Timeline Index on RamCloud
Master Thesis

By Filip Ćurčić

Supervised by
Prof. Donald Kossmann
Martin Kaufmann

October 2012 – April 2013
Abstract

One of the most recent problems of modern storage systems lies in its scalability. Today’s storage systems have recently started steadily going towards NoSQL, new generation of simple and scalable storage systems. Ramcloud is a RAM based key-value store that provides scalable and efficient access to data.

At the same time, maintaining temporal data has become very important in commercial applications. However, algorithms used for operations on temporal data often exhibit poor performance. Timeline index addresses these issues, as a unified data structure that supports temporal operations such as temporal aggregation, time-travel and temporal join.

This thesis presents a solution on how to efficiently store temporal data and run temporal operators on Ramcloud key-value store.

First, we show that there is a scalable and efficient solution to storing and querying temporal data on RamCloud. In the next step, we implemented such system and provide benchmarking results of this system. Lastly, we explore what could be potential improvements of this system.
1. Introduction

1.1. Motivation
With the expansion of the internet and cloud computing, today’s database systems are reaching limits of their performance. Big companies, like Facebook for example, put a lot of effort into optimizing their data storage technologies. It should be noted that Facebook in fact stores temporal data, displayed to the users in the form of Facebook Timeline. Temporal data is of increasing importance for data mining and business analysis, because past trends are very useful when making predictions. Accessing temporal data can be quite inefficient and most of the existing data structures do not support all the temporal operators at once. We can conclude that motivational background of this work stems from two major topics. First one is about storing data on a scalable system, and the second one matters management of temporal data in such system. This is the motivation for implementing the Timeline index on a RamCloud system.

The Timeline Index is a data structure for processing queries on temporal data. It supports 3 types of queries: temporal aggregation, time-travel and temporal join.

RamCloud is DRAM based key-value store that provides strong consistency guarantees and atomic operations.

1.2. Problem statement
Most of the currently used temporal database systems use hard drive as their storage, and store temporal data rather inefficiently, or they have been developed for a single type temporal query. This makes these systems unscalable and functionally limited.

Our solution substitutes traditional database system with a modern key-value store – RamCloud. We use RamCloud not only to store temporal data, but the Timeline Index data structure as well.

1.3. Thesis structure
In Section 2, we give a brief description of RamCloud and we compare it to the traditional database systems. In Section 3, we describe temporal data management and Timeline Index and compare it to the more common non-temporal databases. Section 4 describes architecture and implementation of the system, and the following section presents benchmark setup and results. In the last section, Section 6, we discuss challenges, possible improvements and future work.
2. RamCloud overview

This section describes one of the 2 integral parts of the project – RamCloud. First, we will give a brief motivation behind the RamCloud project. Afterwards, we will give overview of RamCloud architecture. Finally, we will present most important of its features (for this project) and present RamCloud API.

2.1. Motivation

The first problem with traditional databases is latency. Standard traditional single machine applications had average about 1µs latency, which was acceptable. However, with development of world wide web came development of large scale web applications where average network latency is about 0.5-10ms, meaning that it has degraded up to 10 000 times.

Another problem with traditional databases is that they do not scale. It is especially visible in the example of web applications, like aforementioned Facebook, which consists of 4000 MySQL instances and 2000 memcached servers [2].

Third problem lies in the fact that disk access rate is not keeping up with disk capacity. Moore’s law states that number of transistors on a chip doubles every 18 months, while disc access rate does not offer as close as much room for improvement [3].

RamCloud answers problems of latency, scalability and disk technology in today’s applications. In its core, RamCloud is a key-value store that provides high performance and low latency by using RAM instead of disks as storage media. Aim is to enable a new breed of information-intensive applications. Until now, RamCloud is able to serve 1000-10 000 clients with 5-10µs latency, which is around 1000 better than latency on traditional databases [2].

2.2. Architecture

RamCloud supports running up to 10 000 storage servers, each storage server consisting of master and a backup. Master manages data in a DRAM and responds to client requests. Backup keeps redundant copies of data from other masters.

Next to storage servers, an integral part of the system is a specialized machine called coordinator. It is used to manage configuration of the cluster and directs recoveries when one of the storage servers fail. When application starts up, it gets information about the tables from coordinator. Afterwards all the communication is done directly between application and server, unless something changes in the structure, in which case application goes back to coordinator.

Thus RamCloud can serve 10 000 application servers and each of them is equipped with RamCloud library [2].

Overview of RamCloud architecture is given in the Figure 1. Abbreviation AS stands for application servers.
2.3. Features

In RamCloud, data is organized in tables. Table entries are key-value pairs. Keys are 64bit unsigned integers, and values are basically just byte arrays. User can create tables, read, write and modify entries, and finally, delete tables.

