Research Online Publishing
www.ronpub.com

Open Access

*Open Journal of Databases (OJDB)*
*Volume 2, Issue 1, 2015*

http://www.ronpub.com/ojdb
**ISSN 2199-3459**

# Causal Consistent Databases

Mawahib Musa Elbushra [A], Jan Lindström [B]

[A] College of Graduate Studies, Sudan University of Science and Technology,
Steen Street block 85 house no. 207 , 111 Khartoum, Sudan, mawahib.elbushra@hotmail.com
[B] MariaDB Corporation, Tekniikantie 12, FIN-02150 Espoo, Finland, jan.lindstrom@mariadb.com

## Abstract

*Many consistency criteria have been considered in databases and the causal consistency is one of them. The causal consistency model has gained much attention in recent years because it provides ordering of relative operations. The causal consistency requires that all writes, which are potentially causally related, must be seen in the same order by all processes. The causal consistency is a weaker criteria than the sequential consistency, because there exists an execution, which is causally consistent but not sequentially consistent, however all executions satisfying the sequential consistency are also causally consistent. Furthermore, the causal consistency supports non-blocking operations; i.e. processes may complete read or write operations without waiting for global computation. Therefore, the causal consistency overcomes the primary limit of stronger criteria: communication latency. Additionally, several application semantics are precisely captured by the causal consistency, e.g. collaborative tools. In this paper, we review the state-of-the-art of causal consistent databases, discuss the features, functionalities and applications of the causal consistency model, and systematically compare it with other consistency models. We also discuss the implementation of causal consistency databases and identify limitations of the causal consistency model.*

## Type of Paper and Keywords

Research review: *causal consistency, consistency models, distributed databases*

## 1 Introduction

Distributed fully replicated databases provide copies of the same data on several, geographically distributed locations. For example, Facebook [23] distributes its data (profiles, friends lists, likes, and so on) on multiple data centers on the East and West coast of the United States of America and in Europe. The distribution brings two key benefits to services: fault tolerance and low latency. Fault tolerance is provided through redundancy: if one of the databases fails, others can continue to provide service. Low latency is provided by proximity: clients can be directed to and served by a nearby data center. Distribution naturally brings its challenges, however. The famous CAP theorem, conjectured by Brewer [26] and proved by Gilbert and Lynch [47], states that it is impossible for a distributed system to simultaneously provide all three of the following CAP guarantees:

- Consistency: all nodes see the same data at the same time.
- Availability: a guarantee that every request receives a response about whether it was successful or failed.
- Partition tolerance: the system continues to operate despite arbitrary message loss or failure of part of the system.

According to the theorem, a distributed system can satisfy any two of these guarantees at the same time, but not all three. Consistency as formally proven is a property known as linearizability [47].

The principle of consistency is similar to the atomicity of ACID properties guaranteeing that database transactions are processed reliably. Each transaction will be atomic in strictly consistent databases [41]. On the flip side, if a database is not strongly consistent, then different nodes may have different views of the same data [1].

The availability principle means that services provided by the distributed system are entirely available at all times [66]. There is an important notion about the response time associated with this principle. A highly available system avoids delays in responding to the queries of users [12]. Availability ensures when parts of nodes in a distributed system become inaccessible as a result of failures, the other nodes should continue to operate [86]. It is important that intended responses are received for each request even if other parts of the system fail [74]. One of the main reasons for distributing a system is to provide high availability, as more nodes join the system and they share some data; the system becomes more tolerant to particular node failures.

The partition tolerance is achieved when a distributed system is built to allow arbitrarily loss of messages sent from one node to another [47]. The requirement of availability makes it impractical to keep all data at one source. This is because when the source fails, the entire system becomes unavailable. Therefore, the partition tolerance allows for system states to be kept in different locations [10]. In the case of a distributed database, if it is partition-tolerant then it will still be able to perform read/write operations while partitioned. If it is not partition-tolerant, when partitioned, the database may become completely unusable or only available for read operations [34].

CAP summarizes trade-offs [91] from decades of distributed-system designs and shows that maintaining a single-system image in a distributed system has a cost [50]. If processes in a distributed system are partitioned then updates cannot be synchronously propagated to all processes without blocking. Under partitions, a system cannot safely complete updates and hence presents unavailability to some or all of its users. Moreover, even without partitions, a system that chooses availability over consistency enjoys benefits of low latency: if a server can safely respond to a user's request when it is partitioned from all other servers, then it can also respond to a user's request without contacting other servers even when it is able to do so [25]. Sacrificing the partition tolerance is not an option as noted [52]. The choice is between consistency and availability.

Therefore, in recent years there has been growing interest on weaker consistency models, which can operate on the presence of network partitions [35]. An advantage of weaker consistency models is increased performance potential [75, 54, 92]. Their disadvantage is more complex programming models for implementation [45, 46]. Weaker consistency levels are useful in such systems, where transactions either acquire fewer locks, or hold them for shorter time [5]. This situation leads to less delay, since less transactions attempt to access the same objects in conflicting lock modes. Additionally, the danger of deadlock is decreased [4]. The benefits achieved from weaker consistency models are also important for many applications in distributed systems [72] and in mobile systems [29, 76], such as the services like Facebook [23], Google, LinkedIn, Twitter, Yahoo [32], auctions and Massively Multi-player Online Role-Playing Games (MMORPG). Therefore, it is desirable to allow application programmers to take advantage of the weaker levels (when this makes sense) and trade off the consistency for better performance [3].

In [38] we reviewed the state-of-the art of the databases that use eventual consistency. Based on that review we discussed the advantages and disadvantages of the eventual consistency model. The review showed that there are several mature and popular database systems using eventual consistency. Most of these are actively developed and there is a strong community behind them. In [38] we concentrated on the eventual consistency, whereas this paper focuses on the causal consistency model. In this paper we make following contributions:

- We synthesize the necessary theoretical background to understand the causal consistency model.
- We discuss the features, functionalities, applications and limitations of the causal consistent model and present different causalities.
- We also systematically compare the causal consistency with other consistency models, and provide an insight on differences into functionalities, requirements and limitations between them.
- We present possible methods of implementing causal consistency, and conclude that it is significantly harder to implement causal consistency than eventual consistency. This explains the fact why there is not even a single commercial database system that uses causal consistency.
- We identify the problems caused by different consistency models by real-word examples.
- We also discuss the applications and database systems, which may employ the causal consistency model.

This paper is organized as follows. In section 2 we present the necessary theoretical background by definitions and examples. Section 3 presents the causal consistency and section 4 compares serializability, eventual and causal consistency using a running example.

In section 5 we present one possible method to implement causal consistency. Section 6 contains a discussion on no-causality, local causality and global causality, and presents some examples where both eventual consistency and causal consistency have different problems. In section 7 we present potential applications and databases that could use the causal consistency as consistency model. Finally, section 8 concludes this paper.

## 2 Theoretical Background

Many distributed applications require highly available data, e.g. Facebook [23], Google, LinkedIn, Twitter, Yahoo [32], auctions and massively multi-player online role-playing games (MMORPG). Data is highly available if it can be accessed at any time. The connectivity of the network affects the availability of data. For example, data can become unavailable when network partitions occur. To increase availability, data is often replicated. This allows an application to access data replicas that are present locally, or are present in a local partition. Accessing distributed or replicated data introduces the data consistency problem [82]. The consistency of replicated storage systems can be defined based on the ordering of read and write operations that each node observes.

In Table 1 we have summarized important symbols and notations used later, in order to help readers easily follow the technical details.

**Table 1: Notations used**

| Notation | Short description |
|---|---|
| R | Relation |
| $\sim$ | Reflexive relation |
| $\prec$ | Total order relation |
| $\preceq$ | Partial order relation |
| $r_n[x] = v$ | A read operation $r$ of the transaction $n$ reads from a data item $x$ the value $v$. The transaction $n$ and value $v$ may be omitted if not needed. |
| $w_n[x] = v$ | A write operation $w$ of the transaction $n$ writes a value $v$ to the data item $x$. The transaction $n$ and value $v$ may be omitted if not needed. |
| $o_n$ | Read or write operation of the transaction $n$ |
| $H$ | History produced by a set of transactions |
| $a \rightarrow b$ | An event or operation $a$ happens before the event or operation $b$ |

To reason about the consistency guarantees, we need the concept of order with respect to time.

**Definition 1 (Antisymmetry):** *a binary relation $R$ on a set $X$ is antisymmetric if there is no pair of distinct elements of $X$ each of which is related by $R$ to the other. Formally: $\forall a, b \in X, R(a,b) \wedge R(b,c) \Rightarrow a = b$.* □

For example, the relation $<$ (less-than) is antisymmetric in natural numbers because $\forall x, y \in \mathbb{N}$ : if $x < y \Rightarrow y \not< x$.

**Definition 2 (Transitivity):** *a binary relation R over a set X is transitive if whenever an element a is related to an element b, and b is in turn related to an element c, then a is also related to c. Formally: $\forall a, b, c \in X, aRb \wedge bRc \Rightarrow aRc$.* □

For example, the greater-than relation is transitive, i.e. whenever $A > B$ and $B > C$, then also $A > C$. Transitivity is a key property of both partial order relations and equivalence relations.

