slowdown cascades: having one slow node will slow the entire system. Indeed, if a propagated write operation depends on a version of the value that takes time to arrive, this operation will be delayed too. And all the operations that depend on this delayed operation will also be delayed, resulting in a global delay in the entire system. Occult aims at removing these cascades by moving responsibilities from the system to the client libraries.
In Occult, the datacenters do not wait for dependencies; they immediately update their local store at the reception of each remote update. On the other hand, the client libraries keep in memory a causal timestamp for each client. At each write, the causal timestamp of the client is updated. At each read, the client library compares the timestamp of the client with the one of the node responsible of the given key. If it is safe to read (if the timestamp of the client is greater), the client library forward the value to the client and update the client timestamp. If not, there is several possibilities:

- try again.
- contact a master replica with the latest version of the data item.
- take the risk of causal anomalies and returns the value to the client.

Occult uses a hybrid strategy: It request to the locally the value up to r times, with exponentially increasing delay between each requests), and then, if the value is still not safe to read, it reads from the master replica.
Occult also implements transactions. To prevent slowdown cascades, Occult has lightened the consistency on the transactions. While most systems totally order transaction on the same replica, Occult only orders transaction from the same client session.

2.5 Summary and comparison

In this final section of the chapter, we compare the described systems. We focus on the consequences that their way of implementing causal consistency has in performance. Table 2.1 summarizes them using the previously defined categories.

We have seen many systems. They all have one (or more) objective. They can aim at:

- increasing the throughput.
- reducing the metadata.
- offering a new consistency.
- provide better transactions.
- ...

	Type	Mechanism	Conv.	Metadata	Data Model	RTx	WTx
COPS	potential	message	lww	vector[key]	key-value	no	no
COPS-GT	potential	message	lww	vector[key]	key-value	yes	no
Eiger	potential	message	lww	vector[key]	column	yes	yes
Bolt-on	explicit	message	/	vector[key]	key-value	no	no
SwiftCloud	potential	sequencer	crdts	vector[dc]	key-value	yes	yes
GentleRain	potential	stab.	lww	scalar	key-value	yes	no
Cure	potential	stab.	crdts	vector[dc]	key-value	yes	yes
Saturn	potential	tree	no	scalar	key-value	no	no
Occult	potential	message	lww	vector[server]	key-value	yes	?

Table 2.1: Summary of causally consistent systems. *Conv.* stands for convergent, *RTx* for Read-only tx; and *WTx* for Write-only tx

To achieve this objective, they use different mechanisms, data models, algorithms, ... At the end, we can not really say that one system is better than an other. They all provide a unique property and it is up to the developer to find the best system in regard of what he needs.

CHAPTER 3

Hybrid causality

3.1 Motivation

We have seen that the potential causality allows to prevent any causal anomalies, at the cost of many dependencies at each operation. We have also seen that the explicit causality can increase the throughput, but may be hard to implement and any error can be costly. We will now present the hybrid causality. It is a causality that is a mix of them, and gathers the advantages of both of them. The point is to use explicit causality when the explicit dependencies are obvious, and potential when they are not. To better understand the advantage of the hybrid causality, we will now present a scenario that will be use to show the advantage of hybrid causality. Let's imagine that Alice logs in Facebook and reads her wall. Among the many posts she sees, she is interested on a post B made by Bob, to which she replies with message C.
With potential causality, the message C of Alice will depend not only on B, but also in all messages read in her wall. This is not necessary. Explicit causality solves this problem. With Explicit causality, message C of Alice will only depend on B.
Now, let's say that Bob opens his wall, and instead of replying to a post, decides to post something new on his wall. We do not have an explicit dependency here, the application can not know from what depends this new post. The only solution would be to give up consistency. But potential causality includes everything read, so the new post would not allow any causal anomaly under potential causality.
This is where hybrid causality solves the problem. It will use explicit causality when the application knows the dependencies (like in the first example) and potential causality in the other cases (like in the second example).

