Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 78

Systems Group, Department of Computer Science, ETH Zurich

AIM: A System for Handling Enormous Workloads under Strict Latency and Scalability Regulations

by

Georgios Gasparis

Supervised by

Prof. Donald Kossmann
Lucas Braun
Thomas Etter
Martin Kaufmann

April 1, 2013

inf | Informatik
Computer Science

Abstract

In recent years a need emerged for executing online processing on huge amounts of data. Companies and organizations keep on aggregating information and attempt to structure their policies according to the statistics they collect. Systems specifications become more and more demanding and time becomes precious. This Master's Thesis proposes a novel idea for developing a system that can efficiently run real-time processing on streams of incoming events. In addition to events, a set of rules, called campaigns, exist in the system. These rules consist of a number of predicates and should be evaluated against the incoming events in minimal time.

We propose a system that is composed of different building blocks, each of which is entitled to a specific operation. Different approaches for these components will be examined in order to achieve maximal performance. We implement such a system on top of Key-Value stores that reside in main memory, because we want to take advantage of the low latency of main memory access. The proposed system aims at managing workloads characterized by extremely high arrival rates by ensuring low latencies and high event rates.

Contents

1 Introduction

1.1 Motivation

With more than 6 billion active cell phone subscriptions worldwide, an abundance of information is generated day by day. Subscriber-generated events, such as telephone calls and network activity, build an ongoing huge stream of data, from which valuable information can be extracted. This information can be used for example, to determine premium customers, or provide optimal contracts to subscribers. More and more telecommunication companies are eager to design and run their service policies on top of the data they acquire from their own customers. Our work was motivated by a company that is interested in associating subscribers' activities to prizes by the means of a rule-based evaluation system. The rules of that system involve campaigns and subscribers' statistics, and include time as an additional dimension. For instance, the company wants to reward people who spend a certain amount of money on phone calls every week, by giving them a discount on their next few calls. The system implementing this functionality must be able to commence its tasks efficiently, and it must be scalable. Moreover, the system must handle high event rates (several thousand events per second), and subscribers should be notified after only a short delay (a few milliseconds). The rigid time constraints on execution and the large time windows for which the data needs to be maintained make the problem even more intriguing.

1.2 System Overview

The system we propose is called Analytics in Motion (AIM). AIM is a complex system responsible for carrying out a number of computational tasks. The AIM system consists of the following subsystems:

- Stream and Event Processing (SEP)
- Real Time Analytics (RTA)

The main goal of the SEP system is to match campaigns to subscribers in real time. To do that, it stores information related to campaigns, subscribers and events. A subscriber is a customer of a telecommunication company who incurs a number of events per day. For the sake of simplicity, we assume that an event corresponds to a telephone call, although other types of events like network activity events exist. A campaign consists of a rule, which involves a condition, and a reward. If the condition is fulfilled, the subscriber earns the reward associated with the campaign. The input of the SEP system are the user-generated events. A number of campaigns (typically several hundred) exist in our system at each point in time. Technically, there are a number of tasks that the SEP system must accomplish:

1. It must compute and maintain statistics, such as "number of calls this week" for each subscriber, based on events.

2. It must trigger a campaign whenever its condition is met.

The RTA system processes simple, ad-hoc real-time analytical queries, based on the statistics maintained by the SEP system. In addition to these statistics, RTA considers dimension data that is loaded via ETL[1] processes (e.g. region information, etc.). The RTA queries are expressed in a SQL-like manner.

As RTA will be built upon the SEP system, it is natural to start with the latter. This is why this thesis focuses on the design and the implementation of the SEP system.

1.3 Requirements

The key requirements that the SEP system must satisfy are the following:

1. Latency: Every event must be processed within 10 milliseconds. That is, subscribers must be notified in real-time whether a campaign matches their activities.

2. Cost (Machine / Subscriber): Each subscriber incurs an average number of events per second (e.g. three telephone calls per day). We would like to minimize the number of machines needed for a given subscriber population. Correspondingly, we would like to maximize the number of subscribers and their events that can be handled by a single machine.

3. Scalability / Elasticity: It should be possible to add (or remove) machines dynamically to support a growing (or shrinking) number of subscribers.