**Definition 3 (Reflexivity):** *a reflexive relation is a binary relation on a set for which every element is related to itself. In other words, a relation $\sim$ on a set S is reflexive when $x \sim x$ holds true for every x in S. Formally: when $\forall x \in S : x \sim x$ holds.* □

The equal relation on the set of real numbers is an reflexive relation since every real number is equal to itself.

**Definition 4 (Total order):** *A relation $\prec$ is a total order on a set $S$ if it has:*

1. *Antisymmetry: $\forall a, b \in S$ if $a \prec b \wedge b \prec a \Rightarrow a = b$.*
2. *Transitivity: $\forall a, b, c \in S$ if $a \prec b \wedge b \prec c \Rightarrow a \prec c$.*
3. *Totality: $\forall a, b \in S$ either $a \prec b$ or $b \prec a$.*

□

An example of total order is letters in an alphabet ordered by the standard dictionary order, e.g., $A < B < C$ etc.

**Definition 5 (Partial order):** *A relation $\preceq$ is a partial order on a set $S$ if it has:*

1. *Reflexivity: $\forall a \in S, a \preceq a$.*
2. *Antisymmetry: $\forall a, b \in S$ if $a \preceq b \wedge b \preceq a \Rightarrow a = b$*
3. *Transitivity: $\forall a, b, c \in S$ if $a \preceq b \wedge b \preceq c \Rightarrow a \prec c$.*

□

An example of the partial order is the real numbers ordered by the standard less-than-or-equal relation $<=$.

In the standard transaction model, a consistent database state is implicitly defined by assuming that each transaction, when executed in isolation, maps a consistent database state to another consistent database state. The correctness in case of concurrent execution is defined in terms of serializability [19].

**Definition 6:** *A* **database** *consists of a finite set $D$ of data items. For each data item $d^{'} \in D$, there is a value $v^{'}$, which is in the the value domain $V(d^{'})$ of $d^{'}$. A* **database state** *consists of a data item $d^{'}$ and its value $v^{'}$. The set of a database state, denoted by $DS$, is a set of ordered pairs of data items $d^{'}$ and their values $v^{'}$,*

$$DS = \{(d^{'}, v^{'}) : d^{'} \in D \wedge v^{'} \in Dom(d^{'})\}.$$

*$DS$ has the property: if $(d^{'}, v_1^{'}) \in DS$ and $(d^{'}, v_2^{'}) \in DS$, then $v_1^{'} = v_2^{'}$. The restriction of $DS$ to data items $d \subseteq D$ is denoted by $DS^d$,*

$$DS^d = \{(d^{'}, v^{'}) : d^{'} \in d \wedge (d^{'}, v^{'}) \in DS\}.$$

□

A transaction is a sequence of operations resulting from the execution of a **transaction program**. A transaction program is usually written in a high-level programming language with assignments, loops, conditional statements and other complex control structures. Execution of a transaction program starting at different database states may result in different transactions. Formally:

**Definition 7:** *A* **transaction** *$T_i = (O_{T_i}, \prec_{T_i})$, where $O_{T_i} = \{o_1, o_2, .., o_n\}$ is a set of operations and $\prec_{T_i}$ is a total order on $O_{T_i}$. An operation $o_i$ is a 3-tuple*

$$(action(o_i), entity(o_i), value(o_i)),$$

*where $action(o_i)$ denotes an operation type, which is either a read (r) or a write (w) operation. $entity(o_i)$ is the data item on which the operation is performed. If the operation is a read operation, $value(o_i)$ is the value returned by the read operation for the data item read. For a write operation $value(o_i)$ is the value assigned to the data item by the write operation. For simplicity of the exposition, we assume that each transaction reads and writes a data item at most once.* □

**Definition 8:** *A* **schedule** *$S = (\tau_s, \prec_s)$ is a finite set $\tau_s$ of transactions, together with a total order, $\prec_s$, on all operations of the transactions such that for any two operations $o_1, o_2$ in $S$ and some transaction $T_i \in \tau_s$, if $o_1 \prec_{T_i} o_2$, then $o_1 \prec_s o_2$.* □

We use the notation $\{DS_1\}TP_i\{DS_2\}$ to denote the fact: when a transaction program $TP_i$ executes from another database state $DS_1$, it results in a database state $DS_2$. Similar notations will be used to denote execution of operations, transactions and schedules.

When a set of transactions execute concurrently, their operations may be interleaved. We model such an execution by a structure called a ***history*** [48]. A history indicates the order in which the operations of the transactions were executed relative to each other. Since some of these operations may be executed in parallel, a history is defined as a *partial order*. If transaction $T_i$ specifies the order of two of its operations, these two operations must appear in that order in any history that includes $T_i$. In addition, we require that a history specifies the order of all conflicting operations that appear in it. Two operations are said to *conflict* if they both operate on the same data item and at least one of them is a write.

**Definition 9:** *Let $T = \{T_1, T_2, ..., T_n\}$ be a set of transactions. A* **complete history H** *over $T$ is a partial order with the ordering relation $\preceq_H$, where*

1. *$H = \bigcup_{i=1}^{n} T_i$,*
2. *$\preceq_H \supseteq \bigcup_{i=1}^{n} \preceq_i$, and*
3. *for any two conflicting operations $p, q \in H$, either $p \preceq_H q$, or $q \preceq_H p$.*

□

Condition 1 says that the execution represented by $H$ involves precisely the operations submitted by $T_1, T_2, ..., T_n$. Condition 2 says that the execution honors all operation orderings specified within each transaction. Finally, the condition 3 says that the ordering of every pair of conflicting operations is determined by the ordering relation of the transactions. A *history* is simply a prefix of a complete history [19].

A complete history $H$ is *serial*, if for every two transactions $T_i$ and $T_j$ that appear in $H$, either all operations of $T_i$ appear before all operations of $T_j$ or vice versa. Thus, a serial history represents an execution in which there is no interleaving of the operations of different transactions. A history $H$ is *serializable* if its committed projection, $C(H)$, is equivalent to a serial history $H$. $C(H)$ is a complete history and it is not an arbitrarily chosen complete history. If $H$ represents the execution so far, it is really only the committed transactions whose execution the database management system has unconditionally guaranteed. All other transactions may be aborted [19].

We can determine whether a history is serializable by analyzing a graph derived from the history called a serialization graph. The *serialization graph* for $H$, denoted $SG(H)$, is a directed graph whose edges are all $T_i \rightarrow T_j (i \neq j)$ such that one of $T_i$'s operations precedes and conflicts with one of $T_j$'s operations in $H$. Therefore, a history $H$ is serializable if and only if $SG(H)$ is acyclic [19].

**Definition 10 (Weak consistency):** *The protocol is said to support weak consistency, if*

1. *all accesses to synchronization variables are seen by all processes (or nodes, processors) in the same*

*order (sequentially) - these are synchronization operations. Accesses to critical sections are seen sequentially,*

2. *all other accesses may be seen in different order on different processes (or nodes, processors), and*
3. *the set of both read and write operations in between different synchronization operations is the same in each process.*

□

Therefore, there can be no access to a synchronization variable if there are pending write operations. Furthermore, there cannot be any new read/write operations started if a system is performing any synchronization operation. More general, a weak consistency may be applied to any consistency models weaker than the sequential consistency [77]. The opposite of the weak consistency is the strong consistency, where parallel processes can observe only one consistent state [87, 12].

In the following definitions, we use $w_n[x] = v$ to denote that a write operation $w$ of the transaction $n$ writes a value $v$ to the data item $x$; $r_n[x] = v$ to denote that a read operation $r$ of the transaction $n$ reads from a data item $x$ the value $v$. We will omit written/read values $v$ and a concrete transaction $n$, when they are not necessary.

**Definition 11 (Linearizability [52]):** *A history H is linearizable iff there exists an execution $H^{'}$, where*

- *H and $H^{'}$ contain same operations,*
- *$H^{'}$ is sequential,*
- *$w[x] = v \prec_{H'} r[x] = v$, and*
- $\forall o_1, o_2 \in H if o_1 \prec_H o_2 \Rightarrow o_1 \prec_{H'} o_2$.

□

An example of the linearizable history is the history of operations on a stack data structure: $push_p^I(x), pop_q^I(x), pop_q^R(x)$, where processes $p$ and $q$ execute operations on a stack $x$; the push operation $push_p^I(x)$ stores a data item $I$ to the stack, and the pop operation $pop_q^R(x)$ takes the data item $R$ from the stack; the stack $x$ is initially empty. This sequence is linearizable because it can be extended with a matching response to $push_p^I(x)$, then linearized by following history: $push_p^I(x), push_p^R(x), pop_q^I(x), pop_q^R(x)$. However, deciding whether history is linearizable or not is very expensive [49, 51].

