**REAL-TIME EVENTUAL CONSISTENCY**

Jan Lindström¹ and Mawahib Musa Elbushra²

¹SkySQL – The MariaDB Company, Espoo, Finland
²Sudan University of Science & Technology, College of Graduate Studies, Khartoum, Sudan

**ABSTRACT**

Many real-time systems are naturally distributed and these distributed systems require not only high-availability but also timely execution of transactions. Consequently, eventual consistency, a weaker type of strong consistency is an attractive choice for a consistency level. Unfortunately, standard eventual consistency does not contain any real-time considerations. In this paper we have extended eventual consistency with real-time constraints and this we call real-time eventual consistency. Followed by this new definition we have proposed a method that follows this new definition. We present a new algorithm using revision diagrams and fork-join data in a real-time distributed environment and we show that the proposed method solves the problem.

**KEYWORDS**

Real-time databases, eventual consistency, transaction processing.

**1. INTRODUCTION**

A real-time database system (RTDBS) is a database system intended to handle workloads whose state is constantly changing [9, 27]. Traditional databases contain persistent data, mostly unaffected by time in contrast. As an example consider application keeping track of all of the markets in a stock market that changes very rapidly and is dynamic. A real-time database is a database where transactions have deadlines or timing constraints [30]. Real-time databases are usually used in real-time applications which require timely access to data. Different applications have different timeliness requirements; from microseconds to minutes [33].

Many real-world applications contain tasks that require time-constrained access to data as well as access to data item that has temporal validity [26]. As an example: a telephone switching system, network management, navigation systems, stock trading, and command and control systems. As an example following task require time-constrained access: looking up the 800 directory, obstacle detection and avoidance, radar tracking and recognition of objects. Tasks in these real-time applications collect data from the dynamic environment, process data gathered earlier, and produce timely and correct response. Additionally, these real-time tasks process both temporal data, which loses its validity after a specific time intervals, as well as historical data.

Traditional databases, from now on referred to as databases, manipulate persistent data. Transactions process this data and preserve its consistency. The goal of transaction and query processing methods chosen in databases is to optimize throughput or response time. Real-time database systems in contrary can also process temporal data, i.e., data that becomes outdated after a certain time. Task in real-time systems have timing constraint because application environment contains data that is outdated in certain time periods. Therefore, real-time tasks use periods or deadlines. These requirement leads to a new goal where real-time systems has to meet the time constraints of the tasks.
Term real-time does not just mean fast [33] and real-time does not mean timing constraints that are in nanoseconds or microseconds. Real-time imply the need to manage explicit time constraints in a predictable fashion using time-cognizant methods to deal with time or periodicity constraints associated with tasks. Databases are necessary in real-time applications because they include many useful features that allow (1) the description of data, (2) the maintenance of correctness and integrity of the data, (3) efficient access to the data, and (4) the correct executions of query and transaction execution in spite of concurrency and failures [29].

Many complex real-time systems require distribution and sharing of extensive amounts of data, with full or partial replication of database. Consider for example integrated vehicle control system were autonomous nodes control individual sub-systems and handle lot of data locally under hard timing constraints (e.g., fuel injection, ignition control). These systems also require parameters from other subsystems on a less critical time scale (e.g. transmission, environment sensors). Similarly, in automated manufacturing, stock trading and, massively multiplayer online role-playing games (MMORPG). In these application scenarios, temporary inconsistencies of the distributed database can be allowed. Therefore, updates are made locally and propagated to all notes in a way that guarantees eventual consistency [37, 38]. To achieve high scalability at low cost, distributed databases are typically highly distributed systems running on commodity hardware. Scale out requires adding a new off the shelf server.

However, Brewer [3,7] introduced CAP theorem proposing that it is impossible for a distributed computer system to concurrently provide all three of the following 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 lost messages or partial failure of the system.

This theorem is also known as Brewer's theorem [7, 3] and it was later formally shown by Gilbert and Lynch [16].

This arouses the concern on which property is the least important one, since all three properties are both important and anticipated in distributed systems [3, 8]. Many applications are often used together with databases where the so called ACID (Atomicity, Consistency, Isolation, and Durability) properties apply and thus strong consistency is required. Databases use transactions to provide strong consistency.

High availability is expected because systems are connected to a network and if the network is available, the system is also supposed to be available [18]. Furthermore, tolerance to network partitions is also expected because it is a basic fault-tolerance technique. Distributed system should continue service even if a network link goes down or a server crashes [16].