Every table has its name and ID. To manage table data, client has to provide table ID. It is possible to read and write multiple entries at once, which has an advantage over single reads and writes because it sends just 1 RPC request per accessed server.

Table 1 shows RamCloud basic API with features that were most important for the project.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void createTable(const char* name)</code></td>
<td>Creates a new table with specified name, or throws an exception if table exists</td>
</tr>
<tr>
<td><code>void dropTable(const char* name)</code></td>
<td>Drops an existing table specified with its name, and deletes all of its entries</td>
</tr>
<tr>
<td><code>void read(uint64_t tableId, const void* key, uint16_t keyLength, Buffer* buffer)</code></td>
<td>Reads a value for a specified key from the table specified with tableId into the buffer</td>
</tr>
<tr>
<td><code>void remove(uint64_t tableId, const void* key, uint16_t keyLength)</code></td>
<td>Removes a value for a specified key from the table with id tableId</td>
</tr>
<tr>
<td><code>void write(uint64_t tableId, const void* key, uint16_t keyLength, void* val)</code></td>
<td>Writes the value given with val, into the table given with tableId to the entry specified with key</td>
</tr>
<tr>
<td><code>void multiRead(MultiReadObject* requests[], uint32_t numRequests)</code></td>
<td>Reads a specified number of objects into the array of objects</td>
</tr>
<tr>
<td><code>void multiWrite(MultiWriteObject* requests[], uint32_t numRequests)</code></td>
<td>Writes an array of objects into the table, numRequest number of objects, speci</td>
</tr>
</tbody>
</table>

| **Table 1: RamCloud basic API** |
3. **Timeline Index overview**
Timeline Index is data structure that is used to efficiently manage temporal data and to perform queries on it. First, we will give brief motivation behind Timeline Index project. Afterwards, we will show features of Timeline Index.

3.1. **Motivation**
Temporal databases are databases with built-in time aspects, more specifically, *application time* and *system time* [4]. In this paper we focus only on system time. What that means is that multiple versions of the same object are stored. There is no usual “update-in-place”, but every time new version of object is created. Thus it is possible to compare current data with the data from the past, querying specific versions from the past and extracting data.

Temporal data management is becoming more important with development of web. Number of use cases of this kind of applications is steadily growing, however supporting data structures are mostly either inefficient, or they are specialized for a certain query that is essential for the system.

Timeline Index is independent of the physical order of data, so it provides flexibility in physical design. Timeline Index shows very predictable performance which is better than the results gained by other approaches. It provides three types of queries: temporal aggregation, temporal join and time travel.

3.2. **Features**
Timeline Index keeps track of all the inserts, updates and deletes in the database. Temporal databases require keeping track of validity of entries, in regards to versions. Therefore, apart from data that is kept in the database, database keeps information such as *from* and *to* for each row, referring to the system time of a tuple. There are two types of entries – ones with open *to* field, which means that they are valid for the current version and ones with *to* field set, i.e. the ones that are outdated.

3.2.1. **Basic operations – insert, update, delete**
Basic operations on temporal databases with timeline index will be explained on the following example:

1.  **INSERT** (111, John, 100.00) **INTO** Client(Key, Name, Balance), **COMMIT**
2.  **INSERT** (222, David, 200.00) **INTO** Client(Key, Name, Balance)
    **INSERT** (333, Ann, 150.00) **INTO** Client(Key, Name, Balance), **COMMIT**
3.  **UPDATE** Client **SET** Balance=0.00 **WHERE** Key=222, **COMMIT**
4.  **DELETE** FROM Client **WHERE** Key=333, **COMMIT**

**Insert.** An example of temporal table after step 2 is given in Table 2a, and corresponding Timeline Index structure is shown in Table 2b.
In Table 2a we can see 3 inserted rows, which represent account balances of bank clients. First row inserted at version 1, and 2 rows inserted at version 2. Timeline index in Table 2b shows that row with \( ROW\ ID = 1 \) was inserted at version 1, and rows with \( ROW\ ID = \{2,3\} \) were inserted at version 2. So as we can see, inserting into temporal database, as with common databases, requires creating a new entry with inserted data in temporal table.

**Update.** To show how updates work, we will observe Table 3a and Table 3b as temporal table and Timeline Index, respectively, after step 3.

As temporal table at version 3 shows, David has withdrawn all of his money from the account. But instead of just editing data in row with \( ROW\ ID=1 \), we set its **to** field to 3. We also add another row to the table, with currently valid data. Timeline Index is also updated so that it shows that in version 3, row with \( ROW\ ID=2 \) has been invalidated, and row with \( ROW\ ID=4 \) has become valid.