3.2 Model

We have modeled hybrid causality assuming a column-based data model, which can be found in well-known data stores, such as Cassandra [10] and Bigtable [9] because it was the one used by facebook, but hybrid causality is not restricted to this model and could by modeled under simpler data models, such as the key-value model, or more complex, such as SQL-like models.

3.2.1 Column-based data model

This model stores the data with tables, in which a row is composed of a key and of an object. This object contains the data associated to the key. It is composed of columns objects of which

there are 2 types:

- simple columns: they only contain the name of the column and the value. The value can be a String, an int, a float, ... or even can be null.
- super columns: they contain a set of simple columns objects. They can be used to store data when the number of data depends on the row (for example, to store the id of the replies to a post, one should have one super column containing one simple column by reply, with the id of the reply as the name of the column, and no value).

The columns in each row are unique. It means that, inside a table, the name of the columns and the number of columns can vary from a row to an other. Columns can also be sorted by time or by name.

3.2.2 Modeling hybrid causality

A system that supports hybrid causality should implement (at least) 2 write operations: one with a set of dependencies as argument (*write_expl*), and one without this extra argument (*write_pot*). The first one should be used when explicit causality is required, and will only use the set of dependencies given as argument when checking if the datacenter can apply the operation. The second one should be used when potential causality is needed.
Ideally, we want to maximize the number of operations that use write_expl, and only use write_pot in last resort. Usually, the best way to find explicit dependencies is to search for chains. For example, a reply to a post is a chain, and the reply should explicitly depend on the post. We can also find chains in likes, retweet or conversations, and the explicit dependencies become obvious. With the same strategy, the potential should be use at the creation of the chain (posting a new tweet, starting a conversation, ...).

CHAPTER 4

HybridCassandra

The domain in which explicit causality should bring the more benefice is the social network. That is why we decide to implement a simplified version of facebook to compare causal consistency with explicit consistency. We implement our system above Cassandra and using his interface to communicate with the layers we have added.

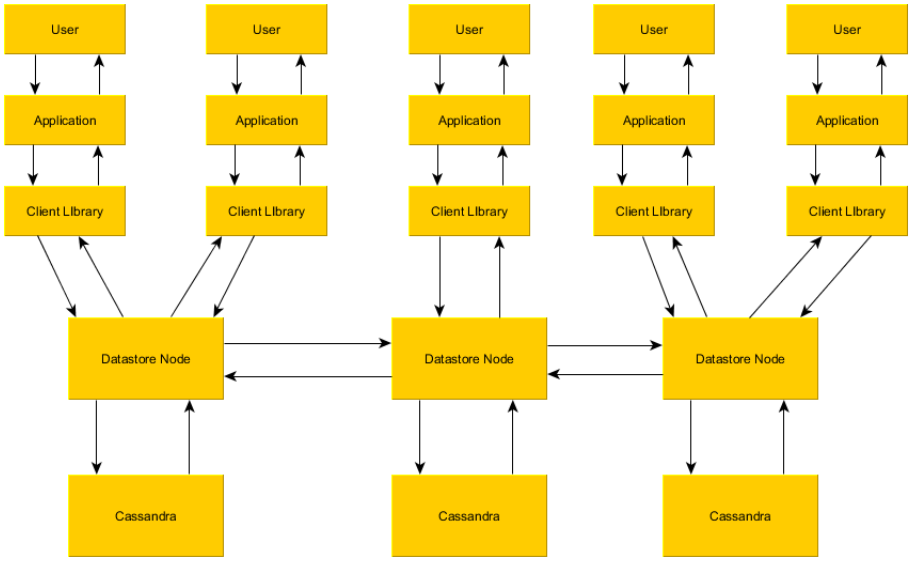

Figure 4.1: architecture of the application

Starting from the bottom, the foundations are the cassandra datastores. Above cassandra, we have the datastore nodes. Each datastore node has a unique cassandra with which they communicate to write and read data. The datastore nodes propagate the writes they receive from the client library to the other datastore nodes, which will check the dependencies and then apply the write when all the dependencies have been applied. Then, there are the client libraries, which store the context of each client and forward the write and read from the application. The client library is in charge to calculate the dependencies required when the potential causality is used. Finally, the application provides a service to the client by accessing the data system through the client library.