**Definition 12 (Legal):** *A read operation $r[x]$ belonging to an execution history H is legal, if $\forall w[x] : w[x] \prec_H r[x] \wedge \nexists w[x] : w[x] \prec_H w[x] \prec_H r[x]$.* □

**Definition 13 (Sequential consistency [57]):** *A history H is sequentially consistent if there exists a legal sequential history S equivalent to H. In other words, H admits a linear extension S in which all reads are legal.* □

Example 1 shows a sequentially consistent execution since there exists for all operations a total order. One such total order for this execution is $w_1(a); w_1(b); r_2(a); r_2(b)$.

Direction of time ---------------->
$P_1:$ $w_1(a)$ $w_1(b)$
$P_2:$ $r_2(a)$ $r_2(b)$

**Example 1: Sequentially consistent execution**

Example 2 shows an execution that is not sequentially consistent since there is no sequential order that satisfies the legal read definition.

Direction of time ---------------->
$P_1:$ $w_1(a)$ $w_1(b)$
$P_2:$ $r_2(b)$ $r_2(a)$

**Example 2: Not sequentially consistent execution**

The causal consistency, introduced by Ahamad et al in [6], defines a consistency criterion, which is weaker than the sequential consistency [55]. The causal consistency allows for a wait-free implementation of read and write operations in a distributed database. In the sequential consistency [33], all processes agree on a same legal history. The causal consistency defines a weaker agreement [53, 88]: Given a history H, it is not required that two processes agree on the same ordering for the write operations, which are not ordered in H. However, reads are required to be legal.

**Definition 14 (Causal consistency [78]):** *Let $H$ be a set of transactions, $\preceq_H$ be a partial order over set $H$, and relation $(H, \preceq_H)$ be a history. H is causally consistent if all its read operations are legal.* □

Direction of time ---------------->
$P_1:$ $w_1[x] = 1$
$P_2:$ $w_2[x] = 2$
$P_3:$ $r_3[x] = 2$
$P_4:$ $r_4[x] = 1$

**Example 3: Causal consistent execution**

In a causally consistent history, all processes see the same partial order of operations. An operation is potentially causal with: (1) the previous operations performed by the same thread of execution, 2) the operations that wrote the value this operation has read, and (3) the operations that are causally after the operation from rule (1) or (2) [65]. One weakness of the typical implementation of the causal consistency is that operations can only be ordered based on the actions observable by the system [21]. As an example, let us see Example 3.

In this example, only ordering restrictions are $w_2[x] = 2 \rightarrow r_3[x] = 2$. Writes $w_1[x] = 1 \parallel w_2[x] = 2$ are concurrent, hence they can be seen in different order by different processes. As a opposite example consider Example 4.

`Direction of time ---------------->`
$P_1$: $w_1[x] = 1$
$P_2$: $r_2[x] = 1$ $w_2[x] = 2$
$P_3$: $r_3[x] = 2$ $r_3[x] = 1$
$P_4$: $r_4[x] = 1$ $r_4[x] = 2$

**Example 4: Not causally consistent execution**

In Example 4, $w_2[x] = 2$ is causally-related on $r_2[x] = 1$, which is causally-related on $w_1[x] = 1$. Therefore, the system must enforce $w[x] = 1 \prec w[x] = 2$ ordering. Furthermore, $P_3$ violates that ordering because it reads value 1 after reading value 2. For more theoretical analysis see e.g. [27].

**Definition 15 (Eventual consistency [28]):** *An eventual consistency guarantees*

- *Eventual delivery: An update executed at a node are eventually executed at all nodes.*
- *Termination: All update executions terminate.*
- *Convergence: The nodes that have executed the same updates eventually reach equivalent state (and stay).*

□

As an example of the eventual consistency, consider Example 5. In this example, all updates are eventually executed, updates executions terminate, and eventually all processes reach equivalent states and all reads return the same value $x = 5$. During the execution, the processes may see updates on different orders, but eventually all reads return the same value. For more theoretical review see [28].

`Direction of time ---------------->`
$P_1$: $w_1[x] = 1$ $w_1[x] = 5$
$P_2$: $w_2[x] = 2$
$P_3$: $r_3[x] = 1$ $r_3[x] = 5$
$P_4$: $r_4[x] = 2$ $r_4[x] = 5$

**Example 5: Eventually consistent execution**

The eventual consistency is a weak consistency model, which is used in many large distributed databases, such as MongoDB [31], Riak [40] and S3 [24, 70]. Such databases require that all changes to a replicated piece of data eventually reach all affected replicas [13]. The conflict resolution is not handled in these databases, and the responsibility for solving conflicts is pushed up to the application authors in the event of conflicting updates.

The eventual consistency is a specific form of weak consistency: the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value [89, 90]. If no failures occur, the maximum size of the inconsistency window can be determined based on the factors such as communication delays, the load of the system, and the number of replicas involved in the replication scheme [2].

A few examples of eventually consistent systems are:

- Domain Name System (DNS)
- Asynchronous master-slave replication on an RDBMS, e.g. MariaDB (www.mariadb.org)
- Memcached in front of MariaDB, which caches reads.

The most popular system that implements the eventual consistency is DNS. Updates to a domain name are distributed according to a configured pattern and time-controlled caches. Eventually, all clients will see the same state. The eventual consistency means that given enough time, over which no changes are performed, all updates will propagate through the system and all replicas will be synchronized. However, at any given point of time, there is no guarantee that the data accessed is consistent, thus the conflicts have to be resolved.

## 3 Causal Consistency

The causal consistency model [54] ensures that the observed outcomes of operations are always consistent with the happened-before partial order as defined by Lamport [58].

**Definition 16:** *The happened-before relation, $\rightarrow$, can be formally defined as the least strict partial order on events,*

- *If events $a$ and $b$ occur on the same process, and the occurrence of event $a$ is before the occurrence of event $b$, then $a \rightarrow b$, and*
- *If an event $a$ is the sender of a message and an event $b$ is the receiptor of the message sent in event $a$, then $a \rightarrow b$.*

□

If there are other causal relationships between events in a given system, such as between the creation of a process and its first event, these relationships are also added to the definition [81].

Like all strict partial orders, the happened-before relation is transitive, irreflexive and antisymmetric, i.e.:

- $\forall a, b, c$, if $a \to b$ and $b \to c$, then $a \to c$ (transitivity).
- $\forall a, a \not\to a$ (irreflexivity).
- $\forall a, b$, if $a \to b \wedge b \to a$, then $a = b$ (antisymmetry).

Because the happened-before relation is both irreflexive and antisymmetric, it follows that: if $a \to b$ then $b \not\to a$.

The processes that make up a distributed system have no knowledge of the happened-before relation unless they use a logical clock, like a Lamport clock [58] or a vector clock [39, 67, 85].

Two events are concurrent if nothing can be said about the order in which they happened (i.e. events form a partial order). Events a, b, c are totally ordered if a happens before b and b happens before c. Same events are partially ordered if event a happens before b and c, and b and c are happening concurrently. For a data store to be causally consistent, the following is a requirement [56]: All writes that are potentially causally related must be seen in the same order by all processes. The model of causal consistency is a weakening of the sequential consistency: There exists an execution, which is causally consistent but not sequentially consistent; all executions satisfying sequential consistency are also causally consistent.

Let A and B be two operations that are performed by the same process. If A was executed before B, then A and B are causally related. There is a causal order between A and B, where A happens before B. Let A be a write operation and B be a read operation that read the value written by A. If A and B can be executed at different processes, then A happens before B in the causal order.

Causal relations are transitive: if A and B are causally related and B and C are causally related, then A and C are causally related. For example,if an operation A happens before an operation B and B happens before C, then A happens before C. Note that, with the sequential consistency, threads are not required to read the last written value as threads' read operations can be ordered in the past. Furthermore, contrary to the sequential consistency, concurrent writes (i.e., two writes that are not causally related) may be observed by different threads in different orders.

When the memory operations are causally related to each other, these operations are constrained. The memory operations, which are not causally related are called concurrent operations and they can be reordered with respect to each other. If the write operations are in the same order, the memory system is called causally consistent, while the concurrent write operations are observed in a different order. The causal consistency requires that potentially causally related writes are seen in the same order by all nodes in the system. The sequential consistency is stricter than the causal consistency: it requires that all operations are executed in a certain sequential order that is consistent with respect to the order observed by each process [9, 77].

## 4 Comparison to other Consistency Models

Strict consistency models are provably incompatible with the low-latency requirement [47]. Therefore, previous partition-tolerant, low-latency systems [36, 83, 42] use the weakest form of consistency, eventual consistency.

The eventual consistency provides the weakest consistency: when a replica is updated, all other replicas receive the updates and all replicas have same set of value after a period of time. The eventual consistency guarantees all sets of writes are propagated to other replica, but not in the same order. This makes programmers have to concern about the ordering issue because different orders may lead to inconsistencies to users. For example, Alice updates her profile then accepts the requests of a friend. This friend may see the profile before updated if only the eventual consistency is guranteed.

