ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Master's Thesis Nr. 90

Systems Group, Department of Computer Science, ETH Zurich

Real-Time Analytics in a High Volume Event Processing System

by

Daniel Widmer

Supervised by

Lucas Braun
Prof. Donald Kossmann

September 1, 2013

inf | Informatik
Computer Science

Abstract

In recent years the need to process large volumes of data in real-time was fueled by advances in the field of memory and database technologies. Increases in memory capacity and reduced latency make it possible to keep entire databases in main memory. Modern networking technologies allow to distribute in-memory databases across hundreds or even thousands of nodes and process vast amounts of data in parallel.

The objective of this thesis is to design and implement a system, which can handle real-time analytical queries while being subject to updates at a high rate. Particular attention is given to address scalability, low latency and high throughput.

We evaluate different architectural options and devise two variants built upon *RAMCloud* and the *dense_hash_map* library.

Based on a detailed analysis of these approaches, we propose a column-oriented in-memory data structure and implement efficient query processing algorithms that allow the system to answer analytical queries in real-time. In order to achieve higher query throughput, we adopt the concept of *shared scan* and employ several query performance improvements.

We further present an experimental analysis of the proposed system and show that our approach is well-suited for processing large volumes of data.

Contents

1 Introduction

1.1 Background and Motivation

In 2013 the International Telecommunications Union, a United Nations agency, estimated that there are around 6.8 billion mobile phone subscriptions worldwide [10]. This number is believed to exceed the world's population in 2014.

With number of mobile phone subscriptions on the rise, phone service providers are keen on delivering the best possible service to their customers while maximizing their profit margin. Subscribers' usage of mobile networks creates vast amounts of data, from which valuable insights can be gained. The analysis of usage data allows the providers to make business decisions and create new custom-tailored offers to its growing base of subscribers. The systems facilitating these functionalities must be able to deal with a high rate of events, arising from subscribers using the provider's network. In order to keep up with today's fast-paced business world, data analysis is expected to be real-time and include up-to-date data. Moreover, the system is expected to be both efficient and scalable.

The vast amount of data at high update rates paired with time constraints make the implementation of such a system a compelling challenge.

1.2 System Overview

We propose Analytics in Motion (AIM), a system that allows for high volume stream processing, data storage and analysis.

Analytics in Motion is comprised of three components:

- The AIM server consisting of:
  - Stream and Event Processing (SEP)
  - Real-Time Analytics (RTA)
- Several SEP-clients
- Several RTA-clients

Figure 1 depicts an overview of the overall system.

An instance of the SEP-client creates events, which correspond to the meta data of telephone calls. Meta data consists of information such as the caller's id, the callee's id, call duration, call cost, etc. These events are sent to the AIM server at a rate of several thousand events per second (e.g. 100'000 events/sec).

The AIM-server is responsible for processing these events. Based on an event, the SEP-subsystem computes statistics for each subscriber and stores them in

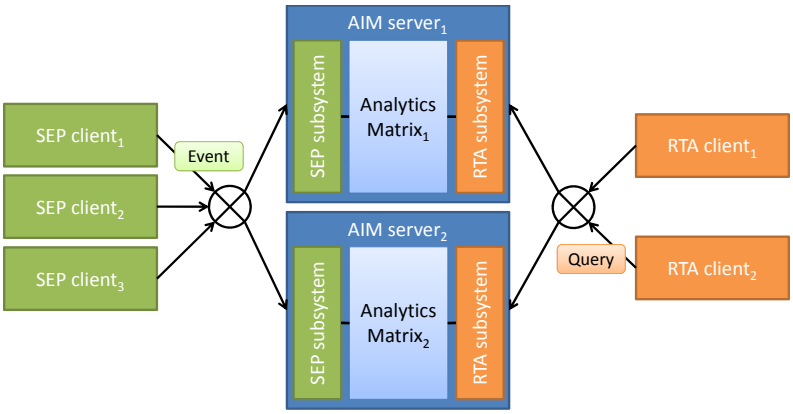


Figure 1: System overview

a large structure called *Analytics Matrix*. Conceptually the *Analytics Matrix* is a very wide table containing hundreds of subscriber attributes such as:

- number of local calls today
- number of local calls this week
- duration of calls this week
- etc.

These attributes are maintained by the SEP subsystem and updated according to the information obtained from the incoming events. The *Analytics Matrix* is a materialized view as we store aggregates such as SUM, MIN, MAX or COUNT instead of individual updates. This allows to look up desired values rather than computing them. For the AVG however, we keep a separate SUM and COUNT and compute the AVG on the fly.

In addition to updating the *Analytics Matrix*, the SEP subsystem matches incoming events to campaigns. A campaign is a rule which consists of a condition and a reward. When the condition is met, the subscriber receives the associated reward.

The RTA subsystem uses the *Analytics Matrix* to answer queries submitted by one or several instances of RTA-clients. The RTA subsystem scans through the *Analytics Matrix* and processes the queries before sending the results back to the RTA-clients.

This thesis is building upon the existing implementation of the SEP subsystem, which was part of a previous Master's thesis [6]. The thesis at hand focuses on the design and implementation of the AIM server's storage layer as well as the RTA subsystem and its query processing capabilities.

1.3 Problem statement

The goal of this thesis is to extend the existing AIM system with real-time analytics functionality. The AIM system shall be able to perform simple, ad-hoc real-time analytics (RTA) queries. These RTA queries can be divided into two groups:

- point queries: return one or several attribute values of a specific subscriber
- OLAP queries: perform a 2- or 3-way-join, aggregation and group-by operators

The system should be able to handle up to 5000 point queries at a time within 10 ms. OLAP queries should be performed within 100 ms, 100 queries at a time. The system should be able to handle a total of 100 million subscribers.

In addition to these requirements the initial requirements for the SEP subsystem [6] should still be fulfilled:

- Latency: Every event must be processed within 10 milliseconds.
- Cost: Minimize the number of machines needed in order to keep cost of hardware and operations low.
- Scalability: The system should be able to handle a growing (or shrinking) number of subscribers and adapt to a growing (or shrinking) population of subscribers.

1.3.1 Challenges

Since the SEP and the RTA subsystems operate on the same data, the storage layer has to be designed carefully. It has to take into account the requirements of both subsystems.

While the frequent updates coming from event data favors a key-value store, the execution of OLAP queries perform better on a column store. Therefore the storage layer has to be examined to find an optimum between a pure column store and a pure key-value store. In addition to implementing a well-suited storage architecture, appropriate query execution techniques have to be selected in order to meet the performance requirements.

1.4 Related Work

In recent years, systems to process very large amounts of data have gained traction. As the amount of data that needs to be processed is growing rapidly, enterprises are in need of new techniques to process these vast quantities of data. Most prominently, the MapReduce programming model and the Apache Foundation's implementation Hadoop [3] have sparked a lot of research in the area of distributed large-scale data processing.

Recent systems such as HyPer [5] intent to bridge the gap between OLTP and OLAP database systems by introducing a hybrid system that allows to process analytical queries directly on the transactional data. This is achieved by taking a transaction-consistent snapshot of the OLTP data through forking the OLTP process. Analytical queries can then be run on the snapshot.

HadoopDB [1] tries to bring together the OLTP and OLAP world by implementing a hybrid of MapReduce and DBMS technologies. It is scalable across several thousand nodes and features runtime scheduling and fault-tolerance. It allows to pose queries in a variant of SQL.

As proposed by [11] we make use of a technique called *differential updates*, which maintains a write-optimized delta partition to accumulate all data changes. The data in the delta structure is periodically merged with the read-optimized main partition. In contrast to [11] we do not perform any compression on the involved records. This makes merging fairly easy, but requires additional techniques to speed up query processing on the main partition.

1.5 Thesis Structure

The remainder of this thesis is structured as follows. First, section 2 presents design variants for the storage model and describes our implementation. Section 3 introduces our query processing strategy, which is evaluated in section 4. Section 5 concludes this thesis and points out future work.

2 Data and Storage Model

As the processing of analytical queries always relies on a scan operation, a full table scan on top of the storage structure has to be fast. In order to meet the performance requirements for query processing, the storage model of our system is of vital importance.

