Analytics in Motion

High Performance Event-Processing AND Real-Time Analytics in the Same Database

Lucas Braun, Thomas Etter, Georgios Gasparis,
Martin Kaufmann, Donald Kossmann, Daniel Widmer
Systems Group / Department of Computer Science
ETH Zürich, Switzerland
{braunl, etterth, ggaspari, martinka, donaldk, widmedan}@ethz.ch

Aharon Avitzur, Anthony Iliopoulos, Eliezer Levy, Ning Liang
Huawei Technologies
European Research Center
{aharon.avitzur, anthony.iliopoulos, eliezer.levy, harry.ning}@huawei.com

ABSTRACT

Modern data-centric flows in the telecommunications industry require real time analytical processing over a rapidly changing and large dataset. The traditional approach of separating OLTP and OLAP workloads cannot satisfy this requirement. Instead, a new class of integrated solutions for handling hybrid workloads is needed. This paper presents an industrial use case and a novel architecture that integrates key-value-based event processing and SQL-based analytical processing on the same distributed store while minimizing the total cost of ownership. Our approach combines several well-known techniques such as shared scans, delta processing, a PAX-fashioned storage layout, and an interleaving of scanning and delta merging in a completely new way. Performance experiments show that our system scales out linearly with the number of servers. For instance, our system sustains event streams of 100,000 events per second while simultaneously processing 100 ad-hoc analytical queries per second, using a cluster of 12 commodity servers. In doing so, our system meets all response time goals of our telecommunication customers; that is, 10 milliseconds per event and 100 milliseconds for an ad-hoc analytical query. Moreover, our system beats commercial competitors by a factor of 2.5 in analytical and two orders of magnitude in update performance.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design—*Access methods*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

Keywords

OLTP/OLAP Engine; Analytics, Event-Processing

1. INTRODUCTION

Many data-centric flows in the telecommunications industry start with a high-volume stream of events. These events are often represented as variations of Call Detail Records or Charging Data Records (CDRs) that are generated by probes at nodes of the network (e.g., base stations or routers). These events must be processed in real-time in order to maintain an accurate picture of the state of the network, which is the basis for managing the network effectively. This state is, in turn, represented by a large number (hundreds) of indicators. Examples of such indicators are billing and CRM-related information such as the total call duration per day per subscriber and indicators related to the network quality of service such as the number of dropped calls for each cell.

The data processing requirements for network monitoring are significant. In recent years, a number of additional applications have emerged that need to analyze CDR events in real time (e.g., marketing campaigns and quality of service and experience [32]) and the demands are growing as these applications are becoming more sophisticated. They typically involve a large number of ad-hoc queries on the key indicators of the network, or usage of classes of subscribers. These queries need to be answered in real time (within tens of milliseconds) and need to operate on fresh data. It is not acceptable to load the CDR data into a data warehouse and run the new, emerging analytics applications against the data warehouse because that would violate the data freshness requirements. Furthermore, such a data warehouse would be too expensive as it would require to store the data twice and it would significantly increase the overall TCO.

To address such mixed workloads that involve a high volume of events and carrying out complex analytics in real-time on these events, a number of alternative architectures have been proposed in the past. One approach is to scale out in a cluster of machines. The most prominent systems

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright © 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2742783.

that enable such scale out are Apache Storm [4] for event processing, Apache Hadoop [5] or Apache Spark [37] for the analytics, and Druid [36] for real-time analytics and fast data ingression. While these systems scale well with the number of machines, they are simply not fast enough in event processing if hundreds of indicators have to be maintained at a speed of 100,000s of events per second as it is usually the case in the telecommunications industry. Moreover, they copy events multiple times between the event processing system (e.g., Storm) and the analytics system (e.g., Hadoop) which is a costly operation. For instance, we tested Storm on our workload and found that it was able to handle only some dozens of indicators at a rate of several thousands events per second on a cluster of 12 commodity machines. While Druid achieves sufficiently high event processing rates, the reported numbers were also obtained from only few dozens of indicators (metrics).

In the other extreme, there has been recent work to handle mixed OLTP and OLAP workloads on a single machine in an integrated way; e.g., [21, 2, 17]. While the proposed systems achieve extremely good performance and avoid the cost of moving data around, they are too limited because they cannot scale beyond a single machine. As applications get more demanding, it is critical to go beyond the processing capabilities of a single machine.

The purpose of this paper is to present the AIM system which was designed to process mixed workloads with high volume of events, a high and diverse analytical query workload, and strict response time guarantees for event stream processing and analytic queries. The system must scale as well as distributed systems such as Storm and Hadoop and it must be as efficient as the integrated approaches that process mixed workloads on a single machine. AIM was specifically designed for the needs of the telecommunication industry and it is a core building block in a number of Huawei products. However, we are convinced that its general ideas apply to many verticals.

Figure 1 shows the architecture of AIM, which is composed of three layers: (a) an event stream processing layer (ESP); (b) a storage layer implemented as a key-value store; and (c) a SQL query processing layer (RTA). Each of these layers scales separately as the processing requirements increase. For instance, if the event load increases (more CDRs per second), machines are added to the event stream processing layer.

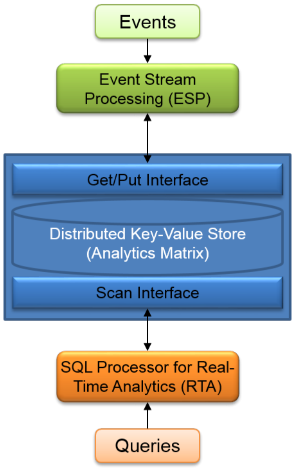

Events
Event Stream Processing (ESP)
Get/Put Interface
Distributed Key-Value Store (Analytics Matrix)
Scan Interface
SQL Processor for Real-Time Analytics (RTA)
Queries

Figure 1: AIM Architecture

While the architecture of Figure 1 has a number of crucial advantages, it also imposes a number of new challenges. One particular challenge is to implement the storage layer in such a way that it can sustain both, the read/update (Get/Put) workload of the event stream processing system and at the same time the bulk read workload of the real-time analytics. AIM addresses this challenge by combining a number of techniques that were recently designed for main-memory database systems: (a) shared scans in order to sustain the high workload [34]; (b) deterministic scheduling of read and write operations in order to control consistency [33, 18, 20, 30]; and (c) a novel columnar storage layout that is based on PAX in order to meet the response time goals of analytic queries [1]. Furthermore, AIM natively supports a data structure called *Analytics Matrix*, which is a materialized view on the CDR events and pre-computes hundreds of indicators. The *Analytics Matrix* is maintained and updated for each event by the event stream processing system.

In summary, the key contributions of this paper are the following:

- An industrial use case for a special type of hybrid workload processing in a Main-Memory Database (MMDB). This use case is not handled well by state-of-the-art systems and motivated the design of AIM.
- A novel architecture and a set of MMDB techniques to process mixed workloads efficiently, thereby meeting strict response time goals.
- The results of comprehensive performance experiments that demonstrate the scalability and efficiency of the proposed techniques.

The remainder of this paper is organized as follows: Section 2 gives an overview of the AIM system and the use case that motivated its design. Sections 3 and 4 detail different design options for its implementation. Section 5 reports on results of a comprehensive experimental evaluation of AIM. Section 6 discusses related work. Section 7 contains conclusions and possible avenues for future work.

2. THE SYSTEM AND ITS DESIGN GOAL

In the following, we give an overview of the AIM system, by means of outlining a running example. The example is a simplified version of the actual use case that is used to illustrate the system functionality. We describe the main components, *Analytics Matrix*, *ESP subsystem*, and *RTA subsystem*, by means of this example. Finally, we formulate the design goal of AIM which is to minimize *Total Cost of Ownership* (TCO) for specific throughput, latency and data freshness requirements. We will focus on the use case that motivated this work and to which we refer as the *Huawei use case* [32] as it comes from a prominent Huawei customer.

2.1 Analytics Matrix

Traditionally, billing data is stored off-line in the data warehouse of a mobile phone operator and is used to implement marketing campaigns, such as offering discounts or new calling plans to loyal customers. The goal is to make this analysis more flexible so that customers can benefit immediately from such marketing campaigns.

Typical marketing campaigns and analytical queries do not depend on single events (caused by phone calls, messages, or network requests), but on calculating indicators per *Entity* (i.e. per subscriber or per cell). All the indicators of an Entity (often referred to as *maintained aggregates* in the literature) are held in an *Entity Record* which is part of a huge materialized view that we call the *Analytics Matrix*. A simplified example of an Analytics Matrix that focuses on subscribers and phone calls is depicted in Table 1. It contains Entity Records for three different subscribers.

subscr-ID	no. calls today	total duration today	total cost today	...
134525	3	1921 secs	$7.50	...
585210	10	3201 secs	$13.80	...
346732	0	0 secs	$0.00	...