The causal consistency ensures a partial order between dependent operations; the sequential consistency ensures a global order among all operations for each key; the eventual consistency, a catch-all term used today, suggests the eventual convergence to some agreement [71].

The main difference from the eventual consistency is that the causal consistency allows non-blocking operations [84]: processes may complete read or write operations without waiting for global computation. Therefore, the causal consistency overcomes the biggest limitation of the strong consistency: delay in communication. Moreover, the semantic of several applications is precisely captured by the causal consistency, e.g. collaborative tools [44, 43]. Therefore, implementing a stricter consistency criterion not only induces unnecessary complexity of maintaining consistency, but also reduces the level of possible concurrency.

The eventual consistency does not provide any consistency guarantees during accesses. It has been observed that this does not necessarily cause serious problems for shared files [80, 61]. Often, the files that are heavily shared are not frequently updated by multiple users. Since the level of write-sharing is low, few consistency conflicts are generated. Unfortunately, this argument ignores read-dependency issues for shared files.

When read-dependencies are ignored, older versions of files may be read. As well as reading the latest version, a use can also access a older version. Since reads do not generate conflicts, the eventual consistency does not detect old reads, or provides any guarantees when the data is read. Therefore, a user may frist access the latest version, and then a older version. Under this case, the user will remember the old version and use it for future operations.

Let us use an example to illustrate the problem. A user edits a replicated file and checks-in it using the source code control system. The replica crashes before it can propagate its changes to other replicas. The user gets an old version of the file from another replica when the file is checked-out again. Often, users do not update this old copy of the file, and must remember that this copy of the file is old and should not update it. Moreover, if they do update the file, a conflict will necessarily happen. Worst of all, the user gets no hints that these problems might arise.

The eventual consistency ignores write-dependencies as well, thus writes can be made to older copies of data. However, often a single user may update multiple copies of data and thereby cause conflicts. This is different from the old read problem because writes may be done without reading the data.

Suppose a user is updating a replicated file on a replica A. Replica A becomes heavily loaded after the first update has reached it. The system switches to use the replica B, and writes are sent to the replica B. This causes conflicts since an older version of the file at the replica B gets updated. Both the problems of old reads and of old writes occur because the consistency guarantees are not provided when the access is done.

The existing systems [65, 60, 64] provide the causal consistency in order to achieve a better performance in a geo-distributed setting. In these systems, all operations from clients are served from the local data center, and are asynchronously pushed to remote data centers, but committed only after all causally dependent operations have been previously committed [59].

Hence, the requirements on clients can be captured by maintaining lists of transactions and propagating them among clients and servers [15]. However, due to the transitive nature of the dependency relationship, these lists can become very large resulting in significant memory, processing and network overheads [11].

To better describe the differences among the consistency models, lets take a online house auction as an example. We will use a very much simplified model, where two concurrent customers make an offer for the same house. Additionally, there is a maintainance transaction, which reads the current status of the auction and make the auction winner decision by selecting the current highest bidder and updating auction. Customer transactions read the current price for the house and increase the current offer by 100. The maintainance transaction reads the current price and selects the higher offer. Let $r_1(House\_price)$ indicate that the customer transaction 1 reads the house price, and $w_2(House\_price)$ indicate that the customer transaction 2 write the house price. At the beginning, $House\_price = 20000\$$. If the above transactions use serializability as consistency then we could have a history like in Example 6.

$r_1$(House_price)=20000,<br>
$w_1$(House_price +100)=20100, commit<br>
$r_2$(House_price)=20100,<br>
$r_2$(House_price)=20100<br>
$w_2$(House_price +100)=20200, commit<br>
$r_3$(House_price)=20200

**Example 6: Serializable execution**

Traditional databases use the strict Two-Phase Locking (2PL) to maintain the serializability, and the strict 2PL orders conflicting operations serially. Therefore, the both customers $T_1$ and $T_2$ see a consistent database and the correct auction winner is always selected. Now consider a case, where the causal consistency is used in same service. This could lead to a history in Example 7.

$r_1$(House_price)=20000,<br>
$r_3$(House_price)=20000,<br>
$r_2$(House_price)=20000,<br>
$w_1$(House_price +100)=20100, commit<br>
$w_2$(House_price +100)=20100, commit<br>
$r_3$(House_price)=20100

**Example 7: Causally consistent execution**

In this example, all writes are naturally causally related to the read operations and form a acyclic happened-before relation. Since writes are concurrent, they can be seen on different orders. This can naturally lead the lost update problem. Finally, lets consider a case where the eventual consistency is used in this service. This could lead to a history in Example 8 where we have added a third customer with transaction $T_4$.

$r_1$(House_price)=20000,<br>
$r_4$(House_price)=20000,<br>
$w_1$(House_price +100)=20100, commit<br>
$r_2$(House_price)=20100,<br>
$w_2$(House_price +100)=20200, commit<br>
$w_4$(House_price +100)=20100, commit<br>
$r_3$(House_price)=20100

**Example 8: Eventually consistent execution**

In the eventual consistency, writes do not need to be ordered and do not obey the happened-before relation, and this may lead to the lost update problem and to picking a wrong auction winner. In Table 2, we summarize the features of different consistency models based on writes. In Table 3, we compare the consistency, availability and performance features of these consistency models.

**Table 2: Write visibility of consistency models**

| Model | Write visibility |
|---|---|
| Serializability | Set of transactions can see all previous writes. |
| Eventual | Set of transactions can see only subset of writes. There is no guarantee which writes are visible and what is the final value seen. |
| Causal | See causally-related writes in the same order in all transactions. |

**Table 3: Consistency models: consistency, availability and performance**

| Model | Consistency | Avail. | Perf. |
|---|---|---|---|
| Serializability | Strong | Weak | Weak |
| Eventual | Weak | Good | Good |
| Causal | Average | Good | Average |

## 5 Implementation of causal consistency

Causal consistency can be reached by using Lamport clocks [58] or version vectors [67, 39]. The causal consistency model is implemented by using multi-part timestamps [18], which are assigned to each object. These timestamps are stored on a vector that contains the version number of the object at each replica. This vector must be included (in the form of dependencies) in all update and query requests so that operations respect the causal ordering: an operation A can only be processed at a given node if all operations, on which the operation A causally depends, have already been applied at that node [17].

Furthermore, the nodes must be aware of the versions present in other replicas. This information is maintained in a table that is updated by exchanging gossip messages between replicas. When a node is aware that a version has already reached all replicas then it issues an acknowledgment message so that the gossip dissemination protocol can be stopped.

The causal consistency guarantees provided by this approach allow its adaptation to a Geo-replicated scenario while maintaining scalability. Moreover, this technique can handle network partitions since the replicas can synchronize the updates when network recovers from the partition, without affecting the operations of system. However, this approach has a major drawback: large vector timestamps. These timestamps must be stored in the client and can be very large in a system with hundreds of nodes.

Causal consistency requires that all operations, which causally precede a given operation, must take effect before it. In other words, if $x \prec y$, then data item $x$ must be written before data item $y$. We call these preceding values dependencies. We use $write[x]$ to denote a write operation for a data item x. We say $y$ depends on $x$ if and only if $write[x] \prec write(y)$. These dependencies are the reverse of the causal ordering of writes, and by definition are the same as the happens-before relationship [58]. Example 9 illustrates the status updates on e.g. Facebook.

```
1: Alice: I have lost my phone!
2: Alice: I found my phone
   from bathroom.
3: Bob: That is good news!
```

**Example 9: Sequentially consistent execution**

In this example, Bob's read of Alice's posts 1 and 2 creates the causal link, which orders Bob's later comment after Alice's post and comment [63].

```
/* send the vector along with
  the message to all the sites.*/
$\forall\ i \in write\_set : v[i] := v[i] + i$

/* The message contains
(data item, new value) pair
for all date items
that are being written to.*/
```

**Listing 1: Sending write messages**

We have selected a method proposed by Ram et al [76] as an example of implementation. it is one simple and affective method, and is described using a pseudo-code in Listing 1 and Listing 2. Earlier, researchers have been using the vector clock or variations of it for maintaining the consistency. Each element in the vector clock corresponds to one host and the value of element indicates the number of messages sent from that host. The vector clock information is used to order or delay the delivery of messages if necessary, thus maintaining the required consistency. However, for maintaining the consistency of data items, we need information about writes on each data item, and maintaining a clock per data item can help. Therefore, instead of a vector clock of size N (number of hosts), we maintain a vector of size M (number of objects). The value of $v[i]$ in the vector $v$ contains the number of writes on data item $i$. This information can be used to maintain the consistency of each individual data item.

```
/* P is the vector received
with messages and v is
the local vector */
$\forall\ i\ \notin\ write\_set\ :\ v[i] := v[i] + i$
  /* This ensures that writes
  are delayed till the required
  causally preceding writes
  are received. */
  wait until $v[j]\ >=\ P[j]$
$\forall\ i\ \in\ write\_set\ :\ v[i] := v[i] + i$
  /* This ensures that causally
  overwritten messages
  are discarded */
  if $(v[i]\ >=\ P[i])$
    discard_the_message.
  else
    /* write updates */
    $v[i]\ := P[i]$
```

**Listing 2: Receiving write messages**

Each host maintains a vector of size M. Whenever it updates the value of an item, it increments the corresponding element and sends the vector along with the message of data item and new value to every site, which has a copy of the replica. When a host receives an update message, it delays the delivery of a message till each element in its vector is greater than or equal to the one that is piggybacked. After that, the updates to the data items are applied. In this case, the message overhead is $O(M)$ and thus is independent of the number of hosts in the system. This is particularly suitable in a mobile environment where the number of hosts is not fixed.