This section deals with the storage model of our system and its design variants.

2.1 Overview

Our system has to meet two objectives:

- The system needs to process and store continuous updates coming from the stream and event processing subsystem (SEP subsystem) at the rate of several thousand events per second.
- The data updated by the SEP system will be queried by the real-time analytics subsystem (RTA subsystem) and queries must be answered within strict timing requirements (see Section 1.3 for details on the timing requirements).

These goals are essentially conflicting as both the SEP and RTA subsystem operate on the same data. Improving scan performance diminishes the achievable update rates of the SEP subsystem and vice versa.

2.2 Data Model

We use a large table called *Analytics Matrix*. It contains one record per subscriber and stores all subscriber attributes. The *Analytics Matrix* stores aggregates such as total_number_of_calls_this_week, min_call_duration_this_month or sum_call_duration_today. Storing aggregates instead of individual updates has the advantage that individual aggregates can be retrieved fast as they don't have to be computed on-the-fly.

Since there can be many (several hundreds) attributes, the *Analytics Matrix* is a very wide table. In addition to the attributes, the table also stores foreign keys, which point to several dimension tables.

Figure 2 shows our data model.

Two architectural approaches for our use case were examined in [6]: (a) separated storage and (b) integrated storage.

Since the added RTA subsystem operates on the same data as the SEP subsystem, the architectural approaches layed out in [6] need to be revisited in order to find an architecture that can satisfy the needs for both subsystems.

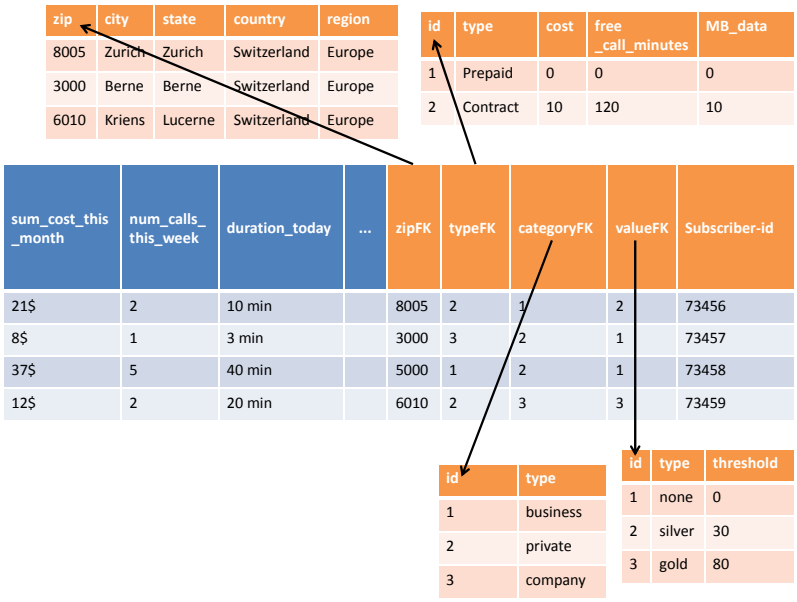


Figure 2: Data model

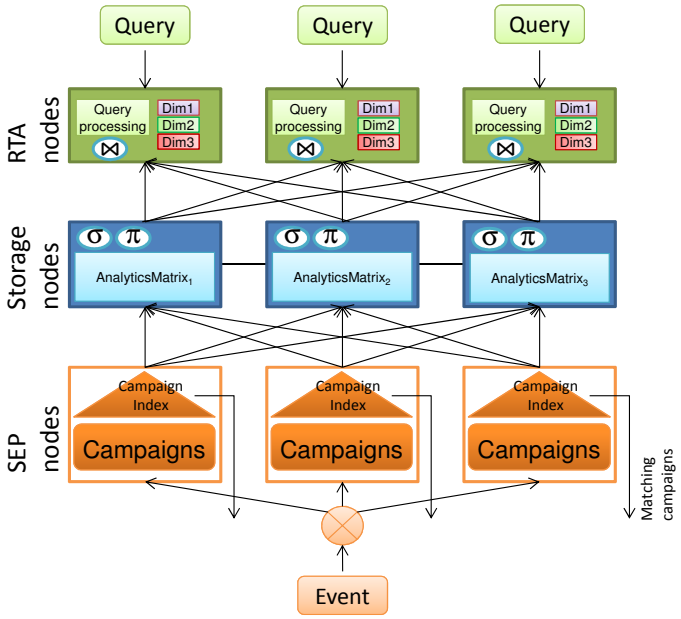
In the remainder of this subsection we explore design variants for the system's overall architecture and storage structure.

2.2.1 Separated Storage

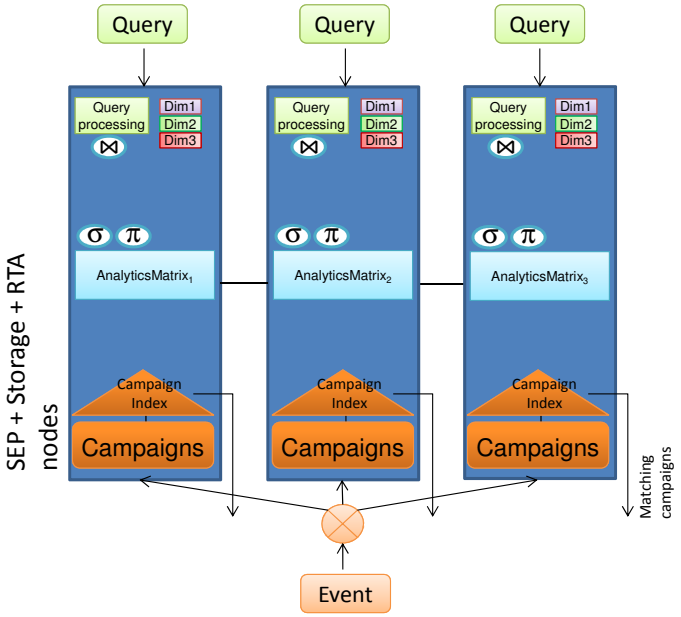
The separated storage model uses several detached storage nodes where the updates from the SEP subsystem are stored. There are nodes for real-time analytics (RTA nodes) responsible for query processing. Several nodes exist for stream and event processing (SEP nodes). Figure 3a shows the separated storage model.

Having storage on a separated layer has the advantage that all machines have access to all records of the *Analytics Matrix* and can thus process update events for every subscriber. Each RTA node can answer queries involving all subscribers by acessing the storage node of interest. A disadvantage of this approach is that the detached storage layer leads to increased latency as data has to be sent from and to these storage nodes.

Such a separated storage layer can be obtained by using RAMCloud [13], a fast in-memory key-value store that features low-latency and scalability. RAMCloud



(a) Separated storage model



(b) Integrated storage model

Figure 3: Separated and Integrated storage model

achieves its low-latency by keeping all information in DRAM, using hard disks only for backup. In combination with modern networking technologies such as InfiniBand [9], RAMCloud allows access to objects stored in remote and distributed DRAM. RAMCloud promises end-to-end access latencies of 5μs-10μs, which make it particularly suited for I/O-intensive workloads [12]. Currently RAMCloud offers a key-value store accessible through a simple read/write API. Tinnefeld et al. found that pushing down bandwidth-intensive database operators into the storage system is beneficial [14]. In order to efficiently perform queries we therefore need a scan algorithm on top of RAMCloud's internal data structures.

Implementing a fast scan on RAMCloud proved to be a challenge. In our micro benchmark we created 3KB records at random and inserted them into RAMCloud using consecutive integer-type keys. Our algorithm then iterated over RAMCloud's internal data structures and retrieved these records again. The measured response times were far beyond our timing requirements (see Figure 4). In addition, [6] found that requests to RAMCloud add significant overhead and lead to increased response times when RAMCloud is used for the SEP subsystem. Consequently we dropped RAMCloud and explored other storage structures.