Network partitioning can be totally avoided or make very unlikely using a several servers on the same rack. This solution is not useful for cloud system because it does not scale and decreases the tolerance against other failure types. Use of more reliable links between the networks does not eliminate the chance of partitioning, in contrary increases the cost significantly. Thus, network partitioning will happen and system designer has to select either consistency or availability.

Consistency requirement is similar to the atomicity property in ACID where every single transaction will be atomic in strictly consistent database. Different server can have different views of the same data item if the database is not strongly consistent.

Distributed system should be designed to “allow arbitrarily loss of messages sent from one server to another to avoid partition tolerance [16]. Storing all data at one server is not an option because
of strict availability requirement. This is because when the only source fails the entire system becomes unavailable. Thus, partition tolerance enables for system states to be kept in different locations. Distributed database that is partition tolerant allows to perform read/write operations while partitioned. If distributed database is not partition tolerant, when partitioned, the database may become completely unusable or only available even for read operations.

Therefore, maintaining a single-system service in a distributed system has a cost as CAP theorem shows [3]. Updates cannot be synchronously propagated to all servers without blocking when processes within a distributed system are partitioned. This blocking presents unavailability to some or all of its users. High availability system can offer low latency even when network is partitioned, because it can respond to a user’s request without contacting other servers [3]. Allowing partitioning is not an option as noted on [18], thus choice is either consistency or availability.

Brewer [8] noted that two of the three requirements can be guaranteed simultaneously if one of the requirements can be traded-off. It could be possible to have every kind of combination, CP, AP and CA. However, every system wants to be tolerant to partitions. Therefore, only open question in CAP is what do we do when some servers are not available?! Choosing CA would be a system that does not tolerate network partitioning and making system unavailable. CP or AP are only choices in the distributed databases. Therefore, most of the NOSQL-databases provide only eventual consistency, a weaker type of strong consistency.

Eventual consistency states that in an updatable replicated database, eventually all copies of each data item converge to the same value. The origin of eventual consistency can be traced back to Thomas’ majority consensus algorithm [36]. Eventual consistency was also used by Terry et al. [35] and later made more popular by Amazon in their Dynamo system, which supported only eventual consistency [13].

Eventual consistency is a weaker form a consistency compared to strong consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value [5]. When failures do not happen, the maximum size of the inconsistency window can be calculated based on communication delays, the load on the system, and the number of servers participating in the replication. The most popular system that uses eventual consistency is DNS (Domain Name System). Update to a name is distributed based on order and time intervals configured; eventually, all client applications will see the latest name [37]. When an applications fetches a data item there is no guarantee that the data accessed is consistent, therefore the conflicts have to be resolved in application [37].

In this paper our major motivation is to use eventual consistency as a consistency level in a real-time distributed relational database system. We make following contributions in this paper:

- To authors knowledge this is the first paper were real-time eventual consistency is defined and used on real-time database systems.
- While lacy replication is used on DeeDS and this lacy replication is sometimes called as eventual consistent it has not been shown that DeeDS can maintain eventual consistency on all configurations and all failure cases. Furthermore, traditional eventual consistency does not provide any real-time constraints.
- Proposed method allows multiple-masters and requires that transactions is successfully executed on all servers before transaction deadline.
- We will rigorously show that proposed real-time eventual consistency method produces correct results.
The rest of this paper is organized as follows. We first review the related research in Section 2. This is followed by definition of real-time eventual consistency in Section 3. A new real-time eventually consistent method is presented in Section 4. Finally, conclusions are presented in Section 5.

2. RELATED WORK

Most of the previous work on real-time databases has been based on simulation. However, several prototypes of real-time databases have been introduced. Real-time disk-based transaction processing system RT-CARAT [20], was one of the firsts [20]. Similarly, REACH (Real-time Active and Heterogeneous mediator system project) [10] and the STRIP (Stanford Real-time Information Processor) project [1].

Kim and Son [22] have presented StarBase real-time database architecture. StarBase architecture has been developed over a real-time microkernel operating system and it is based on relational model. Wolfe & al. [39] have implemented a prototype of object-oriented real-time database architecture RTSORAC. RTSORAC architecture is based on open object-oriented database architecture with real-time extensions. Database is implemented over a thread-based POSIX-compliant operating system. Similarly, DeeDS project at the University of Skövde [2, 15, 17] and the BeeHive project at the University of Virginia [34] are examples of more recent RTDBS prototype projects.