We come to conclusion that updates in temporal databases are in fact 2 operations – setting **to** field of currently valid data to the current version and inserting new entry with **from** field set to the current version.

**Delete.** Deleting an entry from table is illustrated in Table 4a, while corresponding operations on Timeline Index are shown in Table 4b. Look of these tables corresponds to the end of step 4.
Table 4a: History Table

<table>
<thead>
<tr>
<th>ROW ID</th>
<th>Data</th>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>111 John</td>
<td>100.00</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>222 David</td>
<td>200.00</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>333 Ann</td>
<td>150.00</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>222 David</td>
<td>0.00</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4b: Timeline Index

<table>
<thead>
<tr>
<th>Version</th>
<th>ROW ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1+</td>
</tr>
<tr>
<td>2</td>
<td>2+, 3+</td>
</tr>
<tr>
<td>3</td>
<td>2-, 4+</td>
</tr>
<tr>
<td>4</td>
<td>3-</td>
</tr>
</tbody>
</table>

Temporal Table shows that at version 4, Ann’s account has been closed. To field of row with ROW ID=3 is set to current version. Timeline Index is updated with entry for version 4, which suggests that row with ROW ID=3 has been invalidated.

3.2.2. Temporal Aggregation

Temporal aggregation is in fact aggregation of data over versions, temporal grouping [5]. Temporal aggregation computes (groups) the aggregates as running values for time points or time intervals. Timeline Index supports three types of temporal aggregations – cumulative, selective and general. Cumulative examples of temporal aggregation are Temporal Sum and Temporal Mean. Examples of selective aggregates are Temporal Minimum, Maximum and Temporal Median.

Temporal Sum

Temporal Sum and other cumulative aggregation algorithms are simplest algorithms for Timeline Index to solve. Aggregate value for the current version is being calculated directly by knowing the aggregate value of previous version and the change in current version. In the matter of space consumption, no additional intermediate structures are needed. In the matter of time performance, complexity of SUM algorithm is O(n), where n is number of rows in temporal table. Temporal Sum’s implementation on RamCloud is discussed in more detail in Section 4.2.3.

Temporal Maximum

Temporal maximum and minimum are noncumulative algorithms, which makes them a bit more complex than cumulative ones. Timeline Index uses Top-K algorithm inspired by Skyline computation, which keeps track of Top K extreme values during computation. It requires additional memory space for Top-K algorithm and exhibits time complexity of O(n log n) in a real-world scenario. Implementation of Temporal Maximum on RamCloud is explained in Section 4.2.3.

3.3.4. Time Travel

Time Travel concerns establishing a consistent view of the data in a specific version [6]. It is most commonly used temporal query in commercial systems. Time Travel implementation on RamCloud is explained in more detail in Section 4.2.4.
3.2.3. Temporal Join

In difference to common join operation, temporal join contains predicates on both key and time domains. Thus it creates big challenges on the matter of selecting priority dimension of those two, which leads to difficult tradeoffs [7]. Timeline index selects temporal aspect as the more important. Temporal join on RamCloud will be explained in more detail in Section 4.2.5.
4. Timeline Index on RamCloud

In the last two sections, we have introduced RamCloud and Timeline Index. In this section we will show main topic of this thesis, and that is how Timeline Index has been designed and implemented on RamCloud system. First we will show architecture of the system, and afterwards we will discuss implementation details.

4.1. Architecture

An overview of the system architecture can be seen in Figure 2. Client is a program that runs operations on Timeline Index, which is stored in RamCloud. Server is typical RamCloud master server described in section 2.2. Necessary part of the system is also a coordinator which is described in the same section.

![Figure 2: Timeline Index on RamCloud](image)

From Figure 2 we can see that regular RamCloud architecture with coordinator and multiple masters is used in the system. Apart from client’s data that is stored on master servers, Timeline Index data structure is also stored on the server. Timeline Index Query Engine (TIQE) has role of managing data and executing queries, using data stored in Timeline Index structure.

4.2. Implementation

First we will explain our solution for storing temporal data in RamCloud. Afterwards, we will give overview of algorithms for managing data and query execution.
4.2.1. Storage

**Storing client’s data**

For every table of client’s data, two RamCloud tables are maintained – *current table* and *history table*. This is the same approach that is used in SAP’s HANA temporal database and IBM DB2 [8][9]. Current table holds currently valid data, while history table stores both current data and data that used to be valid in the past. This redundancy is useful for the following 3 reasons:

1) It provides quick access to currently valid data.
2) Snapshot Isolation and concurrency
3) At the moment of inserting/updating/deleting we cannot easily obtain information about key presence only from history table. We cannot do it solely based on primary key of the tuple that is being inserted, because history table holds unique keys, which are combination of entry’s primary key and version at which the entry is inserted in history table. On the other hand, current table’s keys are in fact just primary keys of the entries, so it is possible to access current table for information about key presence. We need this information in order to change existing rows in history table when updating and deleting, for example.