1.4 System Components and Design Space

There are many alternative ways and technology building blocks to carry out the tasks mentioned before. One approach, for instance, is to implement each campaign as a continuous query, the incoming events as a data stream, and then use a data stream management system, such as Esper[1], or Storm[2] as a basis for the SEP system.

Our approach is based on a combination of different building blocks. We refer to it as the "Materialized View"[2] approach, and we will study it in the remainder of this thesis. Figure 1 gives an overview of this approach. Here are its most important design points:

1. There is an "Analytics Matrix" (AM) that records the current value of all statistics for every subscriber. This table changes with every event. We label the set of statistics kept for a specific subscriber as his/her "Profile".

[1] Extract, transform and load process that is mainly used in data warehousing.

[2] We borrow this terminology from the relational database world, as the AM component of our approach imitates the behaviour of a view[3], in the sense that it precalculates operations.

2. There is a "Campaign Index" that keeps track of all the active campaigns. This component is updated for each new campaign and for each existing campaign that expired. The campaign index is probed for each event using the updated subscriber profile, in order to find out which campaigns are relevant for the subscriber.

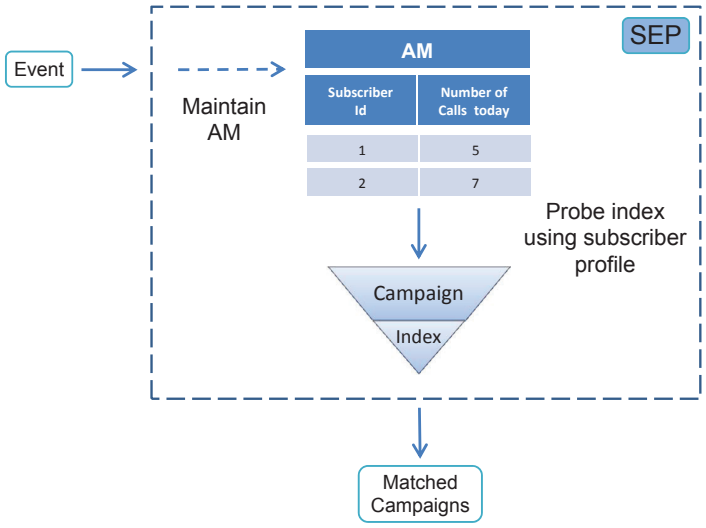

Figure 1: The Materialized View approach

1.5 Related Work

There has been a significant amount of work on use cases very much alike to the one presented in this Master's Thesis. This work is broadly known as either rule condition testing, or subscription matching. The main idea involves a system that must test each newly arrived piece of information against a collection of rules to find those matching. A large number of proposals tailored to different principles try to address the problem of evaluating conditions efficiently. These solutions vary from triggers on relational databases to continuous query systems and main memory matching algorithms.

Databases offer trigger functionality for checking a condition when an event comes in. Scalability is an issue, as upper bounds in the number of triggers exist. General purpose databases do not account for efficient execution of triggers. Much effort has been put into overcoming these barriers and transforming databases from simple storage layers to active systems being able to capture changes [4, 5]. In the use case we consider, conditions are evaluated against aggregated information computed over events. This necessitates an increased

flexibility in storing and updating information that relational database systems do not provide us with. Further aggravating the problem are the time requirements specified in section 1.3, which eventually impose the use of in-memory-storages refraining from I/O bottlenecks.

Data Stream Management Systems have been around for many years. They follow a more active approach than relational databases, monitoring data feeds and reporting to conditions of interest. Some of them [6, 7] are still at experimental level, and therefore cannot be employed, but there are others like Esper[1], or the recently released Storm[2] that seem promising. We consider the evaluation of such systems as future work. Although their performance might indeed be comparable to what we succeeded in, the fact that no access to internal data structures or data is permitted, is a limiting factor for our use case. Specifically, in section 1.2 we stated that the RTA system depends on the tasks that the SEP system carries out[3]. As a consequence, having no data at hand poses an obstacle in the path of RTA system's execution.