M2RTSS main-memory object oriented database system architecture is presented by Cha & al. [12]. M2RTSS is object-oriented storage manager implementing real-time transaction scheduling, and recovery. Real-Time Object-Oriented Database Architecture for Intelligent Networks (RODAIN) [23], is a real-time, object-oriented, and fault-tolerant database management system. RODAIN is a main-memory real-time database using priority and criticality based scheduling and optimistic concurrency control.

Concurrently, commercial “real-time” database system products have started to appear in the market such as Eaglespeed-RTDB [14], Clustra [21], Timesten [24], and SolidDB [4, 28]. While these products are not real-time databases based on strict definition of real-time since most of them only have very limited real-time features, they represent a very important step forward to the success of real-time database systems. In these commercial database management systems many use in-memory database techniques to achieve a better real-time performance.

Distributed real-time database systems have been also studied on literature e.g. [19, 20, 25, 27, 32, 34]. To authors knowedge DeeDS is the only distributed real-time database system mentioning eventual consistency. Therefore, we review DeeDS more detailed way.

The DeeDS [2, 15, 17] database prototype stores the fully replicated database in main memory to make transaction timeliness independent of disk accesses. Because, DeeDS uses full replication transactions can be executed on the local node entirely, independent of network delays. Updates are send to other servers using detached replication independently from the execution of the transaction. Clients using the database sees the distributed database as a single local database and does need not to handle data location or how to synchronize concurrent updates of different replicas of the same data. Full replication uses a lot of resources, since all updates needs to be replicated to all the servers. Additionally, this causes scalability problem for resource usage, such as bandwidth for replication of updates, storage for data item replicas, and processing for replicating updates and resolving conflicts for concurrent updates at different servers for the same data item. Currently, real-time databases do not scale well but in same time the need for larger distributed real-time databases is increasing.
The DeeDS architecture supports multi-master replication. The replication scheme adopted in DeeDS (referred to as lazy replication) is used in order to make real-time database access predictable [2]. That is, by replicating all data onto every node and accepting updates on every node, an application will never have to access data over the network. Instead, it can always access and change data locally. Further, since eventual consistency is employed, updates can be made on different nodes simultaneously without any need to lock the entire database, only the local replica. When an update has committed locally, it is propagated to the other nodes. Thus, an application can always apply any changes necessary locally, without waiting for the remote replicas. This implies that the database may become temporarily globally inconsistent, and that conflicts may occur. Any conflict that occurs must be detected and resolved. Local consistency is enforced by pessimistic locking of the local replica of the database.

Replication in DeeDS is handled by the replication module. The replication module interacts with the storage manager TDBM, and with the DeeDS Operating system Interface (DOI). TDBM is a store manager with support for nested transactions [6]. The DOI is a layer added between the DeeDS database and the underlying hardware which makes it possible to make the DeeDS platform independent [11]. DeeDS replication module and architecture is shown in the Figure 1.

The replication module logs all updates on data made during a local transaction. When the local transaction has been committed, the log is propagated to all other DeeDS nodes. The changes are then integrated into the remote replica(s). This is currently implemented in DeeDS by using a number of modules: the logger, the propagator, the integrator, the version vector handler (VV handler) and the log filter (depicted in above figure). The logger is responsible for logging any changes made during a local transaction. When a transaction has been committed, the log is forwarded to the propagator module. The propagator distributes the update message containing the log(s) to all other replicas of the database. Currently, the log is distributed over TCP via point to point connections.

A more efficient solution would be to use broadcast or multicast. On the remote nodes, the update message is received by the integrator module. The integrator module consists of several sub modules, these are: the receiver, the conflict detector, the conflict resolver and the updater. The receiver sub module is responsible for receiving update messages sent by propagators on other DeeDS nodes. The conflict detector then checks if the update made is in conflict with any earlier updates on this local node. This is done by using version vectors2 and the log filter. Three different types of conflicts may occur: write-write conflicts, read-write conflicts, and read-write cycles [2].

![Figure 1. Replication in DeeDS [3].](image-url)
If a global conflict is detected, the conflict resolver has to resolve the conflict. To achieve predictability, DeeDS detects and resolves conflicts locally, without communication between replicas over the network. If a conflict is detected, it is vital that it is resolved within a predictable amount of time in order to achieve predictable real-time database access, and in a deterministic way for global consistency. That is, if two or more replicas are in conflict, all replicas must resolve the conflict in the same way. DeeDS support a set of simple generic replication policies.