Since RamCloud is key value store, data is kept in such way that whole tuple is being inserted as a value into key value store. Keys for current table are just primary keys of tuples, while keys for history table are in form *key-version*, where *key* is primary key of the tuple, and *version* is version at which they are inserted.

For every temporal table, there is exactly one Timeline Index structure stored in RamCloud.

**Storing Timeline Index**

Timeline Index on RamCloud consists of Activation Index, Invalidation Index and Checkpoints. Activation and Invalidation Index are stored together in 1 RamCloud table – Event List table. Event List is updated on every commit operation. When new entry in Event List is created key of the entry is version number, while value contains the following information:

\[
\{A, \{AKey1, AKey2... AKeyN\}\}, \{D, \{DKey1, DKey2... DKeyM\}\}
\]

Where A is number of activated tuples, keys AKey1…AKeyN are keys that have been activated at that version, while D stands for number of deactivated tuples, and keys DKey1…DKeyM are keys that have been deactivated at that version. Event List therefore consists of Activation Index (first parenthesis) and Deactivation Index (in second parenthesis).
Even though we encode the differences between versions in Event List, establishing a view on a specific version is still a demanding operation. For that reason, apart from storing Event List in one table, another table is used for storing checkpoint versions. Checkpoint versions are especially useful for time travel operation. Number of checkpoint versions in our work is usually between 10 and 20. It is obvious that checkpoint structure requires certain cost of maintaining, both in time and space aspects.

Example of checkpoint table in RamCloud is given in Table 5.

<table>
<thead>
<tr>
<th>Checkpoints Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version</strong></td>
</tr>
<tr>
<td>2000000</td>
</tr>
<tr>
<td>4000000</td>
</tr>
<tr>
<td>6000000</td>
</tr>
</tbody>
</table>

Table 5: Checkpoints Table

Example in Table 5 shows 3 checkpoint versions, with checkpoint difference of 2000000. Checkpoint contains a list of active tuples’ keys from history table.

4.2.2. Basic operations - Insert, update, delete

Basic operations insert, update and delete will be explained on the same course of operations as in section 2.2.

---

1. **Step 1.** INSERT (111, John, 100.00) INTO Client(Key, Name, Balance), COMMIT
2. **Step 2.** INSERT(222, David, 200.00) INTO Client(Key, Name, Balance)
   INSERT(333, Ann, 150.00) INTO Client(Key, Name, Balance), COMMIT
3. **Step 3.** UPDATE Client SET Balance=0.00 WHERE Key=222, COMMIT
4. **Step 4.** DELETE FROM Client WHERE Key=333, COMMIT

---

Insert

Inserting into temporal database in fact requires the following RamCloud basic operations:

- WRITE {Key, Data} INTO Current Table
- WRITE {Key-Version, Data} INTO History Table

Let us take the same example as for nontemporal databases – deposits on John’s, Ann’s and David’s accounts. Tables 5a and 5b represent actual data in history table and Timeline Index, respectively, after Step 2.
From the history table we can see that data is now stored as a blob in a key-value store. Current table is not shown, but it contains exactly the same entries as history table, except the difference that its keys are simple primary keys of actual tuples. Timeline Index stores information that in version 1, 1 blob with key 111-1 has been added to the database, and that at version 2, 2 blobs with keys 222-2 and 333-2 have been added to the database.

**Update**

Updating temporal database requires the following course of basic RamCloud operations:

- READ (Key, DataOld) FROM Current Table
- WRITE (Key-Version, Data) INTO History Table
- WRITE (Key, Data) INTO Current Table

First step is necessary in order to find out version at which the blob with the same key was previously added. That version is needed for updating Timeline Index structure. Second and third step are just updating history table and current table.

Table 6a shows the look of the history table after updating entry with key 222. Table 6b shows the Timeline Index after committing version 3.
From history table, we can see that account balance for the entry with key 222 has been changed to 0. Timeline Index is also updated accordingly. Upon commit operation new entry is added that shows that in version 3, key 222-2 has been deactivated, and key 222-3 became active. Thus every update is in fact represented as combination of single activation and deactivation.

Delete

Deleting from temporal database is implemented using the following course of RamCloud basic operations:

- READ (Key, DataOld) FROM Current Table
- DELETE (Key) FROM Current Table

Once again, like when updating, reading old data from current table is necessary in order to find out the version at which the blob with desired key has previously been added. Second step concerns deleting data from the current table.