Table 1: Example Analytics Matrix

The Analytics Matrix can be very wide and several hundreds of attributes (columns). In a typical large installation, the number of rows in the Analytics Matrix can reach 80M and the number of columns can reach 2000. Obviously, the number of rows changes as subscribers are added or removed, but this is a rather slow change. Some of these attributes are indicators that are part of the Cartesian product of the set of event properties (e.g. cost, duration, local/long-distance call, preferred number), the set of aggregation functions (count, sum, avg, min, max), and the set of aggregation windows.Aggregation windows are either tumbling (today, this week, this month, ...), sliding (in the last 24 hours, in the last seven days, in the last 30 days, ...), or event-based (since the last 10 events, 100 events, 1.000 events, ...). Other attributes are used for ad-hoc segmentation and profiling of the subscribers population. These segmentation attributes are updated in a different manner than the event-driven indicators, but for the purpose of our discussion, this distinction is not significant. In some of the installations, the Analytics Matrix includes dimension attributes. These attributes are either in normalized form, i.e. they are pointers to other tables or denormalized, which means they inline dimension records, thereby eliminating join-processing. In addition, typically, there is a requirement to add (and remove) attributes in the Analytics Matrix in a flexible manner in order to support ad-hoc campaigns, but we will not discuss this aspect further in this paper. We assume that the initial schema of the Analytics Matrix is known at creation time, e.g. stored in a file or meta-database.

In some countries, the telecommunications market is strongly regulated and it is prohibited to gather statistics about individual subscribers. In that case, AIM employs subscriber groups (based on various attributes such as contract-type, age, gender) as master-entities, which is a sufficient measure of anonymization.

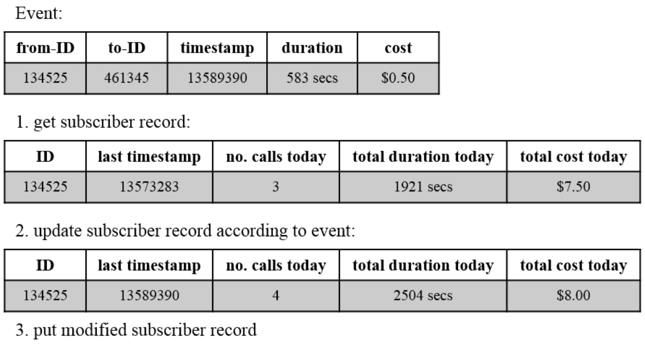
2.2 Event Stream Processing Subsystem

The first AIM subsystem we describe is called Event Stream Processing (ESP). Its responsibility is to receive events from a variety of sources and devices, update the Analytics Matrix according to the aggregation logic and evaluate Business Rules against the updated Entity Record (and the event itself).

```
1: function UPDATE_MATRIX(Event $e$)
2:     UUID $id \leftarrow$ get_id($e$)
3:     EntityRecord $r \leftarrow get(id)$
4:     for all AttributeGroup $attr\_group \in r$ do
5:         $attr\_group \leftarrow \text{update}_{attr\_group}(e, attr\_group)$
6:     $put(id, r)$
7:     return $r$
```

Algorithm 1: Updating Indicators in the Analytics Matrix

Algorithm 1 shows the pseudo code for updating the Analytics Matrix. We denote the function that updates a certain attribute group as $\text{update}_{attr\_group}$. Attribute groups are usually small and contain interdependent attributes, as for example count, sum and average of the same metric, or, if we have to maintain a minimum or maximum value of a specific metric over a sliding window, the set of top-N values with their timestamps.[1] In order for the indicators to be consistent, steps 3 to 6 in Algorithm 1 must happen atomically. This means that between the Get (step 3) and the Put (step 6) operation of a record, no other process is allowed to modify this record in the Analytics Matrix. An example execution of the algorithm is illustrated in Figure 2.



Event:

from-ID	to-ID	timestamp	duration	cost
134525	461345	13589390	583 secs	$0.50

1. get subscriber record:

ID	last timestamp	no. calls today	total duration today	total cost today
134525	13573283	3	1921 secs	$7.50

2. update subscriber record according to event:

ID	last timestamp	no. calls today	total duration today	total cost today
134525	13589390	4	2504 secs	$8.00

3. put modified subscriber record

Figure 2: Illustration of an Update of an Entity Record

In essence, the Analytics Matrix is updated only by *Single-Row Transactions* (as defined in Google's Big Table [9]). This includes the updates which are not driven by event processing (e.g., the updates to the segmentation attributes).

The second important functionality of ESP is Business Rule evaluation. This evaluation must happen in real-time, which means that each rule has to be evaluated against each new event and the resulting updated Entity Record. Business Rules in a telecommunications billing system are mainly used for marketing campaigns (rule 1), but could also trigger alerts, e.g. for potential phone misuse (rule 2) as shown in Table 2. Actions not only include messages being sent back to customers or to maintenance, but could also trigger new events being fed back into the system or modify *Entity Records* (including segmentation attributes). In order to prevent the system from being flooded with rules that continuously trigger once their condition is met, a *Firing Policy* is defined for each rule. This policy defines how many times a rule can trigger within a certain (tumbling or sliding) time window (see [13] for details).

nr	rule	action
1	number-of-calls-today > 20 AND total-cost-today > $100 AND event.duration > 300 secs	inform subscriber that the next 10 phone minutes today will be for free
2	number-of-calls-today > 30 AND (total-duration-today / number-of-calls-today) < 10 secs	advise subscriber to activate the screen lock as it appears that his smart phone is making calls on its own

Table 2: Example Business Rules

[1] The attentive reader might notice that it can happen that all top-N values become invalid as they fall off the window, which means that we have to query the archive of recent events in order to find the minimum/maximum value of the current window. This archive is part of AIM's production version and is not further described in this paper.

A straight-forward method for Business Rule evaluation is shown in Algorithm 2. The method takes as input an up-to-date Entity Record (as produced by the UPDATE_MATRIX function) as well as the event itself and checks them against all rules. Algorithm 2 assumes that rules are in disjunctive normal form (DNF) and are therefore encoded as a list of conjuncts, each of which contains a list of predicates. Algorithm 2 features early abort and early success: (a) whenever a predicate evaluates to false, the whole conjunct evaluates to false and hence we can continue with the next conjunct (steps 7 to 9), and (b) whenever a conjunct evaluates to true, the whole rule evaluates to true and hence we can continue evaluating the next rule in the rule set (steps 10 to 12). Note that Algorithm 2 can still be optimized under certain conditions as we will discuss in Section 4.4.

```
 1: function EVALUATE_RULES(Event $e$, EntityRecord $r$)
 2:     Rule-Set $result \leftarrow \emptyset$
 3:     for all Rule $rule$ : $rules$ do
 4:         for all Conjunct $c$: $rule$.conjuncts() do
 5:             boolean $matching \leftarrow$ true
 6:             for all Predicate $p$: $c$.predicates() do
 7:                 if $p$.evaluate($e, r$) = false then
 8:                     $matching \leftarrow$ false
 9:                     $break$
10:             if $matching$ = true then
11:                 $result \leftarrow result \cup rule$
12:                 $break$
13:     return $result$
```

Algorithm 2: Straight-Forward Business Rule Evaluation

nr	query

1	SELECT SUM(total-cost-today)
	FROM Analytics-Matrix
	WHERE number-of-calls-today > 2
	AND total-duration-today > 600 secs;

2	SELECT SUM(total-duration-this-week),
	AVG(total-duration-this-week)
	FROM Analytics-Matrix am, Subscriber s, Region r
	WHERE am.id = s.id AND c.region = r.id
	GROUP BY r.name;

Table 3: Example RTA Queries

2.3 Real-Time Analytics Processing

The other AIM subsystem is called Real-Time Analytics (RTA) Query Processing. The queries processed by that subsystem are used to answer business intelligence questions (also referred to as decision support). Most of these questions are ad-hoc, which means that they are unpredictable and can involve any subset of Analytics Matrix attributes. In addition to these ad-hoc queries, there can also exist parameterized SQL-like stored procedures[2], but they are considered only a small portion of the workload, which makes the use of indexes or specialized materialized views for these queries (as extensively used in Spark [37] and DBToaster [24]) prohibitive. Some examples of RTA queries are shown in Table 3. The queries typically involve many Entity Records in the Analytics Matrix that must be filtered and aggregated based on some business criteria. RTA queries might also trigger joins with additional small tables which are not part of the Analytics Matrix itself and to which we refer as *Dimension Tables*. An example of such a join query is the second query in Table 3.