If each message is an update message, it carries the new value of the data item rather than instructions. Then the delivery of an update on a data-item does not need not wait for the previous updates on the same item. This would not have been possible if vector clocks had been used. In that case, the delivery of a massage would have been delayed even for previous messages that are causally overwritten.

## 6 Levels of Causal Consistency

Causality [58] or causal consistency ensures that a transaction is placed after all transactions that causally affect it. Causality has been used earlier in many non-transactional systems. In this section, we discuss the three levels of causality: no-causality, local causality and global causality. No-causality naturally means that operations do not respect a causal ordering, instead they are executed in e.g. an arrival order [20, 79].

Local causality allows an entity to access data that is consistent with its own operations, weakens the sequential consistency by only requiring the order of operations to respect the local order of operations for each process [7, 69]. The weakening is often exploited by a protocol guaranteeing sequential consistency on a smaller entity, for instance, a page containing objects [68].

Local consistency depends on the actions of the entity: if an entity has never accessed any versions of the data, then any versions satisfy the local causality. It is relatively simple and cheap to implement since the criterion can be checked and enforced by the entity itself. Entities do not have to coordinate among themselves to ensure that they access the data, which is consistent across entities.

Global causality requires that data is consistent with the actions of all other entities. Since the number of entities, which can access data, and the number of the data replicas, which these entities access, are arbitrary, this criterion requires examining all the replicas to see if any accesses have been made. In particular, it will only allow access to the latest data. A global causality is therefore not different from a strong consistency scheme, which provides the latest data on each access but does not provide high availability [30].

For example, there are two users at a public terminal computer. If the first user executes an inquiry and another user comes and executes a new inquiry, there is the no-causality between two inquiries and two users. Actually, the local causality can be useful in many cases. For example, suppose there are two database records: employee and manager. Each employee has one manager. If a manager adds an employee to the database and executes another transaction to inquire the employee's work, then the employee's name should be in the result. Because there are two transactions, the local causality is required. With the no-causality, another transaction inquiring the employee's work could get the empty set if the old database state is accessed.

The global causality is needed when the order of committed transactions is required. For example, in the stock application, a client updates the stock value of the company, then closes the company market. If another client notices that the company is closed, and wants to see the price, then the global causality ensures that the final price is seen. Table 4 summaries the the different levels of causal consistency.

For example, in the modern train an conductor could have mobile devices to sell/check train tickets. While a train is moving, the devices used by conductors can communicate with each other at the same train. How-

**Table 4: Causal consistency levels**

| Level | Order provided | Application |
|---|---|---|
| No-causality | no order | public terminal |
| Local | local order | employee DB |
| Global | global write order | stock market |

ever, there could be situations, where they can not communicate to the server at train stations. At stations, these mobile devices can synchronize their data with the sever of stations.

Just after a new passenger enters into the train, the train drives away and the passenger wants to buy a seat from a conductor. The conductor can sell the ticket without causing any conflicts, as other conductors' mobile devices in this train can see this state change. If other conductors also try to sell seats, they will naturally give different ones. Meanwhile, At a station, a customer wants to buy a seat in the moving train. Since the devices in the train cannot communicate with the server at the station, the station does not know how many new passengers just acquired a seat from the conductors in the train. If the station sells a seat to the customer, a conflict of seats may occur. If there is only one mobile device, the local causality is sufficient; if there is more than one mobile device, the global causality is needed. This example also has a global integrity constraint: free_seats_in_train is greater or equal to zero.

On every strain station, the databases are consistent and reading from the station server and from mobile devices is consistent, and thus reading seat status and reserving seats are causally related and consistent. When a train moves, reading and reserving seats from mobile devices are causally related/consistent. Note that we allow more customers than the seats of train. The only thing we must make sure is that one seat is reserved only to one customer, and this requires the global causal consistency.

When the station server is partitioned from the train, the station cannot safely sell seats of the train to customers, since there is the eventual consistency. Assume that there is only one free seat on the train and one passenger on the train reserves that seat. Concurrently on a station, a new customer also reserves that seat. When the train arrives at the station, There will have two customers on a same seat. To have the eventual consistency, one reservation needs to be rolled back.

The transaction of seat reservation contains a read operation getting free seats and a write operation reserving a seat, these are causally related. Both reservation transactions performed by the mobile device and the station server obey the local causality. However, two concurrent writes, reserving a seat, are are not causally related. In order to ensure that the seat is reserved correctly, the station server should not reserve seats when it cannot communicate with the mobile devices in the train. On train, mobile devices can safely sell free seats (because they know that the station server will not reserve seats.

Another example is the partition of network. A network is partitioned into two parts: the partition 1 has two nodes node1 and node2; the partition 2 has only one node node3. Thus, the partition 1 has a majority of nodes. In this example there may be different problems depending on which consistency model we used. If the traditional strong consistency is used, the system waits responses from all nodes. All nodes can respond only when the network partition is fixed.

If the eventual consistency is used, the system can still work if using the majority rule. This is because the system does know how many nodes it contains. The node executing the transaction also knows how many nodes, with which it can currently communicate. From this information the node knows whether these nodes form a majority in the system or not. If there is a majority, transactions are executed on all these nodes. If there is no majority, transactions are not allowed to execute. Once the network partition is fixed, the nodes, which are not in the part of majority, will then also execute missing transactions. Therefore, eventually all nodes reach a consistent state.

However, there are two problems with the eventual consistency: time is delayed before all nodes reach the same value; the eventual consistency does not restrict ordering of writes. For example in a social network, like Facebook, where Anna and Emy post comments to each other. In the eventual consistency Alice may see all comments but in a random order.

## 7 Applications and Databases using Causal Consistency

In the context of causal message ordering, Kshemkalyani et al. [56] propose an optimal algorithm for generalized causal message ordering. The proposed algorithm is optimal in the space complexity of the overhead of control information. The optimality is achieved by transmitting only minimum required information about causal dependencies, and using an encoding scheme to represent and transmit this information.

Birman et al. [22] propose causal message ordering primitive to ensure that if two event messages destined to the same process are causally related, they are delivered in the order they are supposed to happen. This could lead to a situation, where the method delays messages until the previous message is delivered.

Ahamad et al. [6] propose a method of causal memory abstraction. This method ensures that processes in a system agree on the ordering of the causally related operations. It is also non-blocking: processes can always execute a read or a write operations immediately.

Almeida et. al. [8] propose a geographically distributed key-value data store, ChainReaction. ChainReaction offers a causal+ consistency with high performance, fault-tolerance, and scalability. The causal+ consistency is stronger than the eventual consistency by leveraging a new variant of chain replication. In the causal+ consistency, replicas can temporarily diverge due to concurrent updates at different sites. However, they are guaranteed to eventually converge (a guarantee that is not provided by the causal consistency). On the other hand, in opposition to the eventual consistency, the causal+ consistency provides precise guarantees about the state observed by applications. However, the proposed system is not strictly causally consistent, thus we did not select this system for more detailed discussion.

## 7.1 COPS and Eiger

COPS system [65] (Clusters of Order-Preserving Servers) introduces the causal+ consistency and is designed to support complex online applications that are hosted in a small number of large-scale data-centers, each of which is composed of front-end servers (clients of COPS) and back-end key-value data stores. Eiger [64] has a similar design but a different implementation.

COPS [65] and Eiger [64] support causality through a client library. Both systems replicate writes to geographically distributed data centers and enforce observed ordering. The observed ordering is enforced by delaying the write operations until all causally previous operations have been already applied at that data center.

COPS executes all read and write operations in the local data center in a linearizable fashion, and then replicates data across data centers in a causal+ consistent order in the background. Figure 1 describes the high level architecture of COPS.

COPS also introduces a new type of operations named $get - transactions$. Get-transaction operations allow a client to read a set of keys and makes sure that the dependencies of all keys have been met before the values are returned. For example, there are two writes on objects $A$ and $B$ in a sequential order. It could happen that the write on $B$ is propagated faster among the replicas. If $A$ is read before and $B$, it could happen that we see the old value of $A$ and the new value of $B$, which is correct according to the causal ordering of operations but is not desirable in some applications. For example, in a travel agency, $A$ would be flight availability and $B$ hotel availability; in a net shop, both $A$ and $B$ could be the product availability. To solve this problem, we could use the get-transaction operation, because it guarantees that if we read the new version of $B$, we must also read the new version of $A$ (because the write on $A$ happens-before the write on $B$).