After deleting entry with key 333, history table stays the same as shown in Table 6a. Table 7a shows the final look of the current table, while table 7b shows the Timeline Index.

### Table 7a: History Table

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>{111, John, 100.00}</td>
</tr>
<tr>
<td>222</td>
<td>{222, David, 0.00}</td>
</tr>
</tbody>
</table>

### Table 7b: History Table

<table>
<thead>
<tr>
<th>Version</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1, [111-1]}, {/]</td>
</tr>
<tr>
<td>2</td>
<td>{2, [222-2, 333-2]}, {/]</td>
</tr>
<tr>
<td>3</td>
<td>{1, [222-3]}, {1, [222-2]}</td>
</tr>
<tr>
<td>4</td>
<td>{/]</td>
</tr>
</tbody>
</table>

We can see that the current table holds only 2 entries and that the entry with key 333 has been removed, as current table functions the same way as nontemporal databases. An entry has been added to the Timeline Index that shows that at version 4 there was 1 deactivation of key 333-2.

### 4.2.3. Temporal Aggregation

As previously stated, we make distinction on three aggregates – cumulative, selective and general. Examples of cumulative aggregation are SUM, COUNT, AVERAGE, while best examples of selective temporal aggregation are MIN and MAX.
Cumulative aggregation

We will take SUM as an example of cumulative aggregation. We will show how timeline index on RamCloud is used to solve the following temporal aggregation query:

```
SELECT SUM(Balance) AS TOTAL
FROM Client cl
GROUP BY cl.Version_ID();
```

This query returns a vector of sums of all balances for every version in the system.

We will perform aggregation by traversing the Timeline Index. RamCloud does not support traversing entries in sequential order, so traversing is done by calculating the sum for every single version in the system from 1 to current version, regardless if it made a change to the table that matters for aggregation or not. However, RamCloud does provide a way to read multiple entries in parallel with `multiRead` functionality.

To explain the algorithm, we will make a stop at version 2. The procedure for calculating aggregated value for version 2 is illustrated in Figure 3.

As we can see, at version 2 there are 2 activated tuples with keys 222-2 and 333-2. We use these 2 keys to read data and access Balance field of Client structure that is stored in history table. Result for version 2 is calculated by adding all extracted values for the version to the value of previously calculated version, therefore SUM stands as cumulative. Similar algorithm is used for COUNT and AVERAGE calculation.

An implementation detail that stands specific for Timeline Index on RamCloud is that all versions from the Timeline Index are read in parallel, using RamCloud `multiRead` functionality. This feature enables reading from master servers by sending just 1 RPC request for all the objects that are stored at the same master, and parallel RPC requests if objects are spread across different masters. Then for every read
version, we also perform another `multiRead` to gather all the entries from the history table that are associated with that version at once.

As previously mentioned, this algorithm does calculation for all the versions in the system from 1 to current, regardless if the version includes a change in aggregated table or not. However, there is a possible improvement. If Timeline Index stored NEXT_VERSION as part of the value in every entry, it would be possible to calculate aggregated result only for those versions. However, that creates another problem. It is not possible to know next version in advance, so that would require delayed commits.

**Selective aggregation**

We will take MAX aggregating function as an example of selective aggregation, and we will see how Timeline Index is used to calculate the result of the following query:

```
SELECT MAX(cl.Balance) AS max_balance
FROM Client cl
GROUP BY cl.version_id();
```

This query returns a vector of maximum balance values for every version in the system.

Considering the fact that MAX is selective aggregate, new extreme value could be any value from previously active set and tuples activated in the concerning version. Quick and easy implementation would be to update and sort set of all values for every version, but it creates a significant performance overhead. Timeline Index overcomes this problem already for nontemporal databases by using an algorithm inspired by online Skyline computation – Top-K. Timeline Index on RamCloud deals with the problem in the same way. Instead of fully sorting and updating all the values, only Top-K values are being sorted and updated in a multiset. Other activations and deactivations are being stored in a vector and are used only when Top-K multiset runs out of values. Experiments show that algorithm shows best performance if K is set to 2 percent of data set size.

As with cumulative algorithms, here as well we exploit benefits of `multiRead` RamCloud feature, when reading all the versions from timeline index, and when reading all the entries from history table for each version.

As for the cumulative functions, once again we are traversing timeline index by calculating MAX value for every version, regardless if the version update affected that table or not.

Algorithm for calculating MAX value for version 2 with K=2 is illustrated in Figure 4.
We can see that for version Top-K values are 200.00 and 150.00, and value 200.00 is selected as a result. Value 100.00, which was MAX value for version 1, has been moved to unused active values.