2.4 KPIs and Design Goals

After having described the main AIM components, we are able to state a set of *Key Performance Indicators* (KPIs) that determine the way how AIM has to be implemented. We identify the following KPIs:

Maximum Event Processing Time (t_{ESP}): upper bound on how much time the system can take to process an event and evaluate the entire rule set for the updated Entity Record

Minimum Event Processing Rate (f_{ESP}): lower bound on how many events the system must process per Entity per hour

Maximum RTA Query Response Time (t_{RTA}): upper bound on how much time the system can take to answer a RTA query

Minimum RTA Query Rate (f_{RTA}): lower bound on how many RTA queries the system must answer per second

Freshness (t_{fresh}): upper bound on the time that it takes from the moment an event enters the system until the time when the affected Entity Record is visible to RTA queries

Having defined all these KPIs, the goal that AIM is designed to achieve can be formulated as: *Given a set of attributes to maintain, a set of rules to evaluate and an expected event arrival rate, perform event processing as well as ad-hoc analytical query processing in a way that the given KPIs are satisfied and the number of computing resources per Entity minimized.* This means that instead of optimizing for a particular throughput or response time, we assume that an AIM implementation has to guarantee a certain service quality, but within these bounds should minimize the numbers of machines needed and therefore minimize the incurred TCO.

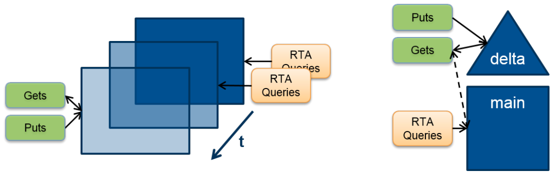
3. COMBINING STREAM PROCESSING AND ANALYTICS

As we have seen in the Introduction, the Data Warehousing approach [23] works well as long as the data in the warehouse can be refreshed every couple of minutes or hours. However, what we want to achieve with AIM is analytical query processing on "real-time" data, i.e. data not older than e.g. one second. Moreover, our design goal is to minimize TCO, and maintaining two separate stores strongly contradicts this goal. As already mentioned, our proposed architecture that employs a single shared store faces several challenges that call for subtle design decisions. This section further details some of these challenges and illustrates different options to address them, thereby drawing the *Design Space* for implementations of the AIM architecture. We do not claim that this enumeration of options for different dimensions is complete, neither is there enough space to (experimentally) contrast pros and cons of each and every option. Nevertheless, we try to argue why in some cases we preferred one option over another for our AIM implementation.

[2] Such procedures could also be used to implement more comples business rules that take into account many *Entity Records* or might even carry out complex iterative algorithms by using the segmentation attributes to store intermediate results.

3.1 Separate Updates from Query Processing

As we have a single store shared by ESP and RTA, we need to solve the problem how to process updates (Puts) in a way that they do not interfere with longer-running analytical queries. Recent research has proposed two different solutions to this problem, both of which are shown in Figure 3.



(a) Copy-on-write (b) Differential Updates

Figure 3: Separating Updates from Query Processing

Copy-on-write, also referred to as *lazy-copying*, is the mechanism employed by most modern operating systems to efficiently manage the initially common memory state of parent and child process after a `fork` system call. Systems like *HyPer* [21] use this OS mechanism to manage different snapshots of their database. While Puts are processed by the parent process on the most current version of the data, analytical query processing happens in the child process(es) on older snapshots.

Differential Updates is a mechanism proposed by Krueger et al. [25]. Their idea is to accumulate all incoming Puts in one data structure (called *delta*) and to process analytical queries in a separated structure (*main*). Periodically, the delta records are applied to the main, which is referred to as *merge*. If response time for Put is critical, we can maintain two deltas, one for new Puts and one for records currently being merged, and atomically switch their reference at the starting point of a *merge*. This approach also guarantees snapshot isolation for the analytical queries as they work on a slightly outdated, but consistent version of the data.

Our current AIM system employs a slightly modified *Differential Updates* technique[3] instead of *Copy-on-write*, the rationale for this being that the KPIs on ESP are so rigorous that the overhead caused by page faults in *Copy-on-write*is unacceptable on vanilla Debian-based Linux.

3.2 Thread Model

As stated in the introduction, our architecture features a distributed key-value store, which means that it must support Get and Put functionality, i.e. Single-Row lookups and updates. In addition to that, we expect our store to support a fast data scan in order to achieve reasonable throughput and response time for RTA processing. This raises the question of how to best utilize the available CPUs. We identify two options: (a) process RTA queries in a multi-threaded way, i.e. employ a separate scan thread for each incoming query and possibly use a thread pool for recycling, and (b) partition the data, thereby assigning one scan thread for each partition. Incoming queries are batched and then processed by all scan threads in parallel, each of them performing a *Shared Scan*. As the second approach gives us extremely high throughput and guaranteed latency(as shown in [14]), this is the approach we chose for AIM.

An alternative to a fixed thread-partition assignment is to partition the data into many small chunks at the start of a Scan and then continuously assign chunks to idle threads until every chunk is processed. This is a simple load-balancing mechanism that overcomes the problem that partitions could become imbalanced. This approach, also known as *work stealing* [7], comes at the additional cost of chunk management.

3.3 Architecture Layering

We consider different options to physically place the three architecture components shown in Figure 1. Although logically separated, it is an option to place ESP and RTA functionality and a storage partition on the same physical node. This approach, to which we refer as the *Fully Integrated Approach*, has the advantage of fast data access through local memory. However, we lose the advantage of the clear separation between storage and processing, which is flexibility. The *Fully Separated Approach* (three separated layers) is more flexible in the sense that it allows to provision resources in a more fine-grained manner; e.g. if we need faster storage access, we just add nodes to the storage layer, leaving the ESP and RTA processing layers unchanged. Obviously, there is a wide range of hybrid models that lie in between fully integrated and fully separated architecture layering. Our AIM implementation actually follows such a hybrid approach (further described in Section 4.2) in order to get as close as possible to the optimization goal stated in Section 2.4.

3.4 Data Placement and Join Processing

While the Analytics Matrix is distributed among the different storage nodes, the question where to store and maintain the rest of the AIM data structures remains. It makes sense to place the ESP Business Rules on the node(s) where the rule evaluation happens, which means to replicate the rule set in several places. Another question is where to place the Dimension Tables and it is closely related to the question where and how to do the join processing. Executing joins in the storage layer is fast as it happens closer to the data while executing joins in a separate processing layer allows for more flexibility in the overall design and is preferable if the storage nodes become overloaded. As the Dimension Tables are relatively small and static, we can replicate them at each storage node and inline the dimension records into the Entity Records. This is the approach that we used in the AIM implementation reported in this paper.

4. AIM IMPLEMENTATION

We implemented an AIM system for the KPIs shown in Table 4. The system is expected to scale well for a number of Entities between 10 and 100 million. Based on the Huawei use case (see Section 2), we developed, and tested our system on, a comprehensive benchmark which is described in more detail in Section 5.

4.1 Observations

Let us start with some general observations about the AIM architecture: (a) the OLTP workload (generated by the event stream) consists of Single-Row Transactions always referring to the primary key (entity-id), which simplifies finding the location of the corresponding record, (b) the Analytics Matrix

[3] In contrast to the original proposition in [25], our delta and main data structures are indexed, which allows a more efficient merge step as no sorting is required. Our merge step is further simplified by omitting dictionary compression (which would only be beneficial if we would have strings in our AnalyticsMatrix.)

uses the same primary key and can hence easily be horizontally partitioned in a transparent way, (c) RTA queries are read-only and can therefore be executed on a read-only snapshot of the Analytics Matrix, (d) the set of Business Rules and the Dimension Tables are small relative to the Analytics Matrix and are updated infrequently, which is why they can be replicated with minimal overhead.

t_{ESP}:	10 msecs	f_{ESP}:	3.6
t_{RTA}:	100 msecs	f_{RTA}:	100
t_{fresh}:	1 sec		

Table 4: SLAs for AIM Implementation

4.2 System Architecture

The 3-tier architecture of our AIM implementation is depicted in Figure 4. It can be seen as a client-server architecture where the storage component acts as a server, and the RTA and ESP component nodes are clients.

We decided to use a dedicated storage layer to store our data structures. As such, it hosts the Analytics Matrix and the Dimension Tables. Note that the Analytics Matrix is distributed (i.e. horizontally partitioned by entity-id) among all storage nodes, while Dimension Tables are replicated at each node. Partitioning the Analytics Matrix was beneficial because we wanted to speed up RTA query processing by scanning the Analytics Matrix in parallel on different nodes. However, as we wanted to reduce communication cost between server and clients, we opted for replicating dimension data at each storage node which allows to perform joins locally. This is valid because the Dimension Tables are assumed relatively static as mentioned above.