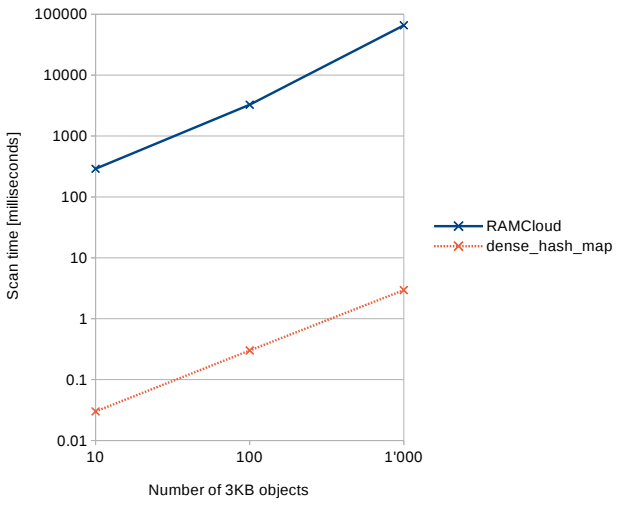


Figure 4: Results of micro benchmark, response time of scan algorithm, scanning 3KB objects on a single RAMCloud node vs. dense_hash_map.

2.2.2 Integrated Storage

In an integrated storage approach, every instance of the application runs on one single node, containing the storage subsystem as well as SEP and RTA capabilities. This architecture is commonly referred to as *shared-nothing*. In this approach, the *Analytics Matrix* is partitioned horizontally by subscriber-id and distributed across the nodes. When querying the subscriber data, partial results from all nodes have to be considered and merged into a final result set. A *shared-nothing* architecture allows to scale the system horizontally when more subscribers need to be added to the system. This is a property that will help us meet the scalability requirements.

Figure 3b depicts the integrated storage approach.

Previously, [6] determined that the integrated approach leads to smaller response times because it doesn't involve a high amount of network overhead when accessing storage. However, the integrated approach might involve large partial results, depending on the query issued. By using InfiniBand [9] networking technology, we will be able to reduce the network induced latencies resulting from transmitting partial results. Therefore the overhead of transmitting partial results and merging them into a final result set should be tolerable.

For the implementation of the integrated storage approach we use the *sparsehash* [8] library, a fast and memory-efficient data structure. Despite its name, it also provides a data structure for dense data called a *dense_hash_map*.[1] It allows to store key-value pairs and can be held entirely in main memory. Storing the entire data in RAM is beneficial as the latency for main memory access is far lower than accessing data on a hard disk drive. *dense_hash_map* offers fast lookups since the underlying C-array ensures data locality. We want to exploit this property in order to get a fast scan algorithm. Our micro benchmarks suggest that the achievable response times are far better than those we could achieve with RAMCloud (see Figure 4).

2.2.3 Hybrid Storage approach

Motivated by these numbers, we adapted the integrated storage architecture and introduced separate RTA-clients running on individual machines. Their responsibilities include sending queries to the AIM server and merging partial results coming from the AIM servers. Each RTA-client contains a copy of the dimension tables and is therefore capable of performing some computations independently. This allows us to shift some computational load from the AIM servers to the RTA-clients (see Section 3.2).

We also introduce separate SEP clients. They generate update events, which are then sent to the AIM server nodes.

[1] Since we use consecutive integer-type keys, our data set is dense by definition

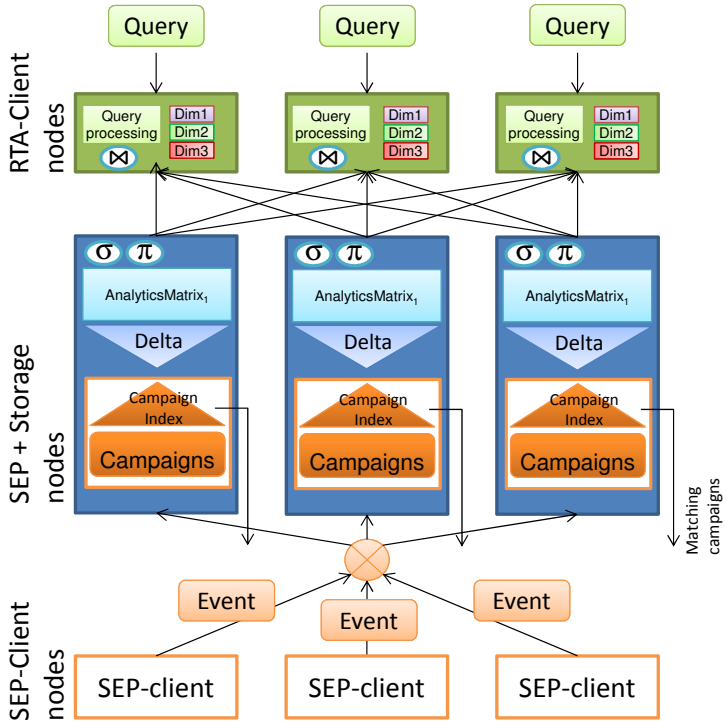


Figure 5: Hybrid approach

The AIM server nodes contain the SEP and RTA subsystem. The RTA subsystem is responsible for query processing. Processing of the incoming SEP events is currently performed on the AIM server. This adds additional computational burden on the AIM server. We chose this architecture for our current implementation, because our queries are simple and don't impose large computational complexity on our AIM servers. As a previous master's thesis determined [6], it is possible to detach the processing of SEP events from the AIM server nodes to introduce additional SEP processing nodes. A fully separated architecture still allows to meet the SEP's performance requirements (see section 5.1).

In our system, storage and computation take place on the same AIM server node and therefore our strategy is very similar to the integrated approach. However, storing the dimension tables on the RTA-clients and allowing the RTA-clients to do some of the computations is a departure from the purely integrated storage approach. We combine a server using integrated storage with additional client tiers. Hence we call this the hybrid storage approach.

Figure 5 depicts the hybrid storage approach.

2.2.4 Record structure

The records stored in the *Analytics Matrix* contain attributes such as:

- local cost today
- non-local cost today
- number of local calls this week
- number of non-local calls this week
- etc.

A full record usually contains several hundred attributes. Therefore an actual record is very wide.

To keep accessing the record of a subscriber simple, a previous implementation of the SEP subsystem concatenated the individual values to a record of approximately 3KB in size. Since the SEP subsystem updates an entire record once an event arrives, a contiguous chunk of memory is an appropriate choice.

Conversely, an RTA query typically operates only on a few attributes. Storing records as a contiguous chunk of memory is therefore detrimental to query performance, because the entire record has to be fetched from memory although only a subset of values are of interest. For the RTA subsystem we therefore favor a different record structure.

The following subsection describes our implementation of the storage structure and records.

2.3 Column Map

As stated in the previous subsection, storing a record contiguously means that an entire record has to be fetched when processing a query. Since a query usually only operates on a small subset of attributes, loading an entire record pollutes the caches and forces replacement of information that may be of interest in the future [2], which results in poor query performance. Thus it becomes clear that unnecessary accesses to main memory have to be avoided.

Ailamaki et al. [2] introduced *Partition Attributes Across* (PAX), grouping together all values of a particular attribute within each page. This leads to improved inter-record spatial locality and high data cache performance. When executing a sequential scan, multiple values for a single attribute are loaded into cache at the same time, resulting in better utilization of cache resources.

We want to exploit this property and therefore employ a storage structure following the PAX paradigm. We refer to our design as *ColumnMap*.

Figure 6 shows the basic structure of *ColumnMap*.