At version 3 key 222-2 gets invalidated and replaced with key 222-3. Since balance value for added key 222-3 is 0.00, and that is smaller than the minimum value in Top K structure, value 0.00 is added to unused activated tuples. Also, since key 222-2 becomes invalid, we have to update Top K structure by removing value 200.00. Figure 5 shows the look of TopK and Unused Activated vector after calculating MAX for version 3.

At version 4 key 333-2 gets invalidated. Therefore, value 150.00 is removed from Top K structure. We have come to the situation where there are no values in Top K structure, so we have to fill it using values from unused activated set, only if they haven’t been invalidated meanwhile (if they are not in unused deactivated). Thus values 100.00 and 0.00 are added to the Top K structure. Final look at Top K and unused activated vector at version 4 are given in Figure 6.
4.2.4. Time Travel

Time travel is most commonly used operator in commercial systems that use temporal data. This query is used to establish a consistent view on a previous version of the database, and as such, enables performing regular queries on a single, older version of the database [6].

We will see how Timeline Index is used for time travel on the following example:

```
SELECT count(*)
FROM Client cl
WHERE cl.Balance < 120
AS OF cl.version_id() = 3;
```

This query returns a single value that represents number of clients that bank had when database was in version 3.

For time travel, we need to establish consistent version by providing access only to those tuples that were valid in selected version. We achieve this by going to the nearest previous checkpoint, or if it doesn’t exist to beginning of the timeline index. Afterwards, we use multiRead feature of RamCloud to read all the versions from checkpoint version to our desired version, and then once again for each version to read entries from history table for keys associated with that version. Figure 7 shows how Timeline Index is used to perform time travel query.

![Timeline Index](image)

**Figure 7: Time Travel to version 3**

In our small example, checkpoints are made on every 2 committed versions. From Figure 7 we can see that if we want to establish a view on version 3, we go to the nearest previous checkpoint at version 2, and after that we read all the versions from checkpoint version + 1 to desired version, which is in this case just version 3. We conclude that there are 3 active tuples at version 3 (with keys 111-1, 333-2 and
at which point we can read them all from history table and count only those that suffice the balance value criteria.

### 4.2.5. Temporal Join

Temporal join is the most complex query described in this thesis. In addition to matching predicates, also version numbers of involved join pairs need to match. As a result, data needs to be selected along both value and time dimensions which leads to very difficult tradeoffs. Timeline Index focuses on the temporal aspect, considering that it should serve temporally selective queries. Its output is in fact extended timeline index, where the entries are not individual keys from tables, but key pairs, with one key for each joined tuple.

Example use case of temporal join would be the following:

```sql
SELECT count(*)
FROM Client TEMPORAL_JOIN Transactions
WHERE cl.Balance < 180.00
AND t.Value > 20.00
AND cl.key = t.cust_key
```

This queries temporal database how many times in history did it happen that customer with balance under 180.00 made a transaction of over 20.00.

To give an example of temporal join, we will use a little bit more complex database sample illustrated in Figure 9. We do the joining of tables A and B, such that A.PK = B.FK.

![Figure 8: Temporal Join](image)

<table>
<thead>
<tr>
<th>Table A</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PrimKey, VFrom</td>
<td>(PrimKey, VFrom)</td>
</tr>
<tr>
<td>W-100</td>
<td>(W, 100)</td>
<td></td>
</tr>
<tr>
<td>Y-100</td>
<td>(Y, 100)</td>
<td></td>
</tr>
<tr>
<td>Z-106</td>
<td>(Z, 106)</td>
<td></td>
</tr>
<tr>
<td>W-109</td>
<td>(W, 109)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index A</th>
<th>Version</th>
<th>Event List</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>(2, [W-100, Y-100]), (1)</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>(1, [Z-106]), (1)</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>(1, [W-109]), (1), [W-100])</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>(1), [Y-100, Z-106], (1)</td>
<td></td>
</tr>
<tr>
<td>132</td>
<td>(1), [W-109]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table B</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PrimKey, FKey, VFrom</td>
<td>(PrimKey, FKey, VFrom)</td>
</tr>
<tr>
<td>A-100</td>
<td>(A, Y, 100)</td>
<td></td>
</tr>
<tr>
<td>C-100</td>
<td>(C, Y, 100)</td>
<td></td>
</tr>
<tr>
<td>D-104</td>
<td>(D, W, 104)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index B</th>
<th>Version</th>
<th>Event List</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>(2, [A-100, C-100]), (1)</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>(1), [C-100]</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>(1), [D-104], (1)</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>(1), [2, A-100, D-104]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index C</th>
<th>Version</th>
<th>Event List</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>(2, [Y-100, A-100], [Y-100, C-100]), (1)</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>(1), [C-100]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Intersection Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
</tr>
<tr>
<td>Y</td>
</tr>
<tr>
<td>W</td>
</tr>
</tbody>
</table>

Figure 8: Temporal Join
Once again we are doing linear scanning of timeline indexes, starting from the smallest version number to the larger ones, taking entries from both timeline index structures at the same time. As with other temporal queries, we implement this linear scanning by first reading in all the entries from both indexes using RamCloud `multiRead` feature, and then going through all of them.