At the bottom of Figure 4, we have the RTA nodes that are in fact lightweight processing nodes that take a query, redirect it to all storage nodes and, later on, merge the partial results before delivering the final result to the end user. As the bigger part of the RTA query processing happens on the storage nodes anyway, we need far fewer RTA nodes than storage nodes. In addition, as RTA nodes are stateless, they can scale independently on demand.

Above the storage nodes, we can see the ESP processing nodes. In contrast to the lightweight RTA processing nodes, they have a much bigger CPU load as they process events and evaluate Business Rules and use the storage nodes only for getting and putting Entity Records. Each ESP node has a copy of the entire set of Business Rules and can optionally use a rule index in order to make evaluation faster. Like RTA nodes, ESP nodes are essentially stateless and can hence scale independently on demand.

Communication between ESP and storage nodes happens synchronously (using the Get/Put interface), while communication between RTA and the storage is asynchronous (answers are sent whenever they are available). In order to be able test different installations we implemented AIM for two different communication protocols: TCP/IP and Infiniband [19]. As Infiniband provides very low network latency, we prefer Infiniband wherever possible.

The fact that the logical design of our implementation has 3 tiers does not imply that the physical design has 3 tiers as well. In fact, we tested two configurations for the ESP/Storage nodes layout and their interaction: (a) separate physical tiers and communication using RDMA over Infiniband and (b) placement at the same physical machine (on different cores) and communication through local memory. While (a) is very beneficial in terms of flexibility of the whole system, (b) helps to tweak the system for the last bit of performance because instead of sending relatively large Entity Records (3 KB) over the network, we can simply send the considerably smaller Events (64 B). Consequently, performance results for option (b) were slightly better and we chose this option for our main performance evaluation.

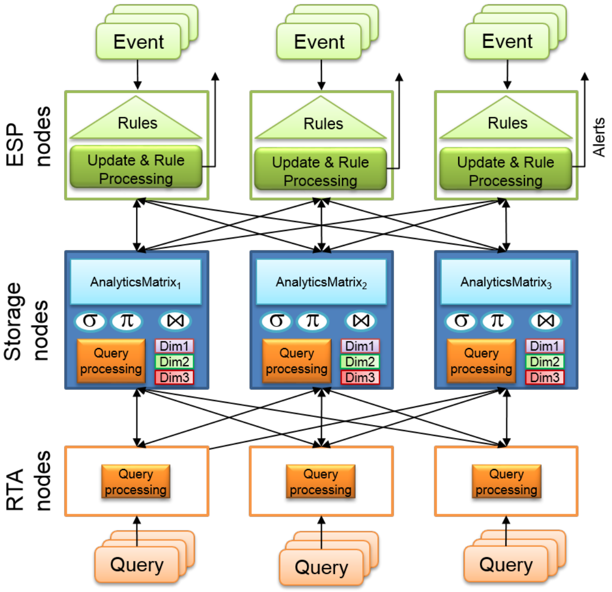


Figure 4: Logical 3-tier Architecture of AIM Prototype

4.3 Updating the Analytics Matrix

Recalling Algorithm 1, we know that each attribute group of the Analytics Matrix has its own, customized C++ update function. Whenever we add a new attribute group to the Analytics Matrix, we construct its update function by combining different (templated) building blocks from a small kernel. This kernel consists of code for different event extraction functions (which allow us to extract certain event attributes like timestamp, cost, or duration of a phone call), different aggregation functions (min, max, sum, count) for different data types (integer, long, float, double) and the different window semantics enumerated in Section 2.1. This essentially means that creating a new attribute group (which happens rarely) is a bit expensive because we have to construct its update function with a huge nested switch-statement. However, the resulting function (which will be called many times, essentially once per event) is highly efficient because (a) it is free of conditional jumps, and (b) it is called by simply dereferencing a function pointer, which again prevents the CPU pipeline from stalling because of branch misprediction.

4.4 Business Rule Evaluation

As the set of Business Rules is relatively static, it makes sense to consider indexing them in order to make rule evaluation fast. We therefore implemented a rule index based on the ideas of Fabre et al. [11]. Interestingly, it turned out that for the 300 Business Rules that we have in our benchmark, this index is no faster than just processing rules without index in a straight-forward manner as shown in Algorithm 2. A micro-benchmark in which we varied the number of rules, showed that using a rule index started paying off for a rule set size of about 1000 and above [13, p. 26]. We conclude that, as long as the rule set is small, we can avoid using an index at all, thereby eliminating the complexity of maintaining such an index.

4.5 ColumnMap

As mentioned in the introduction, the Analytics Matrix is implemented as a distributed in-memory key value store. Early experiments showed that for achieving the KPIs of ESP, *RAMCloud* [28] works well as a key value store [13]. RAMCloud not only provides fast record lookups and writes, but also supports durability and fault tolerance as it follows a log-structured design. However, just as with any row store, we cannot get to a fast enough Scan speed for RTA query processing. In order to allow for a high-speed Scan, traditional analytical query processing engines use a column-oriented storage layout, which in turn is not particularly well-suited for high update rates because of poor locality of the attributes of a record.

Our solution to overcome this dilemma, is to use the *Partition Attributes Across* (PAX) approach [1]. This is an attempt to find the sweet spot between purely row-oriented and purely column-oriented storage layouts. The original idea of PAX was to group records into chunks that fit into a memory page and within a page store records column-wise, i.e. values of a particular attribute are grouped together. Analytical queries that usually process a small subset of the attributes can then profit from data locality as well as from the fact that the entire records of a chunk are present in memory at the same time. We designed *ColumnMap*, a data structure that follows this design with the difference that it is optimized for cache size rather than the size of memory pages. This makes sense as all data structures in AIM are held in memory anyway.

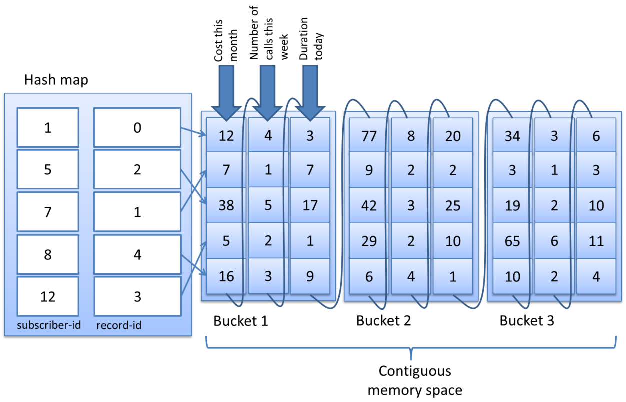


Figure 5: Example of a ColumnMap

The basic structure of ColumnMap is depicted in Figure 5. We group a fixed number of records into logical blocks called *Buckets*. In our current implementation, the default number of records per Bucket (Bucket Size) is 3072[4]. Within a Bucket, data is organized into columns. Each column holds the values for a particular attribute (e.g. cost_this_month). This approach allows us to increase columnar locality, which is beneficial for scan processing of individual attributes (see Section 4.7). In addition to the Buckets, ColumnMap features a small hash map that keeps track of the mapping between entity-id and record-id. The reason for this level of indirection is the fact that entity-ids can be arbitrary application-dependent identifiers, while the record-ids are contiguous numbers starting from 0. Since the 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.

It is worth mentioning that we can also configure ColumnMap to work as a pure row store (by setting the Bucket Size to one) or as pure column store (Bucket Size = total number of Entity Records). In fact, ColumnMap only outperforms a column store with respect to its update performance when records are small enough to fit into a cache line. If they do not (as in our use case where we deal with huge 3 KB records), Bucket Size does not have such a huge impact[5], neither on RTA nor on ESP performance (see Section 5.2) and we could as well use a pure column-store. There are two reasons why we prefer ColumnMap over using an off-the-shelf column store in our system: (a) it has the tunable parameter Bucket Size, which makes it a row, column and hybrid store at the same time and therefore enhances flexibility, and (b) we have direct access to the raw data in the store without the need of going through a SQL interface[6].

4.6 Keeping Track of Updates

As stated in Section 3.1, we have to ensure that updates (Puts) produced by ESP do not interfere with RTA queries because these queries should return a consistent result and therefore work on a consistent snapshot of the Analytics Matrix. In order to solve this, we implemented a variant of Differential Updates [25].

As we cannot afford to block the ESP subsystem at any time (e.g. during the merge-phase), we have to allocate the new delta right before merging, which means that we have two deltas during the merge phase[7]. Get and Put operations have to be adapted accordingly, as shown in Algorithms 3 and 4. Note that these algorithms test whether variable *new-delta* exists in order to determine if there is currently a merge being performed (*new-delta* exists) or not (*new-delta* does not exist). As the algorithms are not thread-safe, we perform Gets and Puts of a specific entity by one dedicated ESP thread. This decision also allows us to update Entity Records atomically by employing *conditional-write*[8], which is an important functional requirement (see Section 2.2).