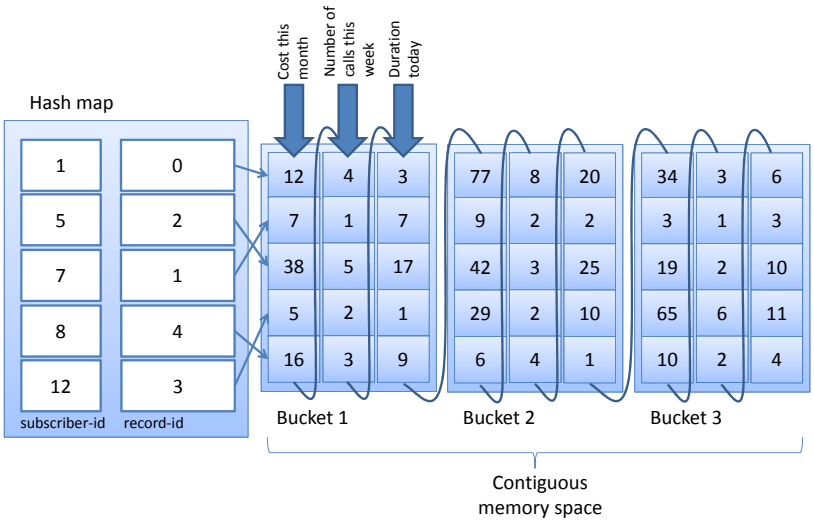


Figure 6: Basic structure of ColumnMap

The *ColumnMap* features a hash map which associates subscriber-ids with internal record-ids. Records are stored in a large chunk of consecutive memory.

We group a fixed number of records into logical blocks called buckets. In our current implementation, a bucket holds 1024 records. All buckets combined hold the entire *Analytics Matrix*.

Within a bucket, data is organized into columns. Each column holds the values for a particular subscriber attribute (e.g. cost_this_month). This approach allows us to increase inter-record locality, which is beneficial for scan processing of individual attributes (see Section 3).

Since records are of constant size and each bucket consists of a constant number of records, we can compute the address of a specific value from its record-id. This makes lookups for single values fast.

2.3.1 Example: Scanning the Analytics Matrix

Scanning the column of a particular attribute is done as follows. The algorithm retrieves the first bucket from main memory. It then computes the offset of the desired column's first entry. The algorithm then proceeds to sequentially access each individual value of the column (which are stored in consecutive memory locations) until the end of the column is reached. In figure 6 the direction of scan for each column is indicated by the arrows above bucket 1. Once the end of a bucket is reached, the algorithm repeats this procedure for every bucket, until the last bucket has been processed.

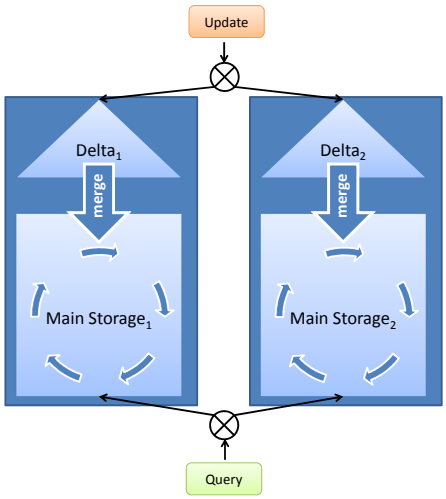


Figure 7: Two scan threads and their Delta and main storage

2.4 Dealing with updates

Our system has to deal with updates at a rate of several thousands per second (e.g. 100'000 update events per second). At the same time, the system intents to perform queries on the same data.

Similar to the technique of *differential updates* presented in [11] we use a buffer called *Delta* to accumulate updates. When an event arrives at the server, the SEP subsystem processes the event and stores the resulting updates in such a *Delta*. Our implementation uses a *dense_hash_map* [8] for the *Delta*, utilizing the subscriber-id as a key and storing a full row-based record.

As main storage structure we use the *ColumnMap* mentioned in the previous subsection. All queries are performed on the *ColumnMap* and updates are collected within our *Delta*. Periodically, the *Delta* is merged into storage (*ColumnMap*) and thus becomes available to the scan routine.

Figure 7 shows two scan threads, each with its own *Delta* and main storage structure. Updates coming from the SEP subsystem are stored in *Delta*. Later, they are merged into the main storage structure. The circular arrows indicate the scan operation being performed on the depicted main storage structure.

The overhead of the merge operation is determined by (a) the size of the *Delta* as well as (b) the complexity of the write operation when updating values in *ColumnMap*.

To ensure fast merging, we try to keep the *Delta* structure small. As a result, we have to merge frequently, which increases the overhead of the merge operation. We chose to merge *Delta* into storage once a full scan has been completed. This

ensures that the data available to the scan operation is consistent throughout the scan operation. The period between merge operations defines the freshness of the data available to the scan operation. We consider data updated after each scan cycle acceptable for our use case.

Instead of compressing the records as described by [11], we use an uncompressed column-oriented main storage structure (our *ColumnMap*) and store aggregates computed from the values of incoming events.

Before merging the updates into main storage, we create a new *Delta* and atomically switch pointers. Incoming updates are now deferred to the newly created *Delta* and the old *Delta* can be traversed and merged into main storage. Once the entire old *Delta* has been successfully merged into the *ColumnMap*, the old *Delta* is discarded.

While *Delta*s are being switched and merged into main storage, there can be no scans running. We ensure this by using spin-locks. Once a *Delta* has been completely merged into main storage, new scan operations can be started.

3 Query processing

3.1 Scan

Most OLAP queries include a full table scan, which makes it imperative to implement a fast scan. Obviously a table scan is greatly influenced by the number of records in the table. In the interest of speeding up full table scans, we partition the *Analytics Matrix* horizontally. We spawn several threads (typically 1-4) and let each thread handle the scan operation on a particular shard. Our current implementation uses a modulo function on the subscriber-id to determine the scan thread for a given subscriber. Since subscriber-ids are currently consecutive integers, subscribers are distributed evenly across all scan threads.[2]

In our implementation, each scan thread has its own *ColumnMap* and *Delta*. Hence, each *Delta* stores only a subset of updates and is later merged into the corresponding *ColumnMap*. Due to this coupling of *Delta* and *ColumnMap*, the number of threads handling SEP updates is equal to the number of scan threads. This design decision makes coordination between individual threads simpler, because we only have to synchronize between the SEP thread and the scan thread of a particular partition.

Traditional database system process one query at a time. Inspired by SharedDB [7] we try to achieve a higher throughput by using a batch-oriented processing

[2]In a real-world scenario, subscriber-ids are most likely not consecutive. In the light of deletes and inserts, a lot of care has to be taken when selecting a hash function to ensure even distribution across all threads. Each scan thread should be responsible for roughly the same amount of subscribers.

technique instead. Our server keeps a queue of queries that were submitted by the RTA-clients. Once a new scan is started, the queries in the queue are processed together in one single scan pass. Such a *shared scan* allows multiple queries to share the same scan. This batch-oriented processing technique reduces undue wait time for individual queries and allows to increase query throughput. In contrast to SharedDB [7] our implementation only allows for a shared scan (as opposed to sharing other calculations as well).

Each scan thread operates on its own *ColumnMap*. A scan operation proceeds bucket by bucket, each time calling a method *processBucket()* on all available queries. Listing 1 shows the bucket-wise processing in pseudo-code.

Listing 1: Query processing in pseudo-code

```
FOR EACH bucket IN buckets {
  FOR EACH query IN queries {
    query.processBucket(thread_id);
  }
}
```

3.1.1 Single instruction multiple data (SIMD)

Many current processors feature explicit *single-instruction multiple data (SIMD)* machinery such as vector registers and specialized instructions to manipulate data stored in these registers. They allow for one instruction to be performed on multiple data points in parallel.

Intel's *Streaming SIMD Extensions* (SSE) operate on registers of 128-bit or 256-bit width. The size of these registers allow to concatenate up to 4 floating-point operands into a single vector and process arithmetical or logical operations in parallel.

As Zhou et al. [15] noted, SIMD instructions allow a degree of parallelism and also often lead to the elimination of conditional branch instructions, reducing branch mispredictions. This makes SIMD instructions very useful for high-performance databases that are more often CPU bound than memory bound due to the increase in RAM capacities. We therefore exploit SIMD instructions to build a fast scan on our *ColumnMap*.

Filtering

When processing the WHERE-clause of a query, we first load the predicate's column from storage. We then load the predicate's operand into a vector and use vectorized operations to obtain a bit mask. This mask indicates whether a particular value should be selected (mask value 0xFFF...) or ignored (mask value 0x000...).