4.1 Cassandra

Cassandra is a data storage focused on availability and scaling, by using multiple nodes at each datacenter. It was developed by facebook to allow users to search through their Facebook inbox and was launched in June of 2008. It uses the column-family model. Every operation under a single row key is atomic per replica. Each row has columns, that can be simple columns, which means they contain a value, or super columns, which means they contain simple columns. Columns can be sorted by time or by name.
Cassandra offers three operations:

- *insert(table,key,rowMutation)*
- *get(table,key,columnName)*
- *delete(table, key, columnName)*

Cassandra partitions data across the different nodes of the cluster thanks to a hash function. Each node is assigned a random value and is responsible for the key whose the hashed value is between its value, and the value of the next node. We call this node the coordinator of the key. The data item will also be stored in N-1 other nodes in the cluster, with N being the degree of replication. There are three strategies to choose these N-1 nodes:

- Rack Unaware
- Rack Aware
- Datacenter Aware

The first strategy is the simplest: the coordinator just chooses the N-1 next nodes in the ring as the replicas. The two following are more complex. A node is elected as the leader. When a new node joins the cluster, it contacts the leader, who assigns to it the range in which the node will replicate data items. The leader will ensure that no node is responsible for more than N-1 ranges in the ring.
For write operations (insert and delete), the cluster forward the request to the replicas, and wait for a quorum to acknowledge the operation. For read operations (get), in function of the system chosen, the cluster can either send the requests to the closest replica, or send it to all, and wait for a quorum to respond.

4.2 Multi-dc cassandra

Cassandra is a non-distributed datastore system. Thus, we first had to add a layer to propagate the writes to the other datacenter. We decided to implement a cops-like system, since it is simple, and consequently perfect for comparing the two consistencies. As we had to compare causal with explicit, we had to create two systems: one who guarantees causal consistency, and one who guarantees hybrid consistency. We didn't implement a full explicit consistency system because, as we explained in the previous chapter, it is impossible to find the explicit dependencies for each operation.

4.3 Causal cassandra

As in cops, client communicates to the system through a client library. This client library offers five operations:

- *ctx_id* ← createContext()

- bool ← deleteContext(*ctx_id*)
- bool ← put(*table,key, rowMutation, ctx_id*)
- columnFamily ← get(*table, key, columnName, ctx_id*)
- bool ← delete(*table,key, columnName, ctx_id*)

The two first ones are exactly the same as cops. The three last ones are the operations proposed by cassandra, with their context. With these five operations, the client library can compute the dependencies of any operation.
The client library transforms the put as input into a put_after(*table, key,rowMutation, deps, version*=∅), where deps is the set of the nearest dependencies, and version is always null. The client library sends the put_after to the data store node, who send it immediately to cassandra, since the dependencies must be hold. Then, the data store node send a put_after(*table, key, rowMutation, deps, version*) with the same argument, except the version which is the key's version number. When it arrives at the remote datacenters, they execute dep_check(*key,version*) on each element of deps, and as soon as all the dep_check operations succeed, the remote datacenters send the write operation to cassandra.
The delete operation follows the exact same path, but with the columnName instead of rowMutation as argument.
The client library also forwards the get operation by only removing the context argument to the data store node, who send it to cassandra, and the answered columnFamily returns to the client by the opposite path.

```
function put_after(table, key, rowMutation, deps, vers):
    if(vers=={}):
        key_vers=find_version(key,table)
        for datastore_node in datastore_nodes:
            send(datastore_node, put_after(table, key, rowMutation, deps,
                  vers=key_vers))
        insert(table,key,rowMutation)
    else:
        for dep in deps:
            check_dep(dep)
        insert(table,key,rowMutation)

function get(table, key, columnName):
    get(table, key, columnName)
```