It is worth mentioning that AIM favors hot spot entities, as this means that corresponding Entity Records might be over-

[4] We chose 3072 because this is the highest power of two such that a Bucket, consequently having size 3072 × record size (3 KB) $\simeq$ 9 MB, fits into the 10 MB L3 cache of our hardware.

[5] However, Bucket Size must be large enough to ensure buckets saturate the SIMD registers, see section 4.7.1.

[6] While there are some mentionable exceptions like Google Supersonic [16], most available off-the-shelf column stores do not expose their internal data structures.

[7] In fact, as mentioned in Section 3.1, we pre-allocate two deltas at startup and always use one of them as new and the other as old delta. This means that new delta allocation effectively becomes swapping the two delta pointers and reseting the new delta, which saves us a lot of memory management and is blazingly fast. However, this delta-switching process has to block the ESP thread for a short moment in order to make sure it will deliver correct results (Algorithm 3), resp. write to the right delta (Algorithm 4). This can be implemented very efficiently with two atomic flags. We point the interested reader to Appendix A for further details.

[8] Each *Get* returns a record that comes along with a timestamp indicating when this record was last modified. The conditional-write function takes this timestamp as an additional argument and only overrides a record if its timestamps matches this argument. If there is no match, an error is returned, which triggers the ESP node to restart the *Single-Row transaction* for the current event.

ridden several times in the *delta* and therefore automatically "compacted" before being written to the *main*.

```
function GET(UUID $id$)
    AMRecord $result$ ←NULL
    if ∃ $new\text{-}delta$ then
        $result$ ← get($new\text{-}delta$, $id$)
    if $result$ = NULL then $result$ ← get($delta$, $id$)
    if $result$ = NULL then $result$ ← get($main$, $id$)
    return $result$
```

Algorithm 3: Analytics Matrix Get

```
function PUT(EntityRecord $r$, EntityID $id$)
    if ∃ $new\text{-}delta$ then
        put($new\text{-}delta$, $r$, $id$)
    else
        put($delta$, $r$, $id$)
```

Algorithm 4: Analytics Matrix Put

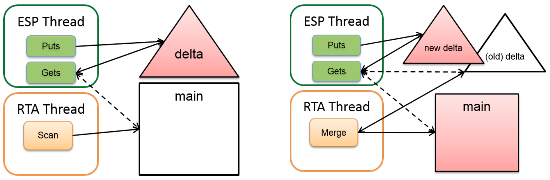
As the *delta* should be optimized for Single-Row Transactions, we implemented it using Google's dense hash map [15]. Additionally, the *main* must feature a fast Scan and needs to be indexed in order for the Single-Row Transactions to work at considerable speed. The index on the primary key (entity-id) is also a requirement for an efficient implementation of the merge-step where we want to be able to do a single pass through the *delta*, lookup the positions of the corresponding records in the *main* and simply replace them. We implemented the *main* as ColumnMap, which is in our case an optimal fit as explained in Section 4.5. There remains the question when and how often we should perform a merge step. In order to prevent the delta structure from growing too large, it would be beneficial to merge as often as possible. On the other, merge steps interrupt RTA query processing and therefore the right moment for merging has to be chosen carefully. Luckily, the merge step can be interleaved optimally with query processing as we show next.

4.7 Query Processing

Traditional database systems process one query at a time. Inspired by *SharedDB* [14], we try to achieve a higher throughput by using a batch-oriented processing technique instead. Our storage server keeps a queue of queries that were submitted by the RTA clients. Once a new Scan is started, all queries currently in the queue are processed together in one single scan pass. Such a *Shared Scan* reduces queuing time for individual queries and at the same time increases query throughput.[9] Moreover, the batch-oriented query execution model nicely fits our delta-main storage layout because scan and merge steps can be interleaved. An RTA query processing thread therefore works in a loop with the following two steps as illustrated in Figure 6:

scan step scan the entire *main* (ColumnMap) as shown in Algorithm 5. During that phase the *main* is read-only and therefore concurrent accesses by the ESP thread (performing Gets) and the RTA thread are safe.

merge step the RTA thread scans the *delta* and applies the updates to the *main* in-place. At the beginning of this step, the *delta* becomes read-only as Puts are redirected to the newly allocated delta (see Section 4.6). The ESP thread never reads an item that the RTA thread is currently replacing, simply because if an item is currently updated in the *main*, this means it must also exist in the *delta*, which implies the ESP thread will get it from there and not from the *main* (see Algorithm 3).



(a) scan step (b) merge step

Figure 6: Collaboration of ESP and RTA Thread

```
function SHARED_SCAN
    for all Bucket $bucket$ : $column\text{-}map$ do
        for all Query $query$ : $current\text{-}query\text{-}batch$ do
            process_bucket($bucket$, $query$)
```

Algorithm 5: Shared Scan Query Processing

4.7.1 Single instruction multiple data (SIMD)

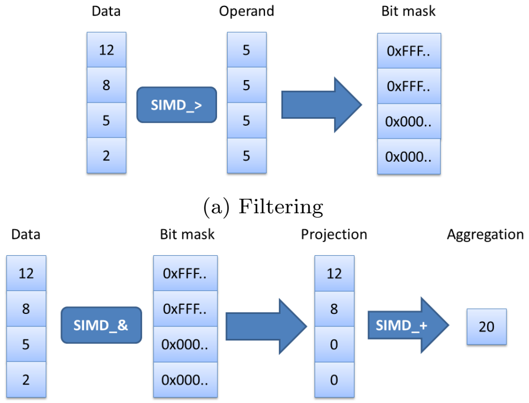
Many current processors feature explicit *single-instruction multiple data (SIMD)* machinery such as vector registers and specialized instructions to manipulate data stored in these registers. They allow for one instruction to be performed on multiple data points in parallel. Intel's *Streaming SIMD Extensions* (SSE) operate on registers of 128-bit or 256-bit width. The size of these registers allows to concatenate up to 4 floating-point operands into a single vector and process arithmetical or logical operations in parallel.

As the research community already pointed out, SIMD instructions allow an additional degree of parallelism and often eliminate some of the conditional branch instructions by reducing branch mispredictions [35, 38]. 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 exploited SIMD instructions to build a fast Scan on *ColumnMap*. This Scan includes filtering (selection) and aggregation (projection) as illustrated in Figure 7:

Filtering: Filtering with SIMD instructions means to first load a column into one vector register and the operand in the other register and then perform a SIMD comparison instruction (e.g. SIMD_<). This results in a bit mask that states whether to include a value in the result (value 0xF..F) or not (value 0x0..0). We combine bit masks from different filters by either SIMD_& or SIMD_| according to the WHERE clause of the query.

Aggregation: We intersect (SIMD_&) the data vector with the bit mask resulting from filtering and then apply an aggregation operator (SIMD_MIN, SIMD_MAX or SIMD_+).

[9] In order to satisfy the real-time KPIs on both, response time and throughput, batch size, as will be shown later, has to be chosen carefully.



(a) Filtering

(b) Projection

Figure 7: SIMD Processing

As becomes clear, such SIMD instructions can only be used in conjunction with a columnar storage layout like ColumnMap. Therefore we have to choose Bucket Size large enough to fill the entire SIMD registers with attribute values. For 256-bit SIMD registers and attribute values of at least one byte, we need at least $256/8 = 32$ records per Bucket.

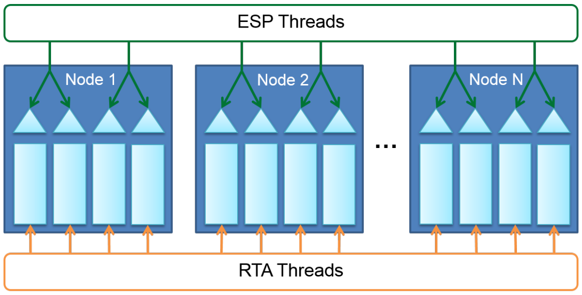


Figure 8: Partitions of Analytics Matrix and Thread Model, $s = 2, k = 2, n = 4$

4.8 Distributed Execution

As explained in Section 3.2, we not only distribute the Analytics Matrix among different nodes, but also partition it further within a node as shown in Figure 8. There are two parameters that determine resource provisioning: number of ESP threads s and number of RTA threads n where n also equals the number of data partitions. Each RTA thread is related to exactly one data partition while an ESP thread works on the delta of several (up to k) partitions. In our implementation we used the simple strategy to first choose s large enough to achieve the SLAs of ESP, and then use the remaining cores for RTA processing and communication. As 2 threads for communication with the other two tiers are used, this means $n = number\text{-}of\text{-}cores - s - 2$. Note that we use the terms *core* and *thread* interchangeably here as we have as many threads as cores. This is to avoid the performance penalty of over-subscription (see Section 5.2). What is more, our implementation is NUMA-aware in the sense that RTA threads are pinned to different cores in such a way that they are collocated with their assigned data partition and can therefore access it through local memory (which is about twice as fast as accessing memory on another NUMA node).