Similarly, the Eiger system provides the linearizability inside each data center and the causally-consistent data store based on a column-family data model to achieve better performance in a geo-distributed setting. All operations from clients are served from the local data center using a client library. The library mediates access to nodes in the local data center, executes the read and write transaction algorithms, and tracks causality and attaches dependencies to write operations. Each replica stores full replica of the database, and operations are handled locally. After an operation is executed locally, the operation is asynchronously pushed to remote data centers, but committed only after all causally dependent operations have been previously committed.

The causal consistency is provided in Eiger by checking whether an operation's nearest dependencies have been applied before applying the operation. This approach is similar to the method used by COPS [65]. The difference is that COPS places dependencies on values, while Eiger uses dependencies on operations.

Eiger represents dependencies by the data pair of a locator and a unique identifier. The locator is used to ensure that operations, which depend on the current executed operation $P$, can locate the node, in which the operation $P$ committed is. As an example, a write opertion $W2$ is executed on the node 1 and depends on an earlier write operation $W1$ executed on the node 2. The locator is used to find the node 2.

The unique identifier is used to map an operation to its dependencies, and is identical to the operation's timestamp. The dependencies in Eiger are checked by sending $dep\_check$ operations to the local data-center node that owns the locator. The local data-center checks local data structures in order to determine if the write operation identified by unique identifier is committed. If the write operation is committed, the local data-center responds immediately. If the write operation is not committed, the local data-center applies the write operation. Therefore, when all $dep\_checks$ return, the server knows that all causally dependent operations have been applied and it can safely apply this operation.

## 7.2 Bolt-on

Bailis et. al. in [14] propose a client-side middle-ware software called Bolt-on. This middle-ware guarantees only application-defined dependencies as an alternative to the causal consistency. Figure 2 describes the architecture of the Bolt-on middle-ware [14].

The Bolt-on architecture assumes that the underlying data store handles most aspects of data management, including replication, availability, and convergence. In the

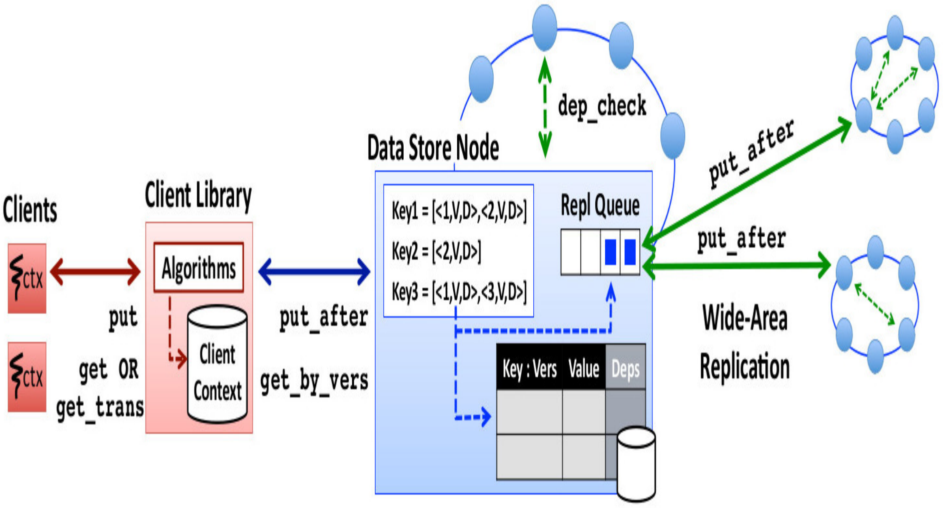

Figure 1: Architecture of COPS [65]

architecture, the underlying data store locally uses the eventual consistency and allows a large space of read and write histories; the middle-ware handles causal dependencies, and consists of a set of rules, which limit the possible histories to the histories that obey the desired consistency model.

Shim layer of Bolt-on is used to solve two problems. Firstly, Sim layer solves the inconsistency problems caused by the eventual consistency in PBS [16]. Secondly, this layer proposes a solution for concurrent overwrites in the causal consistency.

Bolt-on provides highly available read/write transactions, which separate replication, liveness and durability from consistency due to the upgrade from eventual consistency to causal consistency. The causal consistency of Bolt-on offers multi-key guarantees. However, Bolt-on can only capture dependencies explicitly specified by applications. Furthermore, the Bolt-on approach requires detailed knowledge on the conflict resolution strategy of the data store, and this makes it inapplicable for applications providing cloud storage services.

### 7.3 Application: MMORPG

In Massively Multi-player Online Role-Playing Games (MMORPG), players can cooperate with others in a virtual game world, and both players and different game words are naturally distributed. These systems manage large amounts of data, and the biggest problem is how to support data consistency. According to the CAP theorem, we have to sacrifice one of two properties: consistency or availability [1]. If an online game does not guarantee the availability, players' requests may fail. If data is inconsistent, players may get data not conforming to the game logic, and this data can affect their operations. Therefore, it is important for the MMORPG environment to find a balance between the data consistency and the system availability. For this reason, we must analyze the data consistency requirements of MMORPG so as to find the balance [29, 62, 73, 93].

Diao [37] has studied different consistency models for MMORPG and found that there indeed are part of data, where the causal consistency is an appealing choice: Game data. The game data contains e.g. the world appearance, the meta-data of non-player characters (the characters are created by game developers and controlled only by the game logic), the system configuration and game rules. This data is used by players and the game engine in the entire game, but can be only modified by the game developers. Consistency requirements for the game data are not so strict compared e.g. to the account data. Because e.g. a change of non-player character name or of the duration of bird animation may not be noticed by players.

Furthermore, some change of the game data needs to be delivered to all online players synchronously, e.g. a change of the word appearance, the weapon power, non-player characters, game rules and scripts. If there is inconsistency on these areas, it will cause errors on game display and logic errors on players. Therefore, some data needs to be stored on the server side and some on the client side. The game data on the client side could only synchronize with servers when a player logs in to or starts a game. For this reason, the causal consistency is required [37, 29].

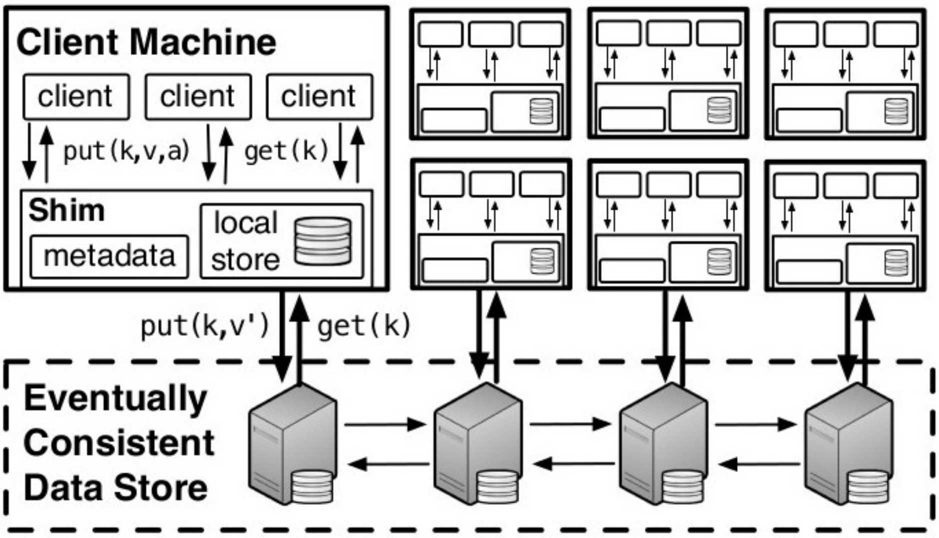

**Figure 2: Bolt-on architecture: a causally consistent shim layer mediates access to an underlying eventually consistent data store [14]**

This could mean that when a player A uses the browser to connect with the game server, the game server will check the current local data and update the game data of the client side in the form of data packets. After updating, all future local accesses will return the updated value. Player B, who has not communicated with the game server, will still retain the outdated game data.

Game servers maintain the primary version of game data, and transfer it to client sides. Additionally, players on different game words cannot discuss to each other. Thus, the only need is to make sure that the game data is consistent in one game word in a time so that all players on that game word are handled equally. This requires using the strong consistency locally in the game word and the causal consistency among different game words. When the game data is modified by developers, the update value should be delivered synchronously to all replicates on that game word, and asynchronously to other game words.

While the possibility of using the causal consistency on MMORPG has been identified on research [37, 29, 73], to the authors' knowledge there is no actual publications or other information that the causal consistency is actually used on MMORPG games.

### 7.4 Application: Facebook

When you log into your account on Facebook, the server will show your own status messages and your friends' status messages at that point in time. Status messages on Facebook may contain pictures, shared links and stories or your own messages. Naturally, your account data requires a strong consistency, but for status data the weaker consistency models are acceptable. During the time the user is online, the status updates of a user's friends and of the user do not need to be strictly ordered, and the causal ordering is enough. Thus when a user A sends a status update and a user B replies to that update, there is a causal order on the two updates. However, when users C and D do a totally unrelated update, the order these updates appear to users A and B is not relevant. This is because users A and B do not know in which order updates are performed.