Figure 4.2: datastore node (part 1)

```
function delete_after(table, key, columnName, deps, vers):
    if(vers=={}):
        key_vers=find_version(key,table)
        for datastore_node in datastore_nodes:
            send(datastore_node, delete_after(table, key, columnName, deps,
                  vers=key_vers))
        delete(table,key,columnName)
    else:
        for dep in deps:
            check_dep(dep)
    delete(table,key,columnName)
```

Figure 4.3: datastore node(part 2)

```
function put(table, key, rowMutation, ctx_id):
    deps=find_dependence(ctx_id)
    put_after(table, key, rowMutation, deps, {})
    set_new_dependency(ctx_id, ''put'', table, key)

function get(table, key, columnName, ctx_id):
    get(table, key, columnName)
    set_new_dependency(ctx_id, ''get'', table, key)

function delete(table, key, columnName, ctx_id):
    deps=find_dependence(ctx_id)
    delete_after(table, key, columnName, deps, {})
    set_new_dependency(ctx_id, ''delete'', table, key)
```

Figure 4.4: causal client library

4.4 Hybrid cassandra

The hybrid cassandra is similar to the causal cassandra. The only difference is the client library. In addition to the five operations of the causal cassandra, the hybrid client library offers two additional operations:

- bool ← put_explicit(*table,key, rowMutation, ctx_id, deps*)
- bool ← delete_explicit(*table,key, columName, ctx_id, deps*)

where deps are the explicit dependencies that will overwrite the nearest ones when the client library will call put_after. Since ctx_id is still given, the client library can add the operation to the potential tree. When the potential put operation is called, the client library will compute the nearest dependencies and give them as argument in the put_after or delete_after. As explained previously, there are different way to implement the algorithm to compute the nearest dependencies: causal, where it just computes the potential nearest dependencies, and hybrid, where the explicit relations are used to remove some keys from the set of the nearest dependencies.

```
function put_explicit(table, key, rowMutation, ctx_id, deps):
    put_after(table, key, rowMutation, deps, {})
    set_new_dependency(ctx_id, ''put'', table, key)

function delete_explicit(table, key, columnName, ctx_id, deps):
    delete_after(table, key, columnName, deps, {})
    set_new_dependency(ctx_id, ''delete'', table, key)
```

Figure 4.5: hybrid client library

CHAPTER 5

Evaluation

The main goal of our evaluation is to determine if hybrid causality can improve latency and throughput in geo-replicated systems, when compared to potential causality without compromising transparency. For this, we run HybridCassandra, our prototype, under:

- Synthetic workloads that allow us to explore how the different parameters that characterize a workload impact the performance.
- A benchmark based on the Facebook's dataset, to obtain an assessment of HybridCassandra under complex realistic workloads.

For each experience, we compare our prototype with a causal version, using the same operations, nodes and parameters.

5.1 Experimental setup

The evaluation has been done on grid5000, with three nodes composing a unique datacenter, located at Luxembourg, Rennes, and Sophia Antipolis. The first node has an Intel Xeon E5-2630L CPU, with 32GB of memory. The second one has an Intel Xeon E5-2630v3 CPU, with 128GB of memory. And the last one has a AMD Opteron 2218 CPU, with 4GB of memory. Each client generates frequently random operations. In case of a write, the client chooses between the write operations (that can be found in appendix) with a uniform probability. In case of a read, the client chooses between a set of read operations (that can be found in appendix) with a uniform probability.