The chosen value may be:

- based on the new value resulting from an operation, e.g., mean min or max value of two conflicting updates
- chosen from the highest prioritized replica
- based on timestamps (e.g., use the oldest or newest value)

If no conflict has been detected or when any detected conflict has been resolved, the updater writes the changes to the local copy of the database, and to the log filter. This must be done as one atomic action. During the integration process, the involved database objects and the log filter must be locked so that no other process manipulates them when the conflict detector is working. If the local database and the log filter are not kept mutually consistent, conflicts may be undetected with potential system failure as the result. Integration is done by using a special kind of transactions, integration transactions which run on a lower priority than regular database transactions. This makes sure that local access to the database does not need to wait an unpredictable amount of time for any integration processing. Thus, local real-time guarantees are kept.

These design choices makes real-time database accesses in DeeDS predictable since all accesses are made in local main memory. Further, the database is guaranteed to be globally consistent eventually since all changes are propagated to other nodes, where conflicts are detected and resolved in a predictable and deterministic way.

3. Real-Time Eventual Consistency

In this section we first define necessary terminology.

**Definition 1:** Real-time database data item can be denoted by \( d: (\text{value}, \text{read timestamp}, \text{write timestamp}, \text{absolute validity interval}) \), where \( d\{\text{value}\} \) denotes the current state of the \( d \), read timestamp denotes when the last committed transaction has read the state of the \( d \), write timestamp denotes when the last committed transaction has written the \( d \), i.e., when the observation relating to a \( d \) was made, and absolute validity interval, denotes the length of the time interval following read timestamp during which \( d \) is considered to have absolute validity.

A set of data items used to derive a new data item form a relative consistency set \( R \). Each such set \( R \) is associated with a relative validity interval. Assume that \( d \in R \). Data item \( d \) has a correct state if and only [30]:

- \( d\{\text{value}\} \) is logically consistent, i.e., satisfies all integrity constraints.
- \( d \) is temporally consistent:
  - Data item \( d \in R \) is absolutely consistent iff \( \{\text{current time}\} - d\{\text{observation time}\} < d\{\text{absolute validity interval}\} \).
  - Data items are relatively consistent iff \( \{d' \in R \mid d\{\text{timestamp}\} - d'\{\text{timestamp}\} < R\{\text{relative validity interval}\}\} \).
DEFINITION 2: Real-Time Eventual consistency.

- **Real-time ordering**: If update conflicts, a higher priority transaction wins and lower priority transaction is aborted.
- **Real-Time Eventual delivery**: An update executed at node before its deadline evenly executes at all nodes.
- **Real-Time Termination**: All update executions terminate before their deadlines or they are aborted.
- **Convergence**: Nodes that have executed the same updates eventually reach equivalent state (and stay).

**EXAMPLE 1**: Consider a case where data item R=0 on all three nodes. Assume that we have following sequence of writes and commits: W(R=3) C W(R=5) C W(R=7) C in node 0. Now read on node 1 could return R=3 and read from node 2 could return R=5. This is eventually consistent as long as eventually read from all nodes return the same value. Note that that this value could be R=3. Eventual consistency does not restrict the order which the writes must be executed.

4. PROPOSED METHOD

Proposed real-time eventually consistent distributed commit protocol is now presented. Presentation of the proposed real-time eventually consistent method start from presenting how real-time transactions are executed on master (or primary) server and then how changes are replicated to other servers in the system.

Read only real-time transactions can be executed on any of the servers reachable by the client. Because database state does not change on read only real-time transaction, there is no effect on database consistency. Thus, global transaction identifier is not assigned to the real-only transactions. Instead, only a local real-time transaction identifier is used. Because, full replication is used in this work, single database server can execute the transaction.

Write transactions change database state and they are executed on master first and when master has committed the real-time transaction, we replicate it to other servers for execution. Any node that can receive messages from client can be used as master but normally the server replying fastest is used as master.