Routing a Get or a Put request to the correct data storage partition works as follows: first, use a global hash function h to route the request to the node with ID $h(key)$. Next, within node i, apply a node-specific hash function $h_i(key)$ to determine the ID of the partition that hosts this key. Finally, route the request to the ESP thread responsible for this partition.

The distribution of data immediately raises the question of consistency. We implemented intra-node consistency by coordinating the start of the scan-step for all RTA threads on a storage node. This is also beneficial because if all threads start at the same time, they can work on the same query batch. It is not necessary to provide inter-node consistency as events do not have a global order. Distributed transactional consistency is a complex task that is subject to other current research [27].

5. EXPERIMENTS

As illustrated in the previous sections, our AIM system addresses the specific Huawei use case, which calls for a specific benchmark. We therefore, in collaboration with our customer, developed a comprehensive benchmark that tests the ability of a system to cope with the requirements of our use case. This benchmark consists of 300 Business Rules, 546 indicators (that means Entity Records of 3 KB) and seven different parameterized RTA queries (as shown in Table 5). Each rule consists of 1 to 10 conjuncts and each of these conjuncts has 1 to 10 predicates. There are two categories of RTA queries. While queries 1 to 3 query the Analytics Matrix only, queries 4 to 7 involve joins with one or several Dimension Tables. For clarity reasons, we omit details about the Dimension Tables and describe Q6 and Q7 only in textual form as the full SQL statements involve complex and nested sub-queries.

The benchmark parameters are number of Entities (i.e. size of the Analytics Matrix), event rate, number of RTA clients c and query-mix. While we try to send events at a fixed rate, we send RTA queries in closed loops. This means that an RTA client only sends a new query after having received and processed all partial results from the previous query. We can therefore increase the RTA load on the system by increasing c. As our use case states that the system must be able to answer ad-hoc queries, the query workload should be unpredictable. We model this unpredictability by disallowing the use of any index on the Analytics Matrix, except for the primary key. Moreover, query parameters are chosen uniformly at random.

5.1 Setup and Methodology

Our experiments were conducted on servers equipped with a dual-socket 4 core Intel Xeon E5-2609 CPU, each core operating at 2.40 GHz. Each server features 32KB L1 cache, 256KB L2 cache and 10MB L3 cache as well as 4x32GB Samsung DDR3-DIMM, resulting in a total of 128GB RAM. We used a vanilla Debian Linux 4.6.3-1 running kernel 3.4.4 and GCC-4.7.2 and communicated over a 40 Gbit QDR Infiniband network.

As illustrated in Section 4.2, we decided to host ESP and storage on the same physical nodes (communicating through local memory) and RTA processing nodes separately. We used one dedicated machine for generating random events and measuring end-to-end throughput and response time of the ESP subsystem. This machine could be configured to send events at a certain rate (as specified by the benchmark). The RTA clients, responsible for the creation of random RTA

Query 1:
SELECT AVG (total_duration_this_week)
FROM AnalyticsMatrix
WHERE number_of_local_calls_this_week > α;

Query 2:
SELECT MAX (most_expensive_call_this_week)
FROM AnalyticsMatrix
WHERE total_number_of_calls_this_week > β;

Query 3:
SELECT (SUM (total_cost_this_week)) /
(SUM (total_duration_this_week)) as cost_ratio
FROM AnalyticsMatrix
GROUP BY number_of_calls_this_week
LIMIT 100;

Query 4:
SELECT city, AVG(number_of_local_calls_this_week),
SUM(total_duration_of_local_calls_this_week)
FROM AnalyticsMatrix, RegionInfo
WHERE number_of_local_calls_this_week > γ
AND total_duration_of_local_calls_this_week > δ
AND AnalyticsMatrix.zip = RegionInfo.zip
GROUP BY city;

Query 5: SELECT region, SUM (total_cost_of_local_calls_this_week) as local, SUM (total_cost_of_long_distance_calls_this_week) as long_distance FROM AnalyticsMatrix a, SubscriptionType t, Category c, RegionInfo r WHERE t.type = t AND c.category = cat AND a.subscription_type = t.id AND a.category = c.id AND a.zip = r.zip GROUP BY region;
Query 6: *report the entity-ids of the records with the longest call this day and this week for local and long distance calls for a specific country cty*
Query 7: *report the entity-ids of the records with the smallest flat rate (cost of calls divided by the duration of calls this week) for a specific value type v*

Table 5: RTA Queries 1 to 7, $\alpha \in [0, 2]$, $\beta \in [2, 5]$, $\gamma \in [2, 10]$, $\delta \in [20, 150]$, $t \in$ SubscriptionTypes, $cat \in$ Categories, $cty \in$ Countries, $v \in$ CellValueTypes

queries and end-to-end measurements of throughput and response time, could be placed on a single node, using c execution threads. As preliminary results revealed, one node for RTA processing was enough to saturate up to 10 storage server nodes and it could therefore be collocated with the RTA clients on one physical machine.

As stated in the beginning of Section 4, the AIM system should be able to cope with an event rate of 3.6 events per Entity per hour and scale from 10M to 100M Entities. We therefore first executed a number of experiments to determine the optimal resource allocation and parameter setting for 10M Entities and 10,000 events per second (Section 5.2). Then, we analyzed how our system behaves when we increase the RTA load (Section 5.3), or add more resources at the storage layer (Section 5.4). Finally, we wanted to demonstrate how provisioning more resources can help to cope with an increasing number of Entities up to 100M (and 100,000 events per second).

In order to rank the measured performance numbers, we compared AIM to two commercial database products: System M, a main-memory-based column store optimized for real-time analytics, and System D, a disk-based row-organized database system with support for fast updates. Even though we implemented `update_matrix(.)` as a stored procedure, both systems clearly failed in handling the required amount of events: System M could handle about 100, System D about 200 events per second. This is why we only measured their (read-only) RTA performance. To achieve the best possible query and update performance for System D, we let it use its index-advisor to create indexes on the relevant columns despite the benchmark forbidding precisely this. In addition to these commercial systems, we also executed the benchmark on *HyPer* [21] which seems to be the best fit for the Huawei usecase among the available research prototypes because it processes analytical queries on consistent data snapshots with configurable freshness. In contrast to systems M and D, *HyPer* could handle a reasonable number of events per second (5,500 in isolation and 1,940 with one RTA client).

All experiments were conducted using a query mix of all seven queries, drawn at random with equal probability. In the following, we report average end-to-end response time and overall query throughput of RTA queries. As the event rate was configured to meet f_{ESP}, we only report measured ESP throughputs that deviated from the event rate. t_{ESP} was always met and is therefore excluded from the results. KPIs are depicted as dotted lines in the performance graphs. We used the following default values for our experiments: 10M Entities, 10,000 events/sec, 8 RTA client threads ($c = 8$), 1 ESP server thread ($s = 1$), 5 RTA server threads $n = 5$ (= number of data partitions), 1 AIM server (storage server). The measurements do not include the costs for checkpointing and logging.

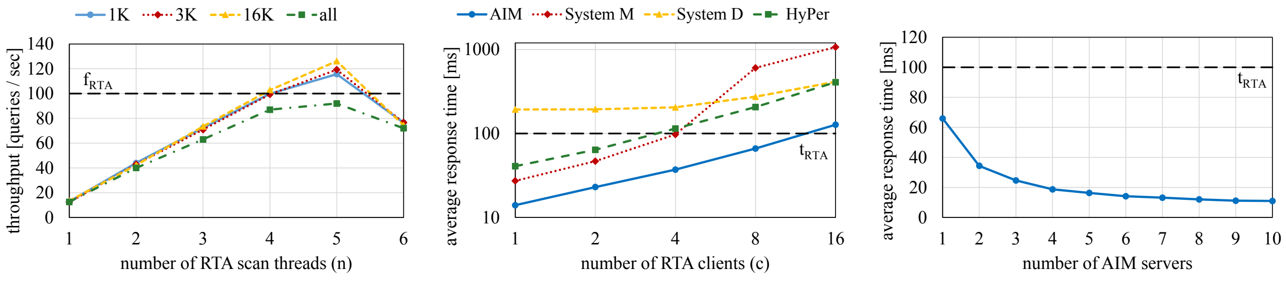
5.2 Optimal Number of Data Partitions

In the first experiment, we tried to find the best number of storage partitions (= RTA server threads) as well as the optimal Bucket Size for the ColumnMaps. Figures 9a and 10a show response time and throughput for different combinations of these two parameters on a single storage server. As hypothesized in Section 4.8, we get optimal performance when allocating exactly as many threads as there are cores. As we have one ESP thread and two communication threads, this means 5 RTA server threads on an 8-core machine. Moreover, we can see that with 4 and 5 partitions, all KPIs are met. For $n = 6$, the ESP throughput was below 10,000 at about 8,000 events/sec (similar for all different Bucket Sizes), which is a direct consequence of the thread thrashing at the storage nodes. As we can see, Bucket Size does not seem to have a major impact on performance as long as it is large enough. Notice that ColumnMap (for different Bucket Sizes) slightly outperforms the pure column store (which is the line denoted by *all*).