Example The query in Listing 2 aims to determine the amount spent on phone calls this week for all subscribers that have used their phone for more than 5 minutes. Figure 8 shows the computation of the bit mask from two operand vectors. Our scan algorithm proceeds bucket by bucket as mentioned in section 3.1. We determine the column offset for *duration_this_week* and for each bucket we load several values of *duration_this_week* into the vector registers. We then load the value 5 several times into another operand register and call SIMD_> to compare the values of *duration_this_week* with the value 5. This yields a bit mask, which we can use to filter all records that match this condition.

Listing 2: Example query

```
SELECT subscriber_id, cost_this_week
FROM AnalyticsMatrix
WHERE duration_this_week > 5;
```

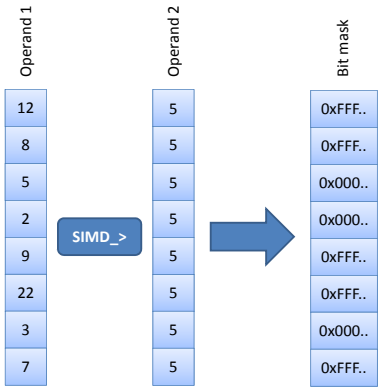


Figure 8: Creating a bit mask by employing SIMD-instructions

Projections

We can use the bit mask from the previous example to compute the projection of raw attributes.

In addition, SIMD also offers operations such as

- SIMD_MIN
- SIMD_MAX
- SIMD_+

which can be used to compute aggregates.

These operations allow to efficiently sum up values (SIMD_+) or compute minima (SIMD_MIN) or maxima (SIMD_MAX).

3.2 Dimension tables

Besides the *Analytics Matrix*, our data model includes four dimension tables:

- RegionInfo
- SubscriptionType
- SubscriberCategory
- SubscriberValue

The *Analytics Matrix* contains foreign keys, which reference these dimension tables. This corresponds to a star-schema, where a large fact table contains references to one or more dimension tables. In our case, the *Analytics Matrix* is the fact table referencing the four dimension tables.

Execution of a query typically involves a JOIN with one or more dimension tables. A dimension table can be stored on the AIM-servers or on the RTA-clients.

Storing the dimensions directly on the AIM-servers allows to fully compute a JOIN before it is sent back to the clients. This adds additional computational burden on the servers, which in turn impacts the performance of scans and SEP-updates.

Storing a copy of the dimension tables each RTA-client requires that these copies are maintained and updated across all RTA-client nodes.

We opted to store the dimension tables on the clients as this allows us to shift some of the computational complexity of the queries from the servers to the clients. For our purposes we assume that these client-side dimension tables are maintained regularly by batch-jobs.

Dimension tables are implemented as a in-memory hash map. They are populated once the RTA client starts. Since the dimension tables in our RTA benchmark [4] are small (approx. 4-10 records per dimension), scanning through the entire dimension is fast. For larger dimension tables a more suitable data structure would have to be selected.

3.3 JOIN queries

JOIN queries include the *Analytics Matrix* as well as one or more dimension tables. Since we decided to store the dimension tables on the RTA-clients, JOINs cannot be computed on the servers.

We keep a client-side lookup table, which contains the foreign keys for each subscriber. These foreign keys point to the primary keys of the dimension tables. The lookup table allows us to generate bitmaps, which can be used to filter subscribers on the server side.

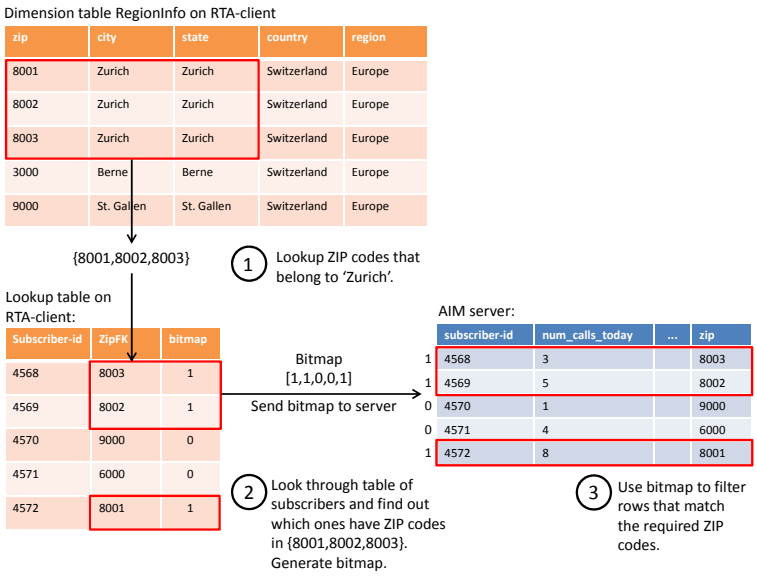


Figure 9: Answering JOIN queries by using a bitmap

The following example illustrates the use of bitmaps.

3.3.1 Example: Using a bitmap for JOIN queries

Figure 9 shows an example how a bitmap can be used to process JOIN queries for the query defined in listing 3.

Listing 3: Example JOIN query

```
SELECT total_cost_this_week
FROM AnalyticsMatrix, RegionInfo
WHERE RegionInfo.zip = AnalyticsMatrix.zip
AND RegionInfo.city = 'Zurich'
```

(1) The RTA client first scans through the dimension table RegionInfo, stored on the client to find all zip codes belonging to 'Zurich'. (2) Besides the dimension tables, the client also stores a lookup table, which contains all subscriber-ids and their foreign keys (zip in our case). The list of zip codes is now used to find all matching customers. If the zip in the lookup table corresponds to one of the zip codes in the list, a '1' is added to the bitmap. In case the zip code does not match any of the zip codes in the list, the bitmap entry is marked with '0'. The obtained bitmap is then sent to the AIM server. (3) The AIM server uses the bitmap to filter subscribers which match the required condition (city='Zurich').

4 Performance Evaluation

In this section we present a performance evaluation of the AIM system by executing a workload of aggregation and JOIN-queries. We measure average end-to-end response time and average query throughput.

4.1 Experimental Setup

Our experiments are conducted on servers equipped with a dual-socket 4 core Intel Xeon E5-2609 CPU, each core operating at 2.40 GHz. The system features 32KB L1 cache, 256KB L2 cache and 10240KB L3 cache.

The machines hold 4x32GB Samsung DDR3-DIMM, resulting in a total of 128GB RAM. We use Debian Linux 4.6.3-1 as operating system and communicate over InfiniBand or TCP.

4.2 Population of Analytics Matrix and Dimension tables

For the *Analytics Matrix* we use a set of 54 unique attributes and repeat them 9 times to make up for a subscriber record of 486 attributes.

Throughout the experiments, the number of attributes in the *Analytics Matrix* remains constant at 486. Together, the attributes make up for a record of approximately 3 KB in size.

The *Analytics Matrix* is populated from a MySQL database at startup and initialized with default values (i.e. number_of_calls_today = 0). Since population of the *Analytics Matrix* is done before queries are executed, the performance of initialization is not measured.

Dimensions are stored on each RTA-client. They are populated with pre-defined values according to the RTA Benchmark [4]. Population takes place as soon as an instance of RTA-client is started. As they are initialized prior to submitting any queries, the time needed for initialization is not measured.

4.3 Initialization of SEP subsystem

The SEP subsystem's campaigns are stored in the same database as the definition of the *Analytics Matrix* attributes. It is used to initialize the campaigns at start time. For our experiments we use 300 campaigns.

The time spent on initialization is not measured because it takes place before the system starts to process events and queries.

4.4 Workload

For the first part of our experiments we devised 3 simple queries using AVG, MAX and MIN operators. The queries are defined as follows:

Listing 4: Workload queries 1-3

```
Query 1:
SELECT AVG(total_duration_of_calls_this_week)
FROM AnalyticsMatrix
WHERE number_of_local_calls_this_week > ALPHA;

Query 2:
SELECT MAX(most_expensive_call_this_week)
FROM AnalyticsMatrix;

Query 3:
SELECT MIN(duration_of_calls_this_week),
MAX(duration_of_calls_this_week)
FROM AnalyticsMatrix
WHERE number_of_calls_this_week < BETA;
```

where *ALPHA* and *BETA* are integer type parameters, drawn uniformly at random from [0,2] and [2,5] respectively.