5.2 Microbenchmarks

For the first microbenchmarks, we run the causal and hybrid models with a varying ratio of write to read. The write operations are chosen uniformly from insert, delete and update operations. The ratio goes from 64 get for 1 write to the reverse.
On the left graph, we can see the average latency by operation. The first thing we can see is that the hybrid causality performs strictly better than the causal one, as we predicted when we explained the model of hybrid causality. We can also notice that the more write there is for each get, the more latency there are. Since the gets do not need to wait, even if the write has more dependencies to wait, it makes sense that the average latency is proportional to the number of write. On the right graph, we can see the global throughput. For the same reason than for the

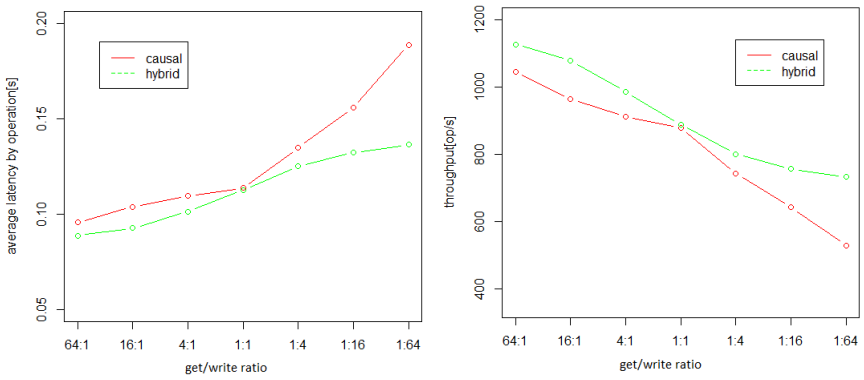

Figure 5.1: average latency by operation and throughput in function of the ration insert/delete

latency, the hybrid model performs better, and the throughput decreases when the ratio of write increases.
The second benchmark run the same simulation, but instead of varying the ratio of write and reads, we vary the number of possible keys by table. For recall, during a write or get operation, the benchmark choses a random table, and then choses a random key in the table (all the tables have been populated before to have the exact number of row desired). The number of keys by table goes from 1 to 1024.

As for the previous microbenchmark, the hybrid model performs strictly better than the causal

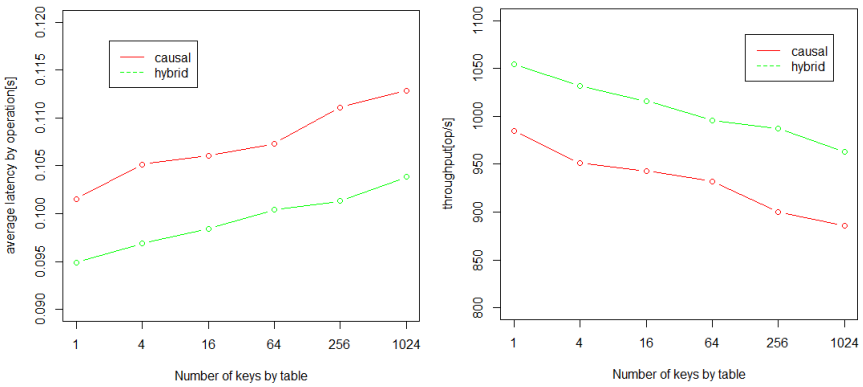

Figure 5.2: average latency by operation and throughput in function of the number of keys by table

one in terms of latency and throughput. We can also notice that the latency increases and the throughput decreases when the number of keys by table increase. It might feel strange, because intuitively, we would say that since the operations are spread across different keys, they should wait for a shorter time and we should see the reverse result. But in reality, when the number of keys by table increases, it means that the operations will depends on more values, increasing the number of dependencies by operation, and thus, increasing the latency and reducing the throughput of the system.