Our proposed system employs memory-based algorithms for manipulating the incoming events and guaranteeing efficient rule checking. A lot of main memory algorithms have been published [8, 9]. The vast majority of them emphasizes tree data structures to index rule predicates [10, 11]. The key idea is that traversing the tree will lead us to potentially matching rules. Apparently, this is advantageous, since not all predicates need to be checked. The "Materialized View" approach belongs to the main memory algorithms category, and assists us in being both fast and generous[4].

1.6 Thesis Structure

The remainder of this thesis is structured as follows. Section 2 describes the Analytics Matrix component mentioned in 1.4. Section 3 details the properties and the implementation of campaigns. Section 4 presents our motivation for the campaign index component, the index building procedure and finally the way we probe the index. In section 5 we illustrate the experimental evaluation of the system. Finally, section 6 concludes the thesis and gives an outlook on future work.

[3] Real-time analytical queries run on the statistics we maintain in the AM.

[4] Accessing the AM table is feasible in that case. Therefore, no problems for the RTA system arise.

2 The Analytics Matrix (AM)

In this section we thoroughly examine the Analytics Matrix (AM) component of the "Materialized View" approach. That is, a table containing one record per subscriber that captures the most recent information of his/hers. This table allows us to precompute aggregations, and to alter their values incrementally minimizing expensive computations during execution. Moreover, an AM record is compounded of a large number of attributes that are statistics computed over a fixed set of metrics described below.

2.1 AM Attributes

The described stream and event processing system deals with numerous user-generated events. Each event is composed of a number of attributes such as: caller id, cost and is-long-distance. Based on these attributes, we deduce a meaningful set of metrics on top of which aggregations are computed. The three basic metrics are:

- Call
- Cost
- Duration

The first one signifies that a call took place and it always has value 1, as every event corresponds to one call. Obviously, "Cost" points to the amount of money the caller spent on a call, and finally, "Duration" refers to how long the underline event lasted. Besides the aforementioned plain metrics, some other more specialized metrics that take location information (is-long-distance attribute) into account are present. Specifically, in addition to "Call" metric, we also consider "local Call" and "non local Call" metrics. Similarly, we keep track of money spent on local and non local calls ("local Cost" and "non local Cost" metrics respectively). The same pattern holds for Duration as well ("local Duration" and "non local Duration").

In section 1.2 we noted that the SEP system retains information regarding subscribers' event history with the aim to match campaigns to them. This information is a set of statistics consisting of aggregations over time regarding the metrics we defined before. For instance, total money spent (Cost) this week, or longest long distance call (maximum non local Duration) today belong to this set. Therefore, they are of importance as they reflect the up-to-date subscriber profile. These statistics are the attributes that form the Analytics Matrix (AM) component. There are a couple of properties associated to any of these attributes. Moreover, an AM attribute has the following properties:

- Value type
- Window type

The combination of these properties determines the size and the computation complexity of an AM attribute.

2.1.1 Value Type

With respect to their value type, AM attributes can be divided into two categories:

- Selective (longest call, highest cost, minimum duration, etc.)
- Accumulative (number of calls, average cost, total duration, etc.)

This distinction is based on the way the value of an AM attribute is calculated. In other words, it originates from the aggregation function applying on the metrics. The aggregation functions: min and max belong to the Selective value type, whereas the sum and the average functions are part of the Accumulative value type.

We would like to state here that for the average aggregation, in contrast to the other functions, we maintain one more value; we need one value recording the current sum of the values and a counter of the observations occurred. We opt not to use another place for storing the actual average value, but to compute it on demand with the aim to keep space consumption as low as possible.

2.1.2 Window Type

AM attributes are also characterized by a window type. Furthermore, we pair an AM attribute value with time information showing the time interval within which the value is valid. For this purpose we use the timestamp attribute of the subscriber's last event, which we store into the AM record. This timestamp indicates when a call event starts. Time windows slide as new events arrive at the system; old values are squeezed out and fresh values replace them. We consider three possible window types:

- Tumbling (e.g. last week)
- Stepwise (e.g. last four days with step equals to two days)
- Continuously moving (e.g. last 24 hours)

As far as the Tumbling window is concerned, we would like to point out that this type is a special case of the stepwise type, in which the window moves by step equal to its size. Tumbling window is the simplest type in terms of space consumption and computational effort. Regarding the former, one value is sufficient for recording all the aggregation types, but the average (we need two values). Given a newly arrived event we must decide whether this event belongs to the same time window as the previous event of this subscriber. If so, we update the corresponding value. Alternatively, we have to reset the value.