Locally, data consistency is ensured by using a multiversion model (Multiversion Concurrency Control, MVCC). For each transaction a snapshot of committed data is created when accessed. This snapshot of data is consistent and transaction can’t view inconsistent data produced by other concurrent transaction updates on the same data rows, providing transaction isolation for each database session. MVCC provides less lock contention in order to allow for reasonable performance in multiuser environments. Furthermore, standard write-a-head logging is used to guarantee durability.

```plaintext
1  GTID := Source_id: + next_transaction_id;
2  If GTID is stored in master’s log {  
3  End; /* already executed */
4  }
5  While every statement Sk in the transaction T {  
6  If Sk is contains at least one write statement {  
7  If # servers reachable from master < (# total servers / 2) + 1 { /* Majority*/
8  Abort transaction T;
9  }
```
10  }
11  fork {
12  revision r = { tuple }
13  If Sk is read {
14      Execute operation;
15  } Else If Sk is Write {
16      call conflict_detection(T, T');
17      If revision = current version {
18         Execute transaction;
19      } Else {
20         request current version
21      }
22  If database inconsistent {
23      Abort;
24  }
25  }; /* Sk is write */
26  }; /* while executes statements on master*/
27  Join r;}
28  If (deadline < current_time) abort;
29  Write transaction operations and GTID to the log;
30  Send transaction redo log and GTID to other server for execution;

Figure 1. Master server algorithm.

Real-time transaction is executed firstly on the master server, the master server assigns real-time transaction a global transaction identifier GTID. Since global transaction identifier contains the server’s UUID, it remains constant for transactions executed on that server. Global transaction identifier is represented as a pair where GTID = source_id:transaction_id, and transaction_id is globally unique. Possible method to implement globally unique transaction identifiers is to use Lamport’s clocks.

Global transaction identifiers are persistently stored to the ARIES style redo log using a new log event called Gtid_log. Gtid_log record is written before a group of operations for a given transaction is to be written to the log. Storing global transaction identifiers in log of the master server makes sure that no real-time transaction is re-executed more than once and two different real-time transactions cannot have the same global transaction identifier.

When master receives a transaction for execution we first check that global transaction identifier is unique in a log file, which means global transaction identifier is not found from the log. If global transaction identifier is found, we have already executed the real-time transaction and we can skip this real-time transaction.

Transactions containing INSERT, UPDATE or DELETE statements are write transactions and master server must be able to send messages and receive replies from majority of the servers. When real-time transaction is a read operation the transaction will find a snapshot of the data using a fork operation that allows creating a copy of the data item and read it. For real-time transactions containing INSERT, UPDATE or DELETE statements, the proposed method finds a snapshot of data item using fork. Proposed method validates that updated data item is not old snapshot or version of data item by equation: current version equal to new version minus one. Then we check integrity constraints to guarantee that the new update not violate integrity constraints.
After all changes has been done, we check that real-time constraint is not violated using transaction deadline and current time.

Finally we use join all operation to merge these updates with main version and then global transaction identifier is written to the log (immediately preceding the real-time transaction itself in the log). Send transaction log and global transaction identifier to other servers, this means other server can’t see this new data item version until committed in the master server.

Now we consider how real-time transactions are executed on a server that is not a master. Either, real-time transaction is read-only transaction or transaction is already committed on a master server and now replicated to this server for execution. For read only real-time transactions we only need to provide current consistent version of the data item. There is no need to use any special method for this. Additional processing is required for write real-time transactions received from master and this processing is presented below.

We can execute read-only real-time transactions on any of the servers, but transactions containing write statements (update, insert or delete SQL-statements) must to be first executed on a master server. Thus, servers are not autonomous i.e. master makes the decision whether the transaction commits or aborts. If transaction is aborted on a non-master server normal REDO-log records of the transaction are requested from the master server and that server executes ARIES style recovery for that transaction.

Every server removes transaction from queue after it checks global transaction identifier is not already executed on this server. If it is, we can skip the transaction. This is followed by validation that this transaction based on its global transaction identifier is next transaction based on already executed transactions in the log file. If not, this server sends a REDO log request to master server and master server sends REDO logs for all missing transactions. Finally server writes transaction operations and global transaction identifier to the log.