We use intermediate Intersection Map to store information about currently intersecting tuples. Entry in Intersection Map contains a pair of sets, one for each timeline index. Those sets contain corresponding history table entry keys. For example, at version 100 we activate 2 tuples in Table A (Y-100 and W-100, with PKs Y and W, respectively), and 2 tuples in Table B (A-100 and C-100, with FKs Y both). We can see that at version 100 there are 2 desired key matchings on primary key Y, and therefore we add an entry to the timeline index C with 2 pairs of joined tuples. Later, at version 103, we find out that there is an invalidation of entry in Table B with key C-100 and PK Y. As a result, we update Intersection Map by removing C-100 for key Y and table B, and we update Index C with new entry that tells us that there was one invalidation and that joined pair (Y-100, C-100) is no longer valid.

Intersection Map and Index C, unlike Indexes A and B, is not being stored in RamCloud, but in client’s local memory, as these are just intermediate structures.
5. Experiments and results
This section presents results that were exhibited when running experiments that assess performance of timeline index for 3 types of queries: temporal aggregation, time travel and temporal join. We will take results from Timeline Index without RamCloud as a baseline and then we will compare implementation of timeline index on RamCloud with the baseline.

5.1. Software and hardware used
Experiments were conducted on Euler ETH cluster. In every experiment 1 node was used for coordinator, 1 node for server and 1 for client.

<table>
<thead>
<tr>
<th>Hardware</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>X86_64</td>
</tr>
<tr>
<td>CPUs</td>
<td>8</td>
</tr>
<tr>
<td>Threads per core</td>
<td>1</td>
</tr>
<tr>
<td>CPU MHz</td>
<td>2394.07</td>
</tr>
<tr>
<td>Mem RAM</td>
<td>128 GB</td>
</tr>
<tr>
<td>L1d cache</td>
<td>32K</td>
</tr>
<tr>
<td>L2i cache</td>
<td>32K</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256K</td>
</tr>
<tr>
<td>L3 cache</td>
<td>10240K</td>
</tr>
</tbody>
</table>

Table 8: Hardware used

5.2. Benchmark
Benchmark dataset developed previously for timeline index has been used. It is based on customer use cases and standard benchmarks such as TPC-H and TPC-C [1, 10].

Benchmark databases are generated in 2 steps:

1) Generating version 0 of the database – created by dbgen tool provided by TPC-H benchmark. Scaling factor of dbgen tool when creating version 0 will from further on be addressed as SF<sub>0</sub>. This scaling factor took values 0.1, 1.0 and 10.

2) Generating further versions by executing TPC-C transactions, which are adapted to run on the TPC-H schema. Number of transactions determines the number of versions in the benchmark database, and represents SF<sub>H</sub>. This scaling factor took same values as SF<sub>0</sub> in every experiment.

Every experiment was performed 5 times and average values have been taken as results. Table 8 shows number of basic operations on every table of concern for SF<sub>0</sub> = 1.0 and SF<sub>H</sub> = 1.0, and final number of tuples in tables, at which point queries were benchmarked.
5.2.1. Basic operations – insert, update, delete
In Figure 9 we can see throughput of inserts, updates and deletes in operations per second. This is majorly influenced by the number of basic RamCloud operations (read, write, delete) and their complexity.

![Throughput Graph](image)

**Figure 9: Basic operations - Throughput**

5.2.2. Aggregation - SUM
The following query was performed:

\[
SELECT \text{sum(L\_EXTENDEDPRICE)} \text{ AS TOTAL} \\
\text{FROM Lineitem Li} \\
\text{WHERE li.L\_LINESTATUS = 'O'} \\
\text{GROUP BY Li.Version\_ID();}
\]

This query answers total value of all unshipped items at each versioned moment in history. Cumulative temporal SUM was performed on various table sizes with scaling factor 1.0.