5.3 Peak Performance compared to Baselines

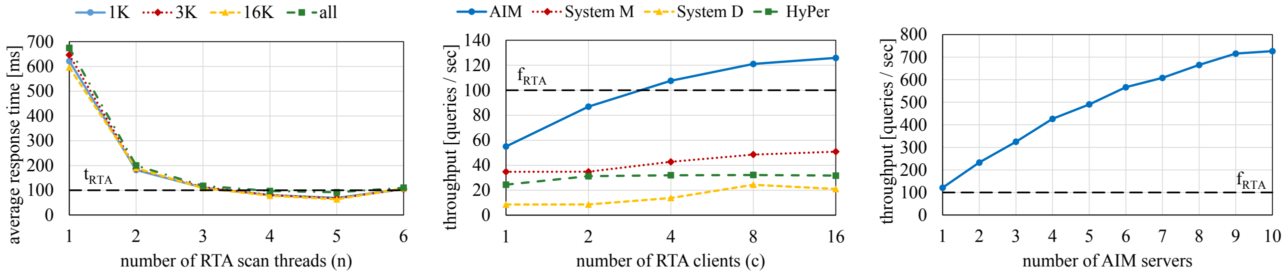
RTA clients work in a closed loop and submit only one query at the time which is why their number is also an upper-bound on the query batch size at the storage server(s). If we want to test the robustness of our system, we can therefore simply increase the RTA load by varying c from 1 to 16 as shown in Figures 9b and 10b. We see that AIM is robust in the sense that once saturation is reached (somewhere between 8 and 16 threads), throughput stays constant but does not drop. Response time increases linearly, but not exponentially, just as we would expect. The fact that we reach peak performance with 8 threads, i.e. satisfy both RTA KPIs ($t_{RTA} < 100$ msecs and $f_{RTA} > 100$ q/sec), suggests to limit query batch size at the storage server to about 8.

We can also see that AIM clearly outperforms all three competitors (system M and D, and *HyPer*) by a factor of at least 2.5 for RTA response time and throughput. Given



(a) vary n from 1 to 6, different Bucket Sizes (b) vary c from 1 to 16 (c) vary AIM servers from 1 to 10

Figure 9: Average Response Time of RTA Queries [msec] for 10M Entities, 10,000 events/sec, default: 1 server, $n = 5$, $c = 8$



(a) vary n from 1 to 6, different Bucket Sizes (b) vary c from 1 to 16 (c) vary AIM servers from 1 to 10

Figure 10: Throughput of RTA Queries [queries/sec] for 10M Entities, 10,000 events/sec, default: 1 server, $n = 5$, $c = 8$

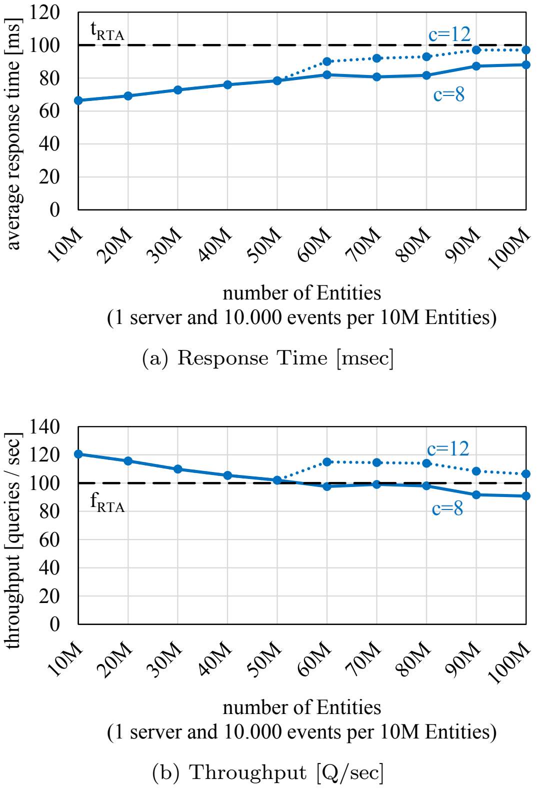
the fact that RTA performance of the competitors was measured in isolation (without concurrent ESP processing), we think that this improvement is quite significant. In addition, as mentioned in section 5.1, AIM achieves an event processing rate improvement ranging from 5x (over *HyPer*) to two orders of magnitude (over systems M and D).

5.4 Scale Out

The previous experiments illustrated that one storage server is enough to accommodate 10M Entities. However, as KPIs might become more stringent, it is important to know whether provisioning more resources would solve the problem. In order to analyze this, we increased the number of storage servers from 1 to 10 as illustrated in Figures 9c, resp. 10c. We see a near linear increase in throughput as well as response time. We conclude that it is possible to scale out with a satisfactorily small overhead.

5.5 Scalability and Total Cost of Ownership

The last experiment concerns scalability, or in other words, how the performance measures change if we not only increase the number of servers, but also the load (number of Entities, event rate) accordingly. In this experiment, for each added server, we also added 10M Entities and 10,000 events per second. Figure 11 shows a decent scalability. Ideally, throughput and response time would be horizontal lines. The fact that they are not illustrates the two types of overhead that additional servers create: (a) synchronization overhead, and (b) result-merging. The more AIM servers, the more partial results have to be collected and be merged at the RTA node. As long as the RTA node is not fully utilized, we can sacrifice some response time to improve throughput scalability by adding more client threads at the RTA node. As we can see from the dotted lines in Figure 11, increasing c from 8 to 12 for 60M Entities and beyond helps us to stay sharply within the KPI limits. If we want to go beyond 100M customers (and 100,000 events/sec), we can still stay within KPI limits, but may have to provision more servers machines per En-



(a) Response Time [msec]

(b) Throughput [Q/sec]

Figure 11: vary AIM servers, data size, and ESP load

tity (see Section 5.4). If the ESP or RTA layer become the bottleneck, they can scale out individually.

In conclusion, the AIM architecture is particularly well-suited for provisioning resources just when and where they are needed (depending on changes in the workload and/or KPIs). In essence, this also means that the total cost of ownership is minimized. In our usecase, the TCO for a system that satisfies the given KPIs for $10 \times n$ millions of Entities (for $1 \leq n \leq 10$), is the operation costs of $n + 2$ of our cluster machines (n AIM storage nodes, 1 ESP node, 1 RTA node) plus the cost of the Infiniband network connecting them.

6. RELATED WORK

There is a vast variety of work in the field, as can be seen on Matthew Aslett's *Data Platforms Landscape Map* [6]. What differentiates AIM from all these systems is the ability to serve a special workload mix of streaming and ad-hoc analytical query processing. While the basic building blocks used in our AIM implementation (Materialized View, Data Partitioning, Shared Scan [34], Differential Updates [25], and SIMD-Processing [35, 38]) are well-known, we claim novelty in the way how we combine them to achieve the particular requirements that AIM is defined for. In the following, we will discuss some of the most relevant related systems, being well aware of the fact that this enumeration is neither extensive nor complete.

On one side of the spectrum, there are the traditional streaming engines, like *Apache Storm* [4], *Esper* [31], and *Microsoft StreamInsight* [3]. These systems are good in handling high event rates and computing some statistics on them. However, the sheer amount of indicators to be kept in AIM (hundreds of indicators to be kept for each Entity) poses a severe challenge on these systems which are typically tailored towards maintaining only dozens of spouts and bolts.

On the other side of the spectrum, there are the fast analytical processing engines, like *SAP-HANA* [12], *C-Store* [29], and *MonetDB* [8]. These systems speed up query execution by organizing the data column-wise, therefore only examining the interesting attributes of a record and profiting from data locality. Again, the number of columns in the Analytics Matrix is a challenge, because an update of an Entity Record would incur 500 random memory accesses. Also the high event rates of AIM pose a severe challenge to such systems.

The concept of the Analytics Matrix is similar to the (Read-Only) WideTable presented in [26]. Our Analytics Matrix can be thought of as such a WideTable that is, however, updated very frequently by the ESP subsystem.

Instead of implementing the Analytics Matrix as a ColumnMap, we could think about using a traditional key-value store like *Google's BigTable* [9], *H-Base* [22], or *RAMCloud* [28]. While these stores can cope easily with the ESP requirements, they lack a fast scan, which makes them ill-suited for ad-hoc RTA processing.