When executing write real-time transactions we need to check that data item versions are consistent. They must represent consistent view. Data item versions must represent consistent view and integrity constraints should not be violated after transaction execution, if they do we send REDO log request to the master server. Master server will provide REDO logs for all data items in a read and write set of the transaction. These REDO logs are then executed on this server using traditional ARIES style recovery. After transaction commit database is consistent (eventually) database state and the transaction log records and global transaction identifier can be written to the log file followed by log record for transaction commit.

```plaintext
1   Remove transaction T from Queue;
2   If gtid received can be found from log file{
3       End; /* Skip */
4   }
5   If transaction_id > transaction_id +1 on log {
6       Put transaction T back to Queue; /* Ordered execution */
7   };
8   For every statement Sk in transaction T {
9       Check data items versions touched on Sk are consistent;
10      Execute the statement;
11      If database inconsistent {
12          Abort transaction;
13          Send REDO log request to Master;
14      End;
```
Finally conflict detection is done as follows:

1. If $t, t' \in T_H$ and $t$ conflicts with $t'$ then
2. If $t_{\text{priority}} \geq t'_{\text{priority}}$ then
3. $t \lt_v t'$
4. Else $t' \lt_v t$.
5. EndIf
6. Endif

Figure 3. Conflict detection.

We have to first define few precise definitions in order to show that proposed method produces correct consistency level and we do this similarly as in [5]. Firstly, all operations must be part of a transaction and transactions can be described using following three types of operations (query-update interface):

1. Updates $u \in U$ issued by the transactions
2. Pairs $(q, p)$ form a query $q \in Q$ that is executed by the transaction together with a result $v \in V$ from the database system.
3. Abort or Commit issued by the transaction.

Formally, we can represent the activity as a stream of operations forming a history.

**DEFINITION 3:** A *history* $H$ for a set $T$ transactions and query-update interface $(Q, V, U)$ is a map $H$ which maps each transaction $t \in T$ to a finite or infinite sequence $H(t)$ operation from alphabet $\Sigma = U \cup (Q \times V) \cup \{\text{end}\}$.

Next we define the order where operations are executed on a transaction i.e. program order.

**DEFINITION 4:** *Program order*. For a given history $H$, there is a partial order $\prec_p$ over operations in the $H$ such that $e \prec_p e'$ iff $e$ appear before $e'$ in some sequence of the $H(t)$.

Then we define an equivalence relation.

**DEFINITION 5:** *Factoring*: We define an equivalence relation $\sim_t$ over events such that $e \sim_t e'$ iff trans $(e) = \text{trans} (e')$. For any partial order $\prec$ over operations, we say that $\prec$ factors over $\sim_t$ iff for any operations $x$ and $y$ from different transactions $x \prec y$ implies $x' \prec y'$ for any $x, y$ such that $x \sim_t x'$ and $y \sim_t y'$. Thus, factoring forms a partial order on the transactions.

With following formalization we can specify the information about relationships between events declaratively, without referring to implementation-level concepts, such as replicas or messages. Namely, $F$ takes as a parameter not a sequence, but an operation context, which encapsulates “all we need to know” about a system execution to determine the return value of a given operation.

Eventual consistency weakens strict consistency models by allowing queries in a transaction $t$ to see only a subset of all transactions that are globally ordered before $t$. This is done by separating a visibility order (a partial order that defines what updates are visible to a query), and an arbitration order (a partial order that determines the relative order of updates).
**DEFINITION 6:** A *history* $H$ is real-time eventually consistent if there exist two partial orders $\prec_v$ (the visibility order) and $\prec_a$ (the arbitration order) over events in $H$, such that the following conditions are satisfied for all events $e_1, e_2, e \in EH$:

1. *Arbitration extends visibility:* if $e_1 \prec_v e_2$ then $e_1 \prec_a e_2$.
2. *Total order on past events:* if $e_1 \prec_v e$ and $e_2 \prec_v e$, then either $e_1 \prec_a e_2$ or $e_2 \prec_a e_1$.
3. *Compatible with program order:* if $e_1 \prec_p e_2$ then $e_1 \prec_v e_2$.
4. *Consistent query results:* for all $(q, v) \in EH$, $v = q^\# \left(\{e \in H \parallel e \prec_v q\}, \prec_a, s_0\right)$.
   Thus query returns the state as it results from applying all preceding visible updates (as determined by the visibility order) to the initial state, in the order given by the arbitration order.
5. *Atomicity:* Both $\prec_v$ and $\prec_a$ factor over $\sim_t$.
6. *Isolation:* If $e_1 \notin$ committed (EH) and $e_1 \prec_v e_2$, then $e_1 \prec_p e_2$. That is, events in uncommitted transactions are visible only to later events by the same client.
7. *Real-Time Eventual delivery:* For all committed transactions $t$, there exists only finitely many transactions $t' \in TH$ such that $t \prec_v t'$ and $t_{\text{deadline}} \leq \text{commit}\_time$.
8. *Real-Time Ordering:* If $t, t' \in TH$ and $t$ conflicts with $t'$ then $t \prec_v t'$ if $tpriority > tpriority$ otherwise $t' \prec_v t$. Thus, real-time ordering forms a partial ordering of transactions.