<table>
<thead>
<tr>
<th>Table</th>
<th>Dbgen</th>
<th>Inserts</th>
<th>Updates</th>
<th>Deletes</th>
<th>Final # of Tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMER</td>
<td>0.2M</td>
<td>0.2M</td>
<td>0.6M</td>
<td>0</td>
<td>0.2M</td>
</tr>
<tr>
<td>LINEITEM</td>
<td>6.0M</td>
<td>1.6M</td>
<td>1.2M</td>
<td>0.2M</td>
<td>7.4M</td>
</tr>
<tr>
<td>ORDERS</td>
<td>1.5M</td>
<td>0.4M</td>
<td>0.3M</td>
<td>0.1M</td>
<td>1.8M</td>
</tr>
<tr>
<td>PARTSUPP</td>
<td>0.8M</td>
<td>0</td>
<td>1.2M</td>
<td>0</td>
<td>0.8M</td>
</tr>
</tbody>
</table>

Table 9: Number of operations
5.2.3. Aggregation–MAX

The following query was performed:

```
SELECT MAX(LI.L_EXTENDEDPRICE) AS MAX_PRICE
FROM Lineitem Li
WHERE Li.L_LINESTATUS = 'O'
GROUP BY li.VERSION_ID()
```

This query answers what is the price of the most expensive unshipped item at each versioned moment in history.

Parameter K’s value for Top-K algorithm was selected to be 2% of table size.

Benchmark results of selective temporal aggregation MAX are shown in Figure 11.
5.2.4. Time Travel

The following query was performed:

```
SELECT Count(*)
FROM Partsupp
WHERE PS_AVAILQTY < 100
AS OF TIMESTAMP '2012-01-01'
```

This query answers how often was the stock level of a product less than 100, on January 1\textsuperscript{st} 2012.

Time travel query was performed on a bigger scaling factor of 10.0, due to its better performance. Benchmark results are presented in Figure 13.
Figure 4: Time Travel
5.2.5. Join

The following query was performed:

```
SELECT count(*)
FROM Client TEMPORAL_JOIN Transactions
WHERE cl.Balance < 180.00
AND t.Value > 20.00
AND cl.key = t.cust_key
```

This query finds all the transactions with amount higher than 20.00 where clients balance is lower than 180.00.

Performance of join query is highly dependent on the number of tables that are being joined together. Also, accessing primary keys of table tuples has smaller price than accessing other columns, because for other columns we need to look into the history table. In this example, we have to look into both history tables.

---

**Figure 5: Join**
6. Conclusion and future work

In this thesis we explored storing and accessing temporal data by using Timeline Index on RamCloud. We tried to compare the performance of Timeline Index on RamCloud to the performance of Timeline Index in RAM on a single machine. Unfortunately, Timeline Index on RamCloud benchmarks show that query execution times are linearly dependent on the Temporal Table size and dominated by cost of basic RamCloud operations. This is accredited to the performance of Timeline Index Query Executor, which is not scalable enough, even though it performs parallel requests to RamCloud servers.

Therefore, future work on this topic would concentrate on improving Timeline Index Query Executor. With certain degree of parallelizing, it should be possible for it to exhibit performance that is better than the one Timeline Index exhibits on disks.
7. References


Table of Contents

1. Introduction ................................................................................................................................. 3

   1.1. Motivation............................................................................................................................... 3

   1.2. Problem statement .................................................................................................................. 3

   1.3. Thesis structure ..................................................................................................................... 3

2. RamCloud overview ................................................................................................................... 4

   2.1. Motivation ............................................................................................................................. 4

   2.2. Architecture .......................................................................................................................... 4

   2.3. Features ............................................................................................................................... 5

3. Timeline Index overview .......................................................................................................... 7

   3.1. Motivation ............................................................................................................................. 7

   3.2. Features ............................................................................................................................... 7

      3.2.1. Basic operations – insert, update, delete ......................................................................... 7

      3.2.2. Temporal Aggregation ................................................................................................... 9

      3.2.3. Temporal Join ................................................................................................................ 10

4. Timeline Index on RamCloud .................................................................................................... 11

   4.1. Architecture .......................................................................................................................... 11

   4.2. Implementation ..................................................................................................................... 11

      4.2.1. Storage ............................................................................................................................ 12

      4.2.2. Basic operations - Insert, update, delete ....................................................................... 13

      4.2.3. Temporal Aggregation ................................................................................................... 15

      4.2.4. Time Travel .................................................................................................................... 19

      4.2.5. Temporal Join ................................................................................................................ 20

5. Experiments and results ............................................................................................................. 22

   5.1. Software and hardware used ............................................................................................... 22
5.2. Benchmark ............................................................................................................................... 22
  5.2.1. Basic operations – insert, update, delete ........................................................................ 23
  5.2.2. Aggregation – SUM ......................................................................................................... 23
  5.2.3. Aggregation – MAX ....................................................................................................... 24
  5.2.4. Time Travel .................................................................................................................. 25
  5.2.5. Join ............................................................................................................................... 27
6. Conclusion and future work ..................................................................................................... 28
7. References ............................................................................................................................. 29