For our purposes we also define a simple JOIN query to evaluate our query processing strategy (see Listing 5).

Listing 5: Query 4

```
SELECT subscriber_id, name, address,
MIN(avg_cost_of_calls_this_week / ↩
    avg_duration_of_calls_this_week) AS flat-rate
FROM AnalyticsMatrix, SubscriberValue v
WHERE AnalyticsMatrix.value = v.id
```

This query serves the purpose of determining which subscriber pays the least for their calls. We compute the fraction of average cost of a call divided by the average duration of a call. We then compute the minimum of these fractions.

For our experiments we use a mixed workload comprised of queries 1 through 3 at a ratio of 40 to 40 to 20. In addition we use an aggregation-only workload, featuring queries 1 and 2 at a ratio of 50 to 50. Query 4 (Listing 5) is measured separately by using a workload of 100% query 4.

For each experiment, an RTA-client uses several threads (e.g. 8) to send queries to the servers. An RTA-client runs for a pre-defined amount of time and sends query after query to the AIM servers. Queries are submitted in a closed-system manner (i.e. before a new query can be submitted, an RTA-client thread waits for the previous query to finish).

Unless otherwise stated, each experiment runs for 1 hour and is preceeded by a 15 minute warm-up period where the SEP-client continuously sends updates to the AIM server nodes.

We measure the end-to-end response time for each query and compute the average response time in milliseconds as well as the query throughput.

4.5 Experimental Results and Analysis

4.5.1 Handling large numbers of subscribers

A first experiment measures the average query response time for an increased number of subscribers on 1 AIM server node. We use a mixed workload and the RTA-client uses 8 threads to submit queries concurrently. The number of updates per second is kept constant at 100'000 updates/sec.

Figure 10 shows that the average query response time increases as we add more subscribers.

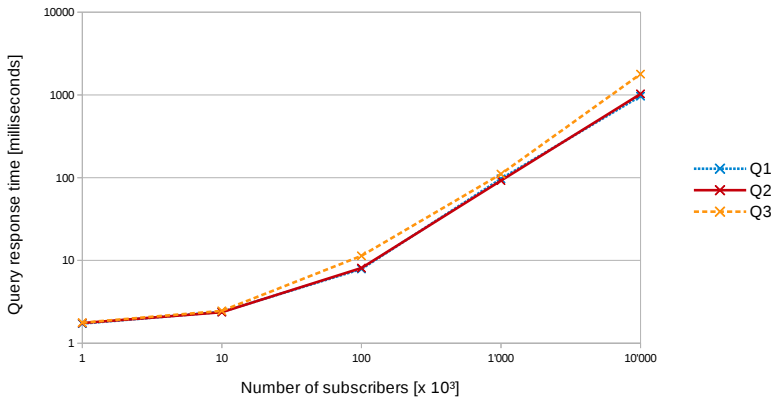


Figure 10: Increasing the number of subscribers on 1 AIM server node

It is obvious that the system requires more time to scan the entire *Analytics Matrix* when the number of subscribers increases.

A possible strategy for dealing with an increasing number of subscribers is to add additional AIM server nodes.

4.5.2 Adding more AIM server nodes

Figure 11 shows the average query response time as AIM server nodes are added to the system. We use one machine to run the SEP-client and another to run the

RTA-client. The RTA-client uses 8 threads to submit queries from the mixed workload. Each instance of the AIM server is executed on its own machine. With each AIM server node, we add 10 million subscribers. The SEP update rate is increased by 100'000 updates/sec every time a new AIM server node is added to account for an increased usage of the provider's network.

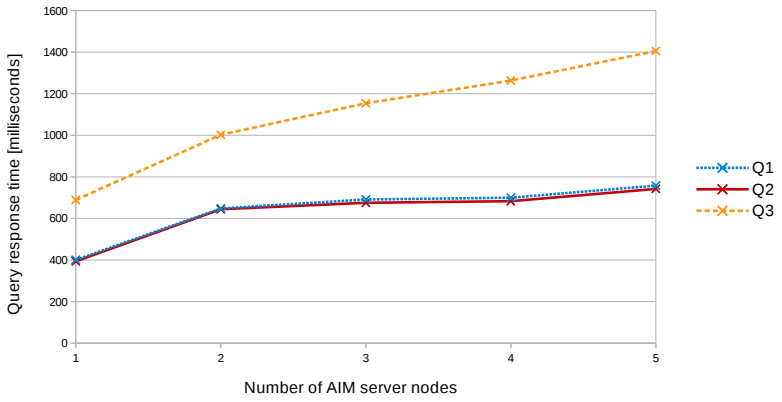


Figure 11: Scaling from 1 to 5 AIM servers, each time adding 10 Mio subscribers to the overall system

Ideally, we would expect the response time to remain stable (a horizontal line), because additional server nodes allow to distribute the increased load. Figure 11 shows the measured results.

The increase in response time when going from 1 to 2 servers can be explained by the overhead of merging partial results. For 1 AIM server node, there is no need to merge any results, which leads to shorter response times.

The esponse time of query 3 increases faster than response time for queries 1 and 2. This is most likely due to the fact that query 3 needs to transmit many partial results over the network. In order to achieve better response time for query 3 and similar queries, further optimizations and query processing techniques have to be explored.

Figure 11 shows that the average query response time increases only slightly for query 1 and 2 when additional AIM server nodes are added. We therefore conclude that the system scales well for query 1 and 2 and can handle an increasing number of subscribers.

4.5.3 Increasing the SEP update rate

Our server nodes need to be able to deal with a continuous stream of updates. The following experiment aims to determine our system's sensitivity to high

update rates.

We measure the average query response time when increasing the number of updates per second. In this experiment we use 1 AIM server node with 4 scan threads. We use 1 RTA-client with a mixed workload and submit 8 queries concurrently. The experiment's duration is 30 minutes.

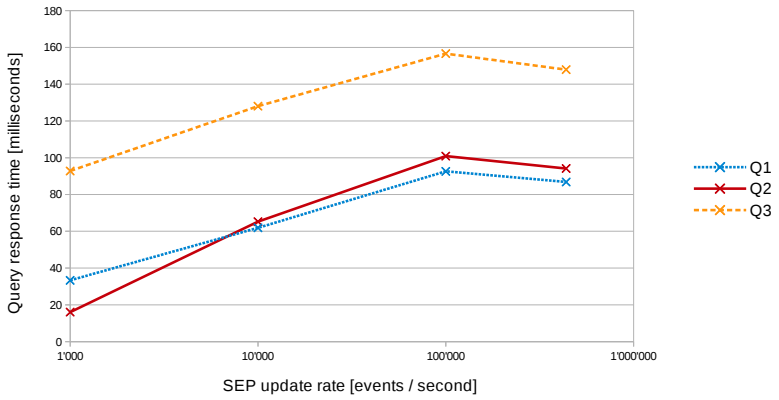


Figure 12: Increasing the number of SEP updates per second

The results in figure 12 show that the query response time is increased when the update rate increases from 1'000 to 100'000 updates/sec.

Although the SEP-client tries to send 1 million updates per second, the effective update rate measured was 437'500 updates per second. We attribute this difference to networking limitations. Figure 12 shows a slight decrease in the average query response time for 437'500 updates/sec.

In the light of results depicted in Figure 12 we conclude that the query response time is affected by the update rate. Further experiments are required, to examine the system's stability under even higher update rates.

4.5.4 Increasing the number of scan threads

In order to explore the impact of the number of scan threads on the system's performance, we measure the average scan response time when varying the number of scan threads. The SEP update rate is fixed at 100'000 updates/sec and 1 AIM server node is used. The experiment is conducted over a period of 30 minutes.

Figure 13 shows that for an increasing number of scan threads, the response time decreases.