The reason why eventual consistency can tolerate temporary network partitions is that the arbitration order can be constructed incrementally, i.e. may remain only partially determined for some time after a transaction commits. Thus, conflicting updates can be committed even in the presence of network partitions.

**THEOREM 1:** A history $H$ produced by the proposed method is real-time eventually consistent.

**PROOF:**

- *Compatible with program order:* All transaction operations are executed on all servers in the order they appear on program in step 9. Thus, proposed real-time eventually consistent method is compatible with program order.
- *Arbitration extends visibility:* In step 2 we make sure that slaves execute transactions in the same order as they are executed on master. Local concurrency control uses MVCC that produces locally serializable histories, thus we know that revision diagram is acyclic and a partial order. Because data items versions use revision diagram fork and join operations (see [6]) $\prec_v$ is acyclic, transitive and partial order. Thus we know that if $e_1 \prec e$ and $e_2 \prec e$, then either $e_1 \prec e_2$ or $e_2 \prec e_1$. Now define $\prec_v = \prec_a = \prec$, then if $e_1 \prec_v e_2$ then $e_1 \prec_a e_2$.
- *Consistent query results:* In step 2 query can continue if and only if all preceding transactions are executed. Result of the query contains state as it results from applying all preceding visible updates.
- *Atomicity:* Step 2 of the proposed real-time method produce total order i.e. if $e_1 \prec e$ and $e_2 \prec e$, thus either $e_1 \prec e_2$ or $e_2 \prec e_1$. Now define $\prec_v = \prec_a = \prec$, then we can easily see that both $\prec_v$ and $\prec_a$ factor over $\sim_t$.
- *Isolation:* MVCC that is used as local concurrency control method (steps 20–24) produces partial ordering (local serializability) and MVCC also maintains program order, thus if $e_1 \in$ committed (EH) and $e_1 \prec_v e_2$, then $e_1 \prec_p e_2$. Thus, updates produced by uncommitted transactions are visible only to later events by the same client.
- *Real-Time Eventual Delivery:* For all executed transactions $t$ at step 28 we check that transaction is executed before its deadline, thus $t_{\text{deadline}} \leq \text{commit}\_time$. When server crashes or there is network failure between servers, the server is unavailable. After server is restored or server can again send and receive messages from rest of server’s, we
requests log records of the missing transactions. ARIES method is then used to process received log records and thus eventually all operations are delivered.

- When server is up it can serve only read-only transactions if the server does not belong to majority. If there is no network partitioning, write transactions are executed only after the missing transactions are executed (step 2). Thus, there can exist only finitely many transactions \( t' \in \mathbf{T} \) such that \( t \prec_{v} t' \) for every transaction \( t \).

- **Real-Time Ordering:** In conflict detection step 16 and Figure 3 we make sure that \( t, t' \in \mathbf{T} \) and \( t \) conflicts with \( t' \) then \( t \prec_{v} t' \) if \( t_{\text{priority}} \geq t'_{\text{priority}} \) otherwise \( t' \prec_{v} t \).

5. CONCLUSIONS

Many real-time systems are naturally distributed and these distributed systems require not only high-availability but also timely execution of transactions. CAP summarizes trade-offs from decades of distributed-system designs and shows that maintaining a single-system image in a distributed system has a cost. Therefore, distributed databases can either be strongly consistent or available. Consequently, eventual consistency, a weaker type of strong consistency is an attractive choice for a consistency level. Unfortunately, standard eventual consistency, does not contain any real-time considerations. In this paper we have extended eventual consistency with real-time constraints and this we call real-time eventual consistency. Followed by this new definition we have proposed a method that follows this new definition.

As a future research we will build a prototype real-time database system using real-time eventual consistency. Another research question is how to allow more flexibility and additional rules on both conflict resolution and recovery of transactions. Furthermore, the question what is the strongest consistency level that can be provided to transactions and still avoid problems with CAP theorem on bounded error scenario in network.

REFERENCES


Authors

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 a MSc. and Ph.D. in Computer Science from the University of Helsinki, Finland.

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 and distributed databases.