5.3 Use case: a social network

The application simulates users that surf on facebook, and communicate to each other by interacting to the datastore through the client library. The simulation generates random

operations, with a probability of 2‰ to be a write and a probability of 998‰ to be a multi-round read. The write operations are the same as in the previous section. The multi-round reads are chosen uniformly between the set of multi-round operations (that can be found in appendix). Usually, the multi-round operations are composed of two rounds: the first ones fetch one row in a table, and the seconds fetch all the row in an other table related to the results of the first round. For example, one multi-round operation consists of fetching the wall of someone, getting all the commentId in the commentsOnWall super column, and fetching all the rows in the Comments table that belong to the wall.
We have tested the simulation on HybridCassandra and on a causal version of it, while varying the number of client by datacenter. On the left graph, we can see the average latency by operation.

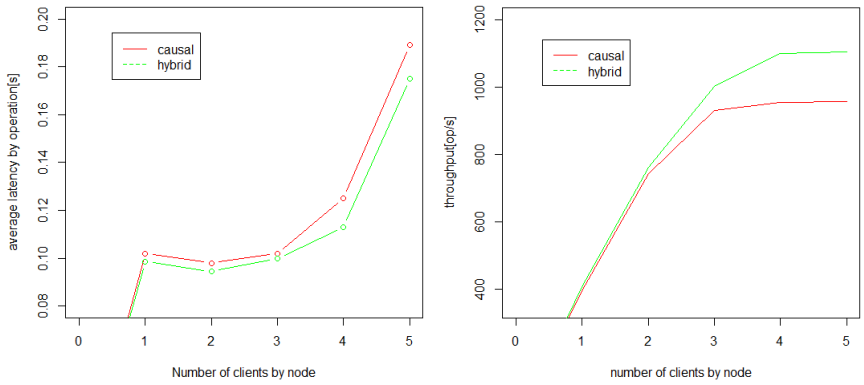

Figure 5.3: average latency by operation and throughput in function of the number of client by node

On the right graph, we can see the global throughput. We can see that the hybrid model outperforms the causal one. We can also notice that the hybrid model can handle a heavier workload before struggling to handle the requests.

CHAPTER 6

Conclusions

6.1 Conclusions

We have presented a new consistency model, namely hybrid causality, that aims at reducing latency and increasing throughput of distributed datastore systems by allowing the application to use its own dependencies when it knows of what depends the operation. This new consistency model was supposed to be better mainly in social network, where the dependencies of a lot of operations are obvious.
In order to assess its benefits, we have implemented from Cassandra, a column-family datastore, a new distributed system and an application that respects the hybrid consistency.
Our evaluation shows that our suppositions were true. In social network, hybrid causality increases the performance of the system and allows a greater workload for the same number of datacenters.

6.2 Future work

However, as we have seen in chapter 2, Hybrid causality is not the only solution found to increase the performance of a system. Many systems can provide a better version than COPS, either by strictly increasing his performance, or by providing a more complete interface. One would want to compare hybrid causality with these systems and see if the hybrid causality still beats these more complex systems. And, in the case it does, evaluates if the additional time required to implement the hybrid causality is worth the increased performance. One should also try to create a new hybrid version from these systems, instead of comparing from a cops-like system. We have no doubt that in social network, hybrid causality will outperform any causal system, but in other domains, it is less likely.