Stepwise windows are defined by their size and step, that is, the slide size. We assume that the window size is dividable by the window step. For example, we cannot think of any meaningful use case for a window having a size of 6 days and a step of 4 days. This assumption results in saving space, as we have to

store size/step values instead of size values. Again here, the average aggregation type needs more space because size/step counters must be kept additionally to the values. Similarly to the previous situation, if an event arrives at the system, we firstly calculate whether the previous existing window is still active, or a new window has displaced it. In the first case, we simply modify the value accordingly, whereas in the latter, we have to compute how many steps the currently active window has been slid from the previous one. This number indicates how many of the values we maintain are stale, and hence, should be reset. Recording the fresh value is the last step of the update procedure.

The Continuously moving window is the most complicated window type. What is more, it is not resettable as it is always active. This window moves constantly, information is added incrementally and it is removed continuously. The space needed for the Continuous window cannot be computed in advance; something that is doable for the other two cases mentioned above.

Each one of the three window types is associated to size information denoting how long the window lasts. Initialization information specifying the start boundary of the very first window becoming active is also needed. Based on this fundamental information we are able to calculate the start, and as a result, the end point of any later window. Moreover, the moment for resetting an attribute's value is needed in the first two window types. For the sake of simplicity, we take for granted that a window always starts at 00:00 time and it ends at 23:59. Let us consider a tumbling window with size of one week. In this instance, the window starts every Monday at 00:00 and it ends the following Sunday at 23:59.

2.1.3 Example

As of now, our system supports all the different value types for the tumbling window type. Table 1 illustrates an instance of the Analytics Matrix in the system. One can see that the table hosts one record per subscriber. Also, the timestamp we use for calculating the active window for each AM attribute is depicted. A number of attributes having tumbling window type and different value types is presented. The "Calls this week" attribute has accumulative value type (sum), whereas the second statistic we maintain (Max local Duration today) has selective value type (max). Finally, the "Avg Cost today" has accumulative value type (average). It is clearly shown that for the average aggregation we store two values: a running sum and a counter of the observations.

Subscriber	Timestamp	Calls this week	Max Duration of local calls today	Avg Cost today	
1	1358938791	3	20 mins	$1.4	1
2	1362948694	7	53 mins	$9.1	3

Table 1: An instance of the Analytics Matrix

2.2 Implementing the AM

Two dimensions have been considered with regard to the implementation of the AM. The first dimension refers to the separation, or integration of storage layer and event processing logic. To be more specific, we can either make the entire AM table visible to all the application servers, or split the AM table up[5] into smaller parts and designate an app server to each one of these parts. Both approaches have advantages and disadvantages that are studied below. The second dimension is related to the technology we employ for the physical store of the AM table. Two alternatives exist: Relational Database Management Systems (RDBMS) and Key-Value stores.

2.2.1 Separated Storage

In the Separated Storage (figure 2) we opt to detach the storage layer from the event processing cycle. One can consider this approach as a shared-disk architecture[12], where the AM table lies in a shared resource. Here, an application server is responsible for storing campaigns along with the campaign index. Additionally, an event may be routed to any available app server. All machines are able to process events of all subscribers as they see the entire AM table. The fact that a new layer (storage layer) is introduced, leads to an increased latency. The ability to scale out/down and to ensure fault-tolerance compensates for the aforementioned overhead. Since all app servers are granted access to the AM, starting a new app server only entails reading the campaigns and building the campaign index. Moreover, no pivotal problem arises in case of an app server failure, since the index is fully reconstructible and the subscribers' history is kept on an external machine.