Finally, there are OLTP/OLAP engines that are in some regards similar to AIM. Among them are *SharedDB* [14], *HyPer* [21], *HYRISE* [17], *Spark (Streaming)* [37], and, most recently, Druid [36]. These systems typically make the assumption that most of the analytical queries are known beforehand and make use of this by employing a special storage layout (HYRISE), specialized views (Spark Streaming), or specialized indexes. Ad-hoc queries are supposed to appear rarely and do therefore not have to meet strict latency requirements. The case for AIM is different because ad-hoc queries are the standard rather than the exception. We recently started studying how the Copy-on-write approach can cope with the AIM workload and our preliminary results show that the total cost of ownership is twice to three times higher than for Differential Updates.

7. CONCLUSION AND FUTURE WORK

This paper introduced AIM, a new architecture for addressing systems with stringent requirements on streaming, frequent updates and execution of analytical queries in real-time. We discussed the design space of such an architecture and implemented the AIM system, a distributed and flexible implementation for a specific workload that features a novel combination of well-established principles, such as *Materialized View*, *horizontal Data Partitioning*, *Shared Scan*, the *PAX paradigm*, efficient distributed super-scalar query execution with *SIMD*, and a new variant of *Differential Updates* for real-time data management. Furthermore, we developed a comprehensive benchmark that captures the essential features of the specific workload of the unique Huawei use case. The experimental evaluation of the AIM system with this benchmark showed that we can indeed meet the required KPIs with minimal resources and beat commercial analytical database systems by a factor of 2.5 in analytical and two orders of magnitude in update performance. This minimal resource allocation features two processing nodes and one storage server node per 10M Entities up to 100M Entities and 100,000 events per second. Moreover, we demonstrated that EPS, RTA and storage layer scale linearly and independently from each other, which allows the system to scale far beyond 100M Entities with a minimum of additional resources.

The production version of AIM includes more features that were not presented in this paper for brevity and focus reasons, for example, a persistent event archive, a special PAX-based main-memory layout that includes support for variable length data, incremental checkpointing and zero-copy logging, as well as policy-enforced and self-balancing queuing mechanisms for handling skews in the ESP subsystem, and multi-threaded NUMA-aware main memory real-time analytics. These features are mostly used for making the system available and durable while preserving its performance. Moreover, we have experimented with replacing the delta-main storage with snapshots of the Analytics Matrix managed by the OS' copy-on-write mechanisms. Our first insight to this is that controlling the frequency of the fork allows trading freshness of real-time analytics for better event processing rate, but we will report on this work in a separate future paper.

We would like to continue our work on AIM to make the system able to cover a wider range of OLTP/OLAP workloads with stringent real-time requirements that include transactions, e.g. the CH-Benchmark [10]. We believe that our distributed storage server with its Get/Put/Scan interface can serve as an important building block for more general OLTP/OLAP engines built on top of key-value stores, as for example TELL [27]. The only additional feature needed would be the ability to keep multiple versions of a record, which seems to be pretty straight-forward, as in the case of AIM, making the *delta* multi-versioned seems sufficient. Multi-versioned *deltas* would, in addition, allow us to maintain multiple Analytics because ESP could use atomic transactions to update the involved Entity Records all at once.

Acknowledgments

We are especially grateful for the detailed feedback and encouraging discussions with Marko Vukolic, Yanlei Diao, and Marica Stojanov. In addition, we want to thank the anonymous reviewers of SIGMOD 2014, VLDB 2014, and SIGMOD 2015 for pointing us to missing parts and pieces as well as helping us overcoming weaknesses of the paper. A very special thanks goes to Thomas Neumann who was unbelievably supportive when we implemented the benchmark on *HyPer* and who did not even hesitate to implement entirely new features that we required. Last, but not least, we would like to thank Huawei's Central Software Institute (CSI) for the support of this industrial-academic collaboration the fruits of which where shown in this paper.

8. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180, 2001.

[2] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A Hands-free Adaptive Store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1103–1114. ACM, 2014.

[3] M. Ali. An introduction to microsoft sql server streaminsight. In *Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & Application*, page 66. ACM, 2010.

[4] Apache Foundation. Apache Storm – A system for processing streaming data in real time.

[5] Apache Foundation. Hadoop. `http://hadoop.apache.org/`.

[6] M. Aslett. Data Platforms Landscape Map. `http://blogs.the451group.com/information_management/2014/03/18/updated-data-platforms-landscape-map-february-2014`.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[8] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, volume 5, pages 225–237, 2005.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[10] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, et al. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, page 8. ACM, 2011.

[11] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. In *ACM SIGMOD Record*, volume 30, pages 115–126. ACM, 2001.

[12] F. Färber et al. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1), 2012.

[13] G. Gasparis. AIM: A System for Handling Enormous Workloads under Strict Latency and Scalability Regulations. Master's thesis, Systems Group, Dep. of CS, ETH Zurich, 2013.

[14] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: killing one thousand queries with one stone. *Proceedings of the VLDB Endowment*, 5(6):526–537, 2012.

[15] Google. Sparsehash. `https://code.google.com/p/sparsehash`.

[16] Google. Supersonic Query Engine. `https://code.google.com/p/supersonic`.

[17] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden. HYRISE - A Main Memory Hybrid Storage Engine. *Proceedings of the VLDB Endowment*, 4(2):105–116, 2010.

[18] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 981–992. ACM, 2008.

[19] InfiniBand Trade Association. `http://www.infinibandta.org`.

[20] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[21] A. Kemper and T. Neumann. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[22] A. Khetrapal and V. Ganesh. Hbase and hypertable for large scale distributed storage systems. *Dept. of Computer Science, Purdue University*, 2006.

[23] R. Kimball. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses.* John Wiley, 1996.

[24] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal*, 23(2):253–278, 2014.

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

[26] Y. Li and J. M. Patel. Widetable: An accelerator for analytical data processing. *Proceedings of the VLDB Endowment*, 7(10), 2014.

[27] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the Design and Scalability of Distributed Shared-Memory Databases. Technical report, ETH Zurich, 2013.

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

[29] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[30] M. Stonebraker and A. Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[31] E. Tech. Event Series Intelligence: Esper & NEsper. `http://esper.codehaus.org`.

[32] TELCO-X Network Analytics Technical Questionnaire. Huawei internal document relating to customer TELCO-X, 2012.

[33] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, 2010.

[34] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable Performance for Unpredictable Workloads. *Proceedings of the VLDB Endowment*, 2(1):706–717, 2009.

[35] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.

[36] F. Yang, E. Tschetter, G. Merlino, N. Ray, X. Léauté, D. Ganguli, and H. Singh. Druid: A Real-time Analytical Data Store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 157–168. ACM, 2014.

[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–17, 2010.

[38] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 145–156. ACM, 2002.

APPENDIX

A. THREAD SYNCHRONIZATION

As illustrated in Figure 6, we carefully manage the access to the different data structures (new and old *delta*, and *main*) in such a way that there is always only one thread writing and that while data is read it is guaranteed not to be changed by the other thread. What is left to do is to make sure that new delta allocation by the RTA thread(as described in Section4.6) does not interfere with *Gets* or *Puts* being performed by the ESP thread. We use two atomic flags to achieve this: *rta_ready* indicates that the RTA thread is ready to allocate a new delta or is already doing so; *esp_waiting* is used by the ESP thread to acknowledge the new delta allocation and to signal that the thread waits until this is finished.

```
function RTA_SWITCH_DELTAS(.)
    rta_ready $\leftarrow$ true                          ▷ set intent
    while $\neg$esp_waiting do                        ▷ spin-loop
    ...           ▷ allocate new delta (switch delta pointers)
    esp_waiting $\leftarrow$ false               ▷ ensure ESP progress
    rta_ready $\leftarrow$ false                     ▷ release intent
```

Algorithm 6: New Delta Allocation when merge step starts

```
while running do
    req $\leftarrow$ dequeue_request()
    if rta_ready then                             ▷ check intent
        esp_waiting $\leftarrow$ true
    while rta_ready do                              ▷ spin-loop
    process_request(req)
```

Algorithm 7: ESP Service Loop

Algorithm 6 illustrates how these flags are used by the RTA thread to make sure that new delta allocation only happens once the ESP thread blocks, while Algorithm 7 shows the ESP service loop. Note that both flags are only modified twice per scan/merge operation. This is favorable for reducing the overall performance penalty because the main synchronization costs come from atomic writes (while atomic reads are relatively inexpensive). Another important remark: the ESP thread only blocks while the new delta is allocated (which is a very fast operation as explained in Footnote 7), but not during the merge step (which lasts orders of magnitudes longer).