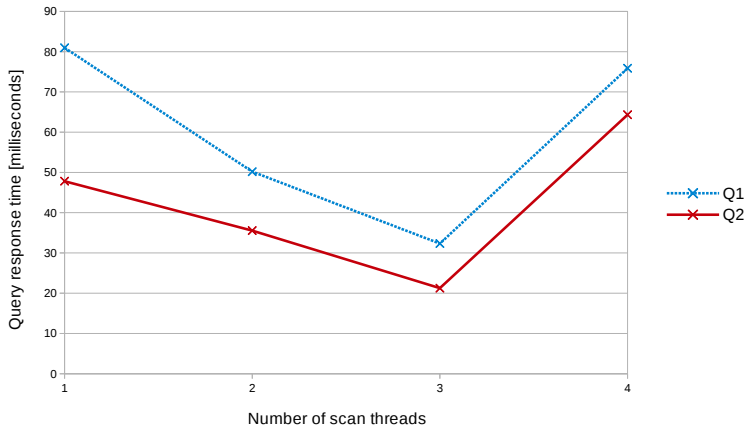


Figure 13: Increasing the number of scan threads

As a result of an increased number of scan threads, each thread has to scan fewer records.[3] With a larger number of scan threads, more partial results have to be merged. This partly explains, why the response time decrease is not strictly linear.

We use one thread for communication with the SEP-client and another thread for communicating with the RTA-client. Together with 3 threads for handling SEP updates and 3 scan threads, this yields 8 threads in total. As the experimental results suggest, our 8-core machine can schedule these 8 threads efficiently. We therefore suspect that the optimal number of scan threads for our hardware lies at 3.

On the other hand, using 4 threads for handling SEP updates and 4 threads for scanning might cause resource contention, leading to the observed increase in response time for 4 scan threads.

When running the same experiment on a machine equipped with 512 GB RAM, 4 x Intel Xeon E5-4640 CPU at 2.4GHz and 32 cores in total, we obtain the results depicted in figure 14.

For each number of scan threads we measure the average query response time when submitting 1 query at a time. Each measurement is performed over a period of 30 minutes.

Figure 14 shows that the average response time decreases when the number of subscribers are partitioned across more scan threads. The response time goes down quickly when using 2-8 scan threads. When using more than 12 threads,

[3]Since we use a modulo function to distribute the subscribers across the individual scan threads, the number of records each thread has to process is $\frac{total\_number\_of\_subscribers}{number\_of\_threads}$

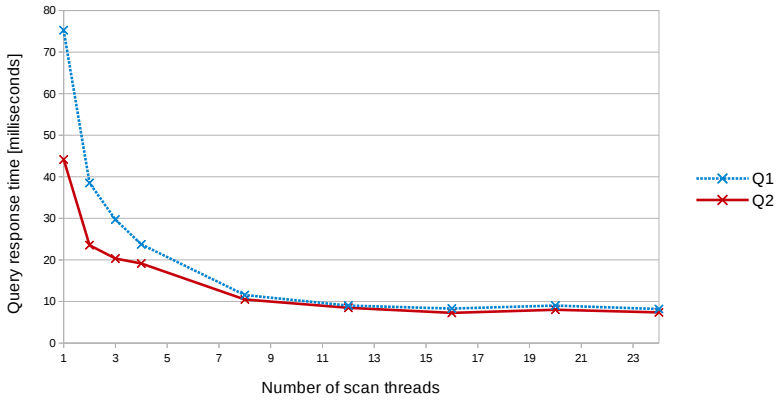


Figure 14: Increasing the number scan threads

the average query response time drops below 10 milliseconds and remains fairly stable.

In contrast to the results from the earlier experiment (Figure 13), the results in figure 14 show no unexpected increase in the query response time. This suggests that we don't experience any resource contention when running the experiment on an Intel Xeon E5-4640 CPU.

Figure 14 shows that it's possible to meet the timing requirements and execute simple queries within 10 milliseconds or less.

4.5.5 Query throughput

Figure 15 shows the query throughput (queries per second) for query 1 as measured at the RTA-client. It shows that for 1, 2 and 3 scan threads, the query throughput increases with the number of scan threads. This supports our argument, that more scan threads lead to a better scan response times and higher query throughput. More scan threads means that each thread has to process fewer subscribers, which leads to faster query processing and higher throughput.

For 4 scan threads, the system shows a slightly diminished query throughput. We attribute this effect to the previously mentioned resource contention on the used 8-core machine.

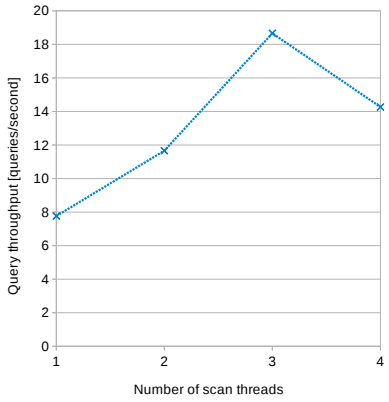
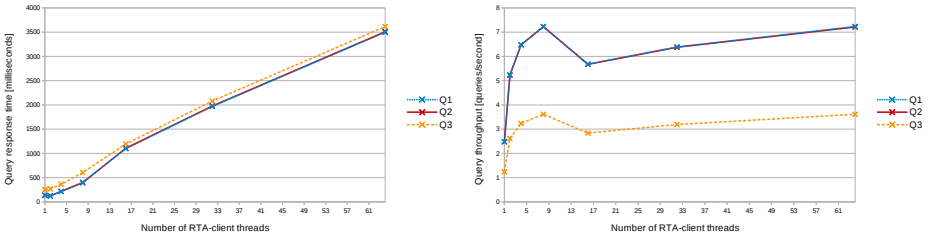


Figure 15: Increasing the number of SEP updates per second



(a) Query response time

(b) Query throughput

Figure 16: Increasing the number of RTA-client threads

4.5.6 Adding more queries

Figure 16a shows the average query response time as the number of concurrently submitted queries is increased. We use 1 AIM server node, processing 4 scan threads in parallel. The SEP update rate is fixed at 100'000 updates/sec.

The results show that the average query response time increases linearly with the number of parallel queries submitted by the RTA-clients.

Figure 16b shows that the query throughput of the system increases as more RTA-client threads are used. The throughput peaks around 8 threads. The drop in throughput around 9 threads could be due to scheduling issues of the 8-core CPU used in our experiments. For more than 8 threads the throughput increases linearly. This result indicates that our implementation of *shared scan* helps to increase the throughput, but also leads to far higher response times.

4.5.7 JOIN query

Figure 17 shows the average response time measured for query 4. We use 1 AIM server node and 4 scan threads. The SEP update rate is varied between 1'000 and 100'000 updates/sec.

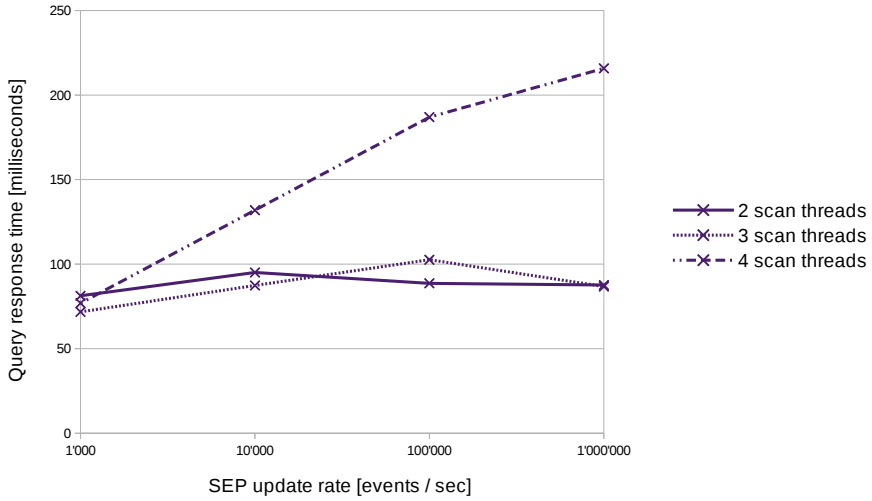


Figure 17: Response time for JOIN query

For 2 and 3 threads, the response time only increases slightly for an increased SEP update rate. For 4 scan threads, the response time increases drastically. We suspect that this can be attributed to resource contention on the used 8-core machine.