Appendix

Data model

You can find the data model in the tables below, with each table being a table in the data model. The first row of the table is the name of the super column families. The second row is reserved to the column families. If they are below a super column family, it means they belong to that super column family. And the last row is first the key, and then the value to each column family. A null means that there will never be any value in the column family. The table contains all the information required to understand the data model, but if one has a doubt and want to check his understanding of the datamodel, we will describe each table one by one.
The first table is the Comments table. There is one row for each comment. The row contains 3 column families: the comment column that contains the comment itself, as a String, the picture column, which contains the id of the picture, and the video column, that contains the video. The row also has 4 super column families, each with null values. The first one is the related comments. Each column family is the id of the comment that replied. The second one is the persons who liked the comment, with each column family being the id of the persons. The third one is the hashtags. Each column family is a hashtag. And the last one is the persons to which the comments are destined, with each column family being the id of the persons.
The second table is the Albums table. There is one row by album. The row contains the name column family, with the name of the album as a string, and a super column family with each column family being the id of the pictures inside the album, and no values.
The third table is the Pictures table. Each row is a picture that is inside an album or a comment. It contains the picture column family, with the picture itself. It also contains 3 super column families, with no values: the persons who liked, with their id, the persons tagged on the picture, with their id, and the comments posted on the picture, with their id.
The fourth table is the Walls table. Each row is the wall associated to a person. It has only one super column family: the comments posted in the wall, with their id as column family, and no values.
The fifth table is the Groups table. Each row is a group. It has 2 super column families: commentsOnGroup, with the comments posted on the wall of the group as column families, and personsOnGroup, with the id of the persons in the group as column families. Both of them have null values.
The sixth table is the Profiles table. Each row is the profile of a person, associated with his id. It has one column family, profileProperty, with the property of the profile. It also have the super column family albums, with the id of the albums, and the super column family conversations, with the id of the conversations. Here again, they have null values.
The seventh table it the Conversations table. Each row is a conversation. It contains 2 super

column families. The first one is personsOnConversation, with the id of the persons in the conversation. The second one is messages, with the id of the messages as column family, and the String message, a picture, or a video as value for each column family.
The eighth table is the Settings table. It is a really small table, with one row by person, and only one column family, settings, with the settings as value.
The last table is the Messages table. It contains one column family, who can be either message, with the corresponding String message, either picture, with the id of the picture, or video, with a video. The 2 super column families, with null values, are hashtags, with the hashtags as column families, and persons@, with the persons tagged.

Comments		relatedComments			personsWhoLiked					hashtags			persons@		
	comment	#commentId1	#commentId2	...	#personId1	#personId2	...	picture	video	#hashtag1	#hashtag2	...	#personId1	#personId2	...
commentId	String	null	null	null	null	null	null	pictureId	video	null	null	null	null	null	null

Albums		pictures		
	name	#pictureId1	#pictureId2	...
albumId	String	null	null	null

Pictures		personsWhoLiked			personsOnPicture			comments		
	picture	#personId1	#personId2	...	#personId1	#personId2	...	#commentId1	#commentId2	...
commentId	picture	null	null	null	null	null	null	null	null	null

Walls	commentsOnWall		
	#commenIdt1	#commentId2	...
persontId	null	null	null

Groups	commentsOnGroup			personsOGroup		
	#commentId1	#commentId2	...	#personId1	#personId2	...
commentId	null	null	null	null	null	null

Profiles		albums			conversations		
	profileProperty	#albumId1	#albumId2	...	#conversationId1	#conversationId2	...
personId	profileProperty	null	null	null	null	null	null

Conversations	personsOnConversation			messages		
	#personId1	#personId2	...	#messageId1	#messageId2	...
conversationId	null	null	null	null	null	null

Settings	
	settings
personId	settings

Messages				hashtags			persons@		
	message	picture	video	#hashtag1	#hashtag2	...	#personId1	#personId2	...
messageId	String	pictureId	video	null	null	null	null	null	null
XOR									

Operations

The operation can be found in the table below. There are first the write operation, with the name, the dependency and the description. Potential means that we use the potential causality. Else, the key to which the operation depends is written. Then, there is the table of the read operations, with the name and the description.

writes		
name	dependency	description
postOnWall	potential	post a message on the user's wall
comment	the post/picture/.... on which the comment is done	comment on a post/picture/...
createAlbum	potential	create a new album
addPicture	the album in which the picture belongs	add a picture to the album
addFriend	potential	send an invitation to be friend
createGroup	potential	create a new group
addPersonToGroup	the group	add a person to a group
postOnGroup	the group	post a message on the group's wall
like	the post/picture/.... on which the like is done	like a post/picture/...
updateComment	the comment itself and the post/picture /... on which the comment is done	update a comment
updateProfile	potential	update a profile
removeComment	the comment itself	remove a comment
removePicture	the picture itself	remove a picture
removeFriend	potential	remove a friend
removeAlbum	the album itself	remove an ablum
sendMessage	if new conversation: potential, else: the message above	send a message to one or more people
refuseFriend	addFriend	refuse the invitation to be friend
acceptFriend	addFriend	accept the invitation to be friend
updateSetting	the setting	update the setting of a user
updateAlbum	the album itslef	update an album
updateGroup	the group itself	update a group