The reason why the eventual consistency is not enough for Facebook status updates is that the eventual consistency does not require any ordering between writes. Consider a case, where the user A first sends a status update, and after few seconds A updates the first status update. With the eventual consistency, all friends of A could see only the first update, because the eventual consistency does not guarantee that first update is performed before the second one. In the causal consistency, as there is a read (by user A) of first update and then write (updated status from user A), these are causally related and all user A's friends will naturally see second update.

Although the causal consistency is the possible consistency model for Facebook status updates and several similar distributed services containing status updates like LinkedIn, Twitter and Yahoo, to author's knowledge there is not scientific or other literature that would show the causal consistency being really used.

## 8 Conclusions

In this paper we review the causal consistency and discuss how it differs from other consistency models, especially the eventual consistency. Additionally, we show how to implement the causal consistency and finally we identify the limitations of the causal consistency.

The causal consistency model can be enforced with Lamport clocks. Transactions using the causal consistency are executed in an order that reflects their causally-related read/write operations' order. Concurrent operations may be committed in different orders and their results can be read also in different orders.

Actually, the causal consistency can solve many problems, which cannot be solved in the eventual consistency, such as ordering operations. The causal consistency ensures that every sees operations in the same causal order, and this makes the causal consistency stronger than the eventual consistency. However, the causal consistency cannot support e.g. distributed integrity constraints.

Although there are a few promising systems developed on research (e.g. COPS [65], Eiger [64] and Boltson [14] ), to the authors' knowledge there is no commercial or mature systems using the causal consistency model.

## References

[1] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *IEEE Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[2] M. Abdallah and P. Pucheral, "A single-phase non-blocking atomic commitment protocol," in *9th International Conference on Database and Expert Systems Applications (DEXA)*, pp. 584–595, Vienna, Austria, August 24-28, 1998.

[3] V. Abramova, J. Bernardino, and P. Furtado, "Evaluating cassandra scalability with YCSB," in *25th International Conference on Database and Expert Systems Applications (DEXA)*, pp. 199–207, Munich, Germany, September 1-4, 2014.

[4] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[5] M. Ahamad and R. Kordale, "Scalable consistency protocols for distributed services," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 9, pp. 888–903, 1999.

[6] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: Definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[7] P. S. Almeida, C. Baquero, R. Gonçalves, N. M. Preguiça, and V. Fonte, "Scalable and accurate causality tracking for eventually consistent stores," in *14th International Conference on Distributed Applications and Interoperable Systems*, pp. 67–81, Berlin, Germany, June 3-5, 2014.

[8] S. Almeida, J. Leitão, and L. Rodrigues, "Chain-reaction: a causal+ consistent datastore based on chain replication," in *Eighth Eurosys Conference*, pp. 85–98, Prague, Czech Republic, April 14-17, 2013.

[9] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Trans. Comput. Syst.*, vol. 12, no. 2, pp. 91–122, 1994.

[10] B. Ayari, A. Khelil, and N. Suri, "Partac: A partition-tolerant atomic commit protocol for manets," in *Eleventh International Conference on Mobile Data Management (MDM)*, pp. 135–144, Kanas City, Missouri, USA, May 23-26, 2010.

[11] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "The potential dangers of causal consistency and an explicit solution," in *ACM Symposium on Cloud Computing*, p. 22, San Jose, CA, USA, October 14-17, 2012.

[12] ——, "HAT, not CAP: towards highly available transactions," in *14th Workshop on Hot Topics in Operating Systems*, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013.

[13] P. Bailis and A. Ghodsi, "Eventual consistency today: limitations, extensions, and beyond," *Commun. ACM*, vol. 56, no. 5, pp. 55–63, 2013.

[14] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 761–772, New York, NY, USA, June 22-27, 2013.

[15] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," *PVLDB*, vol. 5, no. 8, pp. 776–787, 2012.

[16] ——, "Quantifying eventual consistency with PBS," *VLDB J.*, vol. 23, no. 2, pp. 279–302, 2014.

[17] R. Baldoni, A. Milani, and S. T. Piergiovanni, "An optimal protocol for causally consistent distributed shared memory systems," in *18th International Parallel and Distributed Processing Symposium*, Santa Fe, New Mexico, USA, April 26-30, 2004.

[18] C. Benzaid and N. Badache, "Mobi_causal: a protocol for causal message ordering in mobile computing systems," *Mobile Computing and Communications Review*, vol. 9, no. 2, pp. 19–28, 2005.

[19] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[20] L. E. Bertossi and B. Salimi, "Causality in databases, database repairs, and consistency-based diagnosis (extended abstract)," in *Proceedings of the 8th Alberto Mendelzon Workshop on Foundations of Data Management*, Cartagena de Indias, Colombia, June 4-6, 2014.

[21] K. Birman, "A response to cheriton and skeen's criticism of causal and totally ordered communication," *SIGOPS Oper. Syst. Rev.*, vol. 28, no. 1, pp. 11–21, January 1994.

[22] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 47–76, 1987.

[23] D. Borthakur, "Petabyte scale databases and storage systems at facebook," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1267–1268, New York, NY, USA, June 22-27, 2013.

[24] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, "Building a database on s3," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, pp. 251–264, Vancouver, Canada, 2008.

[25] E. Brewer, "Cap twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, Februrary 2012.

[26] ——, "Towards robust distributed systems (abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, Portland, Oregon, USA, July 16-19, 2000.

[27] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak, "From session causality to causal consistency," in *12th Euromicro Workshop on Parallel, Distributed and Network-Based Processing*, pp. 152–158, Coruna, Spain, February 11-13, 2004.

[28] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv, "Eventually consistent transactions," in *21st European Symposium on Programming Languages and Systems*, pp. 67–86, Tallinn, Estonia, March 24 - April 1, 2012.

[29] A. Chandler and J. Finney, "On the effects of loose causal consistency in mobile multiplayer games," in *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games*, ser. NetGames '05, pp. 1–11, 2005.

[30] D. R. Cheriton and D. Skeen, "Understanding the limitations of causally and totally ordered communication," in *SOSP*, pp. 44–57, 1993.

[31] K. Chodorow and M. Dirolf, *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly, 2010.

[32] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: yahoo!'s hosted data serving platform," *PVLDB*, vol. 1, no. 2, pp. 1277–1288, 2008.

[33] T. Cornilleau and E. Gressier-Soudan, "A combined-consistency approach: Sequential&causal-consistency," *Operating Systems Review*, vol. 30, no. 4, pp. 33–44, 1996.

[34] S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in a partitioned network: A survey," *ACM Comput. Surv.*, vol. 17, no. 3, pp. 341–370, September 1985.

[35] J. Dean, "Designs, lessons and advice from building large distributed systems. keynote from ladis," 2009. [Online]. Available: https://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf

[36] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pp. 205–220, Stevenson, Washington, USA, October 14-17, 2007.

[37] Z. Diao, "Consistency models for cloud-based online games: the storage system's perspective," in *25th Workshop on Grundlagen von Datenbanken*, pp. 16–21, Ilmenau, Germany, May 28 - 31, 2013.

[38] M. M. Elbushra and J. Lindström, "Eventual consistent databases: State of the art," *Open Journal of Databases (OJDB), RonPub*, vol. 1, no. 1, pp. 26–41, 2014. [Online]. Available: http://www.ronpub.com/publications/OJDB-v1i1n03_Elbushra.pdf

[39] C. J. Fidge, "Timestamps in message passing systems that preserve the partial ordering," in *Theoretical Computer Science*, 1988.

[40] B. Fink, "Distributed computation on dynamo-style distributed storage: riak pipe," in *Proceedings of the Eleventh ACM SIGPLAN Erlang Workshop*, pp. 43–50, Copenhagen, Denmark, September 14, 2012.

[41] J. J. Florentin, "Consistency auditing of databases," *Comput. J.*, vol. 17, no. 1, pp. 52–58, 1974.

[42] A. Fox, S. D. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier, "Cluster-based scalable network services," in *SOSP*, pp. 78–91, 1997.

[43] R. Galli and Y. Luo, "Mu3d: a causal consistency protocol for a collaborative VRML editor," in *Web3D*, pp. 53–62, 2000.

[44] H. Garcia-Molina, "Using semantic knowledge for transaction processing in a distributed database," *ACM Trans. Database Syst.*, vol. 8, no. 2, pp. 186–213, June 1983.

[45] K. Gharachorloo, A. Gupta, and J. Hennessy, "Performance evaluation of memory consistency models for shared-memory multiprocessors," *SIGPLAN Not.*, vol. 26, no. 4, pp. 245–257, April 1991.

[46] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15–26, Seattle, WA, June, 1990.

[47] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, June 2002.

[48] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufman Publishers, 1993.

[49] R. Guerraoui and E. Ruppert, "Linearizability is not always a safety property," in *Second International Conference on Networked Systems*, pp. 57–69, Marrakech, Morocco, May 15-17, 2014.