The results of this experiment show that our approach to process JOIN queries works well for 2 and 3 scan threads. Furthermore, the results show that the average query response time for our JOIN query is below the required 100 milliseconds.

5 Conclusion

In this thesis we propose and implement a system that is capable of performing real-time analytical queries while at the same time being subject to high update rates.

Our motivating use case comes from the telecommunications industry, where a provider wants to be able to process and analyze the usage data of its subscribers. A subscriber's usage of the provider's network generates update events, which need to be processed and stored. Providers want to query this data in real-time to gain insights about the subscribers' behavior.

The proposed system uses a shared nothing architecture and partitions data horizontally. In order to keep up with high update rates, we use a buffer to store incoming updates. This buffer is periodically merged into main storage, where it becomes available to the query processing threads. As our main storage structure we use a column oriented memory structure we call *ColumnMap*. The memory structure allows for fast scans of subscriber attributes. In our scan algorithm we make use of SIMD instructions which allow to process multiple values at once. All data is held in main memory and partitioned horizontally across several threads.

In addition to presenting our reasoning for our design, we also conducted a set of experiments, measuring the performance of the overall system.

Experimental analysis has shown that the proposed system is able to answer queries within the required timing restrictions. Simple queries can be answered within 10 milliseconds and a JOIN query within approximately 100 milliseconds, all while processing 100'000 update events per second. Furthermore, our experimental analysis revealed that the system scales well for simple queries and an increasing number of subscribers.

The system implemented demonstrates that processing updates and query evaluation can both be performed on the same in-memory data structures. Therefore, the presented system constitutes a solid basis for further improvements.

5.1 Future Work

The research presented in this thesis suggests many opportunities for future work, some of which are outlined in this subsection.

Bottleneck analysis As Zhuravlev et al. noted [16], cache contention might not be the only reason for performance degradation on multi-core systems. Other factors like memory controller contention, memory bus contention and prefetching hardware contention all need to be taken into consideration when optimizing a multi-threaded application. In order to speed up the current implementation of the AIM system, these limiting factors need to be addressed. As [16] suggests, using a cache-aware scheduling algorithm might be helpful to further improve performance. Our results in section 4.5.4 demonstrate, the system's hardware capabilities play a vital role in our system's behavior when multiple scan threads are used. Further optimization is needed to improve our system's performance on our target hardware.

Purely separated architecture As [6] found, a purely separated approach still allows to achieve the performance goals of the SEP subsystem. For the sake of simplicity and because our current queries were simple in nature, our current implementation keeps the SEP subsystem on the AIM server nodes. In a future version, the AIM system should be implemented with a fully separated architecture. This will lead to even better scalability and allows to shift some computational burden from the AIM servers to the SEP processing nodes.

Adaptive data partitioning In a real-world system, some subscribers might be more active than others. Depending on the hash function used for distributing the subscribers across the AIM server nodes, subscribers' activity patterns might lead to heavily used nodes. This form of data skew might create "hot" nodes, which can become a bottleneck of the overall system and slow down response times. In order to avoid "hot" nodes, a suitable hash function or load balancing mechanism has to be selected carefully.

Compression In addition to load balancing, compression of less used data might also be useful. Funke et al. distinguish between heavily used ("hot") data items and rarely used ("cold") data items [5]. This allows them to optimize access to "hot" data by moving it to small memory pages. Rarely used data items are compressed and stored on huge memory pages. This approach might be beneficial for our system since some data items might be rarely queried while others are accessed frequently. This will most likely result in higher response times for queries that include rarely used data items. However, depending on the exact circumstances of the real-world use case this might be acceptable.

GROUP-BY operator As an extension to the current system, grouping could be implemented on the AIM server. In order to be able to handle GROUP-BY clauses which operate on dimension table attributes, RTA clients need to be able to regroup partial results on the client side. This is due to the fact that we decided to store the dimension tables on the client-side instead of storing them on each server. A server can only compute groupings on a dimension's foreign keys (e.g. GROUP BY RegionInfo.zip). If a grouping is desired to be more coarse-grained (e.g. GROUP BY RegionInfo.City), the server can group by zip and the clients regroup by city with the help of the dimension tables.

Persistence of data In order to avoid complete data-loss in case of a power-failure, a persistence mechanism needs to be added. This could be achieved by continuously taking snapshots and writing them to disk. Since our system is exposed to a high rate of updates, the snapshotting mechanism needs to be lightweight. Special attention has to be given to the mechanism's impact on the system's overall performance.

5.2 Acknowledgements

I would like to thank Prof. Donald Kossmann for the encouragement and advice he has provided throughout my master's thesis. His enthusiasm for the subject matter is contagious and has substantially contributed to the spirit of this master's thesis.

I also like to thank my supervisor, Lucas Braun, for his continuous support and patience. He provided many helpful suggestions and guidance throughout this endeavour. He was responsible for designing and implementing the network protocol and SEP/RTA clients.

A special thanks goes to Thomas Etter and Georgios Gasparis for their implementation of *ColumnMap*.

References

[1] Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., and Rasin, A. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proc. VLDB Endow. 2*, 1 (Aug. 2009), 922–933.

[2] Ailamaki, A., DeWitt, D. J., Hill, M. D., and Skounakis, M. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 2001), VLDB '01, Morgan Kaufmann Publishers Inc., pp. 169–180.

[3] Apache Foundation. Hadoop. `http://hadoop.apache.org/`, 2013. [Online; accessed August 2013].

[4] Braun, L., Gasparis, G., and Kauffmann, M. RTA Benchmark. ETH Zurich, Systems Group, 2013. [Internal specifications].

[5] Funke, F., Kemper, A., and Neumann, T. Compacting transactional data in hybrid OLTP & OLAP databases. *Proc. VLDB Endow. 5*, 11 (July 2012), 1424–1435.

[6] Gasparis, G. AIM: A System for Handling Enormous Workloads under Strict Latency and Scalability Regulations. *Master's thesis No.78, Systems Group, Department of Computer Science, ETH Zurich* (2013).

[7] Giannikis, G., Alonso, G., and Kossmann, D. SharedDB: killing one thousand queries with one stone. *Proc. VLDB Endow. 5*, 6 (Feb. 2012), 526–537.

[8] Google. Sparsehash. `https://code.google.com/p/sparsehash/`, 2012. [Online; accessed June 2013].

[9] InfiniBand Trade Association. InfiniBand. `http://www.infinibandta.org/`, 2013. [Online; accessed April 2013].

[10] International Telecommunication Union. The World in 2013: ICT Facts and Figures. `http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2013.pdf`, 2013. [Online; accessed August 2013].

[11] Krueger, J., Kim, C., Grund, M., Satish, N., Schwalb, D., Chhugani, J., Plattner, H., Dubey, P., and Zeier, A. Fast updates on read-optimized databases using multi-core CPUs. *Proc. VLDB Endow. 5*, 1 (Sept. 2011), 61–72.

[12] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Ongaro, D., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R. The case for RAMCloud. *Commun. ACM 54*, 7 (July 2011), 121–130.

[13] Stanford University. RAMCloud. `https://ramcloud.stanford.edu/`, 2013. [Online; accessed May 2013].

[14] Tinnefeld, C., Kossmann, D., Grund, M., Boese, J.-H., Renkes, F., Sikka, V., and Plattner, H. Elastic online analytical processing on RAMCloud. In *Proceedings of the 16th International Conference on Extending Database Technology* (New York, NY, USA, 2013), EDBT '13, ACM, pp. 454–464.

[15] Zhou, J., and Ross, K. A. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2002), SIGMOD '02, ACM, pp. 145–156.

[16] Zhuravlev, S., Blagodurov, S., and Fedorova, A. Addressing shared resource contention in multicore processors via scheduling. *SIGARCH Comput. Archit. News 38*, 1 (Mar. 2010), 129–142.