The first one is getWall, for when a user looks at the wall of a person. At the first round, we read the wall of a person. Then, for each comment in the wall, we read the row associated to it in the Comments table.
The second one is getAlbums, for when a user looks at the albums of a person. At the first round, we read the profile of the person. Then, we get the list of the id of the albums of this person, and we read for each album his associated row in the Albums table.
The third one is getPicture, for when a user looks at a picture. At the first round, we read the row in the Pictures table associated to the picture. Then, for each comment of the picture, we read it in the Comments table.
The fourth one is getGroup, for when a user looks at a group. At the first round, we read the information of the group in the Groups table. Then, for each comment in the group, we read the row associated to it in the Comments table.
The fifth one is getConversations, for when a user opens his list of conversations with his friends. At the first round, we read the profile of the person. Then, we get the list of the id of the conversations of this person, and we read for each conversation his associated row in the Conversations table.
The sixth one is getMessages, for when a user opens a conversation. From the previous multi-

read	
name	description
readComment	get a row in Comments
readAlbum	get a row in Albums
readPicture	get a row in Pictures
readWall	get a row in Wall
readGroup	get a row in Groups
readProfile	get a row in Profiles
readConversation	get a row in Conversations
readMessage	get a row in Messages
readSetting	get a row in Settings
readPicture	get a row in Pictures

Multi-round operations

There are seven multi-round operations:

GetWall	
round1	readWall
round2	∀ commentId: readComment

getAlbums	
round1	readProfile
round2	∀albumId: readAlbum

getPicture	
round1	readPicture
round2	∀commentId: readComment

getGroup	
round1	readGroup
round2	∀commentId: readComment

getConversations	
round1	readProfile
round2	∀conversationId: readConversation

getMessages	
round1	∀ messageId: readMessage

getSettings	
round1	readSetting

round operation, we already have the information of the conversation. We only need to read, for each message of the conversation, the row associated to it in the Messages table.
The last one is getSettings, for when a user wants to access his settings. Here, we only need one round where the user read his setting from the Settings table.

Bibliography

[1] C. Cherry A. Ritter and B. Dolan. Unsupervised modeling of twitter conversations. In *HLT 2010*, pages 172–180, 2010.

[2] Mustaque Ahamad, Gil Neiger, JamesE. Burns, Prince Kohli, and PhillipW. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[3] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *Proceeding of the IEEE 36th International Conference on Distributed Computing Systems*, ICDCS '16, pages 405–414, 2016.

[4] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SoCC '12, pages 22:1–22:7, San Jose, California, 2012.

[5] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, New York, USA, 2013.

[6] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. Saturn: A distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 111–126, Belgrade, Serbia, 2017.

[7] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the 5th ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13, Seattle, WA, USA, 2014.

[8] C. Marlow E. Sun, I. Rosenn and T. Lento. Gesundheit! modeling contagion through facebook news feed. In *ICWSM 2009*, 2009.

[9] Sanjay Ghemawat Wilson C. Hsieh Deborah A. Wallach Mike Burrows Tushar Chandra Andrew Fikes Fay Chang, Jerey Dean and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *In Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, pages 205–218, 2006.

[10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. In *ACM SIGOPS Operating Systems Review*, pages 35–40, 2010.

[11] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), July 1978.

[12] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, Cascais, Portugal, 2011.

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

[14] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, 2017.

[15] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, January 2011.

[16] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Grenoble, France, 2011.

[17] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 75–87, Vancouver, BC, Canada, 2015.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**