[50] C. Hale, "You can't sacrifice partition tolerance," 2010. [Online]. Available: http://codahale.com/you-cant-sacrifice-partition-tolerance/

[51] J. Hamza, "Linearizability is expspace-complete," *CoRR*, vol. abs/1410.5000, 2014.

[52] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[53] F. Hupfeld, "Causal weak consistency replication: a systems approach," Ph.D. dissertation, Humboldt University of Berlin, 2009. [Online]. Available: http://edoc.hu-berlin.de/dissertationen/hupfeld-felix-2009-01-28/PDF/hupfeld.pdf

[54] P. W. Hutto and M. Ahamad, "Slow memory: Weakening consistency to enchance concurrency in distributed shared memories," in *10th International Conference on Distributed Computing Systems*, pp. 302–309, Paris, France, May 28 - June 1, 1990.

[55] E. Jiménez, A. Fernández, and V. Cholvi, "A parametrized algorithm that implements sequential, causal, and cache memory consistency," in *10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing*, pp. 437–444, Canary Islands, Spain, January 9-11, 2002.

[56] A. D. Kshemkalyani and M. Singhal, "Necessary and sufficient conditions on information for causal message ordering and their optimal implementation," *Distributed Computing*, vol. 11, no. 2, pp. 91–111, 1998.

[57] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, September 1979.

[58] ——, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[59] I. Lee, H. Y. Yeom, and T. Park, "A new approach for distributed main memory database systems: A causal commit protocol," *IEICE Transactions*, vol. 87-D, no. 1, pp. 196–204, 2004.

[60] J. H. Lee and K. L. Leung, "A stronger consistency for soft global constraints in weighted constraint satisfaction," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI)*, Atlanta, Georgia, USA, July 11-15, 2010.

[61] R. M. Lefever, M. Cukier, and W. H. Sanders, "An experimental evaluation of correlated network partitions in the coda distributed file system," in *22nd Symposium on Reliable Distributed Systems*, pp. 273–282, Florence, Italy, October 6-8, 2003.

[62] F. W. B. Li, L. W. F. Li, and R. W. H. Lau, "Supporting continuous consistency in multiplayer online games," in *Proceedings of the 12th ACM International Conference on Multimedia*, pp. 388–391, New York, NY, USA, October 10-16, 2004.

[63] R. J. Lipton and J. S. Sandberg, "Pram: A scalable shared memory," Princeton University, Tech. Rep. CS-TR-180-88, September 1988. [Online]. Available: https://www.cs.princeton.edu/research/techreps/TR-180-88

[64] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, pp. 313–328, Lombard, IL, USA, April 2-5, 2013.

[65] ——, “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*, pp. 401–416, Cascais, Portugal, October 23-26, 2011.

[66] P. Mahajan, L. Alvisi, and Dahlin, “Consistency, availability, convergence,” University of Texas, Tech. Rep. TR-11-2, May 2011. [Online]. Available: http://www.cs.utexas.edu/users/dahlin/papers/cac-tr.pdf

[67] F. Mattern, “Virtual time and global states of distributed systems,” in *Parallel and Distributed Algorithms*. North-Holland, pp. 215–226, 1988.

[68] A. Meliou, W. Gatterbauer, J. Y. Halpern, C. Koch, K. F. Moore, and D. Suciu, “Causality in databases,” *IEEE Data Eng. Bull.*, vol. 33, no. 3, pp. 59–67, 2010.

[69] A. Meliou, S. Roy, and D. Suciu, “Causality and explanations in databases,” *PVLDB*, vol. 7, no. 13, pp. 1715–1716, 2014.

[70] J. Murty, *Programming Amazon web services - S3, EC2, SQS, FPS, and SimpleDB: outsource your infrastructure*. O’Reilly, 2008.

[71] A. Naeem, A. Jantsch, and Z. Lu, “Architecture support and comparison of three memory consistency models in noc based systems,” in *15th Euromicro Conference on Digital System Design*, pp. 304–311, Cesme, Izmir, Turkey, September 5-8, 2012.

[72] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[73] W. Palant, C. Griwodz, and P. Halvorsen, “Consistency requirements in multiplayer online games,” in *Proceedings of the 5th Workshop on Network and System Support for Games*, p. 51, Singapore, October 30-31, 2006.

[74] D. Pritchett, “Base: An acid alternative,” *Queue*, vol. 6, no. 3, pp. 48–55, May 2008.

[75] M. R. Rahman, W. M. Golab, A. AuYoung, K. Keeton, and J. J. Wylie, “Toward a principled framework for benchmarking consistency,” in *Proceedings of the Eighth Workshop on Hot Topics in System Dependability*, Hollywood, CA, USA, October 7, 2012.

[76] D. J. Ram, M. U. Mahesh, N. S. K. C. Sekhar, and C. Babu, “Causal consistency in mobile environment,” *Operating Systems Review*, vol. 35, no. 1, pp. 34–40, 2001.

[77] M. Raynal, “Sequential consistency as lazy linearizability,” in *SPAA*, pp. 151–152, 2002.

[78] M. Raynal and A. Schiper, “From causal consistency to sequential consistency in shared memory systems,” in *Foundations of Software Technology and Theoretical Computer Science*, ser. Lecture Notes in Computer Science, vol. 1026, pp. 180-194, 1995.

[79] B. Salimi and L. E. Bertossi, “Causality in databases: The diagnosis and repair connections,” *CoRR*, vol. abs/1404.6857, 2014.

[80] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, “Coda: A highly available file system for a distributed workstation environment,” *IEEE Trans. Computers*, vol. 39, no. 4, pp. 447–459, 1990.

[81] R. Schwarz and F. Mattern, “Detecting causal relationships in distributed computations: In search of the holy grail,” *Distrib. Comput.*, vol. 7, no. 3, pp. 149–174, March 1994.

[82] W. Shi, W. Hu, and Z. Tang, “An interaction of coherence protocols and memory consistency models in dsm systems,” *Operating Systems Review*, vol. 31, no. 4, pp. 41–54, 1997.

[83] S. Sivasubramanian, “Amazon dynamodb: a seamlessly scalable non-relational database service,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 729–730, Scottsdale, AZ, USA, May 20-24, 2012.

[84] D. Skeen, “Nonblocking commit protocols,” in *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, pp. 133–142, Ann Arbor, Michigan, April 29 - May 1, 1981.

[85] J. S. Steinman, C. A. Lee, L. F. Wilson, and D. M. Nicol, “Global virtual time and distributed synchronization,” in *Proceedings of the Ninth Workshop on Parallel and Distributed Simulation*, pp. 139–148, Lake Placid, New York, USA, June 14-16, 1995.

[86] R. Tharakan, “Brewers cap theorem on distributed systems, scalable web architecture,” 2012. [Online]. Available: http://www.royans.net/wp/2010/02/14/brewers-cap-theorem-on-distributed-systems/

[87] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi, “Fast distributed transactions and strongly consistent replication for OLTP database systems,” *ACM Trans. Database Syst.*, vol. 39, no. 2, p. 11, 2014.

[88] F. J. Torres-Rojas and E. Meneses, "Convergence through a weak consistency model: Timed causal consistency," *CLEI Electron. J.*, vol. 8, no. 2, 2005.

[89] W. Vogels, "Eventually consistent," *Queue*, vol. 6, no. 6, pp. 14–19, October 2008.

[90] ——, "Eventually consistent," *ACM Communication*, vol. 52, no. 1, pp. 40–44, January 2009.

[91] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective," in *Fifth Biennial Conference on Innovative Data Systems Research*, pp. 134–143, silomar, CA, USA, January 9-12, 2011.

[92] H. Wang, J. Li, H. Zhang, and Y. Zhou, "Benchmarking replication and consistency strategies in cloud serving databases: Hbase and cassandra," in *Proceddings of 4th and 5th Workshops on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, pp. 71–82, Salt Lake City, USA, March 1, 2014, and Hangzhou, China, September 5, 2014.

[93] K. Zhang and B. Kemme, "Transaction models for massively multiplayer online games," in *30th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pp. 31-40, October 2011.

## AUTHOR BIOGRAPHIES

**M.Sc. Mawahib Elbushra** received her MSc on Computer Science from the College of Graduate Studies, Sudan University of Science & Technology. She is currently aiming PhD on Computer Science. Her research interests include cloud databases, distributed databases and eventual consistency.

**Dr. Jan Lindström** is the principal engineer at SkySQL working on InnoDB storage engine and Galera cluster. Before joining SkySQL he was software developer for IBM DB2 and development manager for IBM solidDB core development. He joined IBM with the acquisition of Solid Information Technology in 2008. Before joining Solid in 2006, Jan worked on Innobase and spent almost 10 years working in the database field as a researcher, developer, author, and educator. He has developed experimental database systems, and has authored, or co-authored, a number of research papers. His research interests include real-time databases, in-memory databases, distributed databases, transaction processing and concurrency control. Jan has an MSc. and Ph.D. in Computer Science from the University of Helsinki, Finland.