The technology we have opted for storing the AM is RamCloud[13]. RamCloud is a scalable high-performance key-value store and it is featured by appreciable low latencies[6]. It is optimized to run in an Infiniband network[17]. Additionally, it allows for conditional writings, which are of utmost importance for this project. In lay terms, a write is applied if and only if a specified condition is fulfilled (the value we retrieved has not been modified in the meantime). We have left the examination of fault-tolerance out of the picture; we will deeply study it later.

2.2.2 Integrated Storage

In the Integrated Storage (figure 3), on the other hand, every application server acts individually (like in a shared-nothing architecture[12]), since it not only stores campaigns and the campaign index, but it also maintains its own part of the AM. Moreover, each application server processes events belonging to a fixed set of subscribers. Particularly, all the events of a specific subscriber are always forwarded to the same server being responsible for maintaining his/her profile.

[5] Horizontal partition based on the subscribers' identifiers.
[6] Network latency dominates.

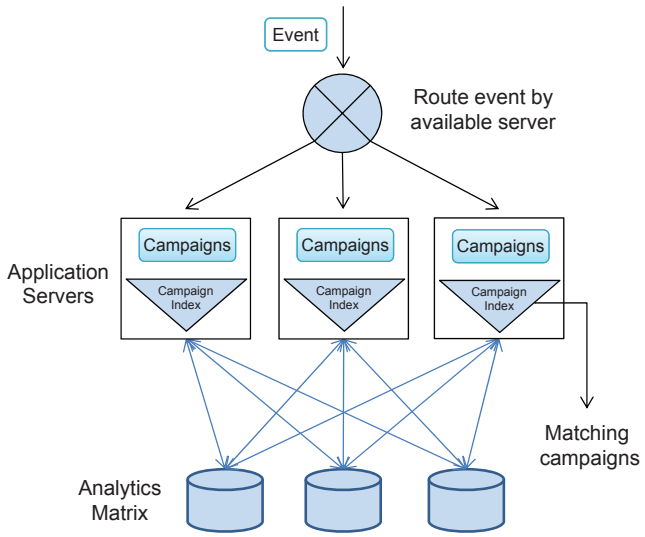

Figure 2: Event processing using the Separated Storage

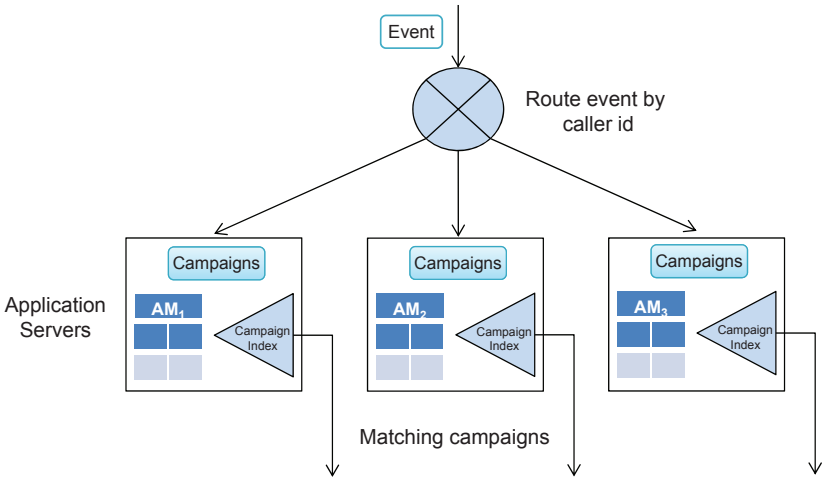

Figure 3: Event processing using the Integrated Storage

For this approach we have decided again to experiment with in-memory hash tables: local hash table[7] as well as MemcacheD[14]. After running some

[7] Lock-free, thread-safe hash table implemented in C++.

microbenchmarks we decided to exlude MemcacheD, as the latencies we acquired were far beyond the specified limits. Given that tuning MemcacheD's performance is not part of this thesis, we stick to the local hash table. As a matter of fact, scaling the system and implementing fault tolerance cannot be done off-the-shelf. Scalability requires re-organizing the subscribers group of each server, while for fault-tolerance some source of backup must be taken into account. Modelling and implementing system's fault-tolerance is not in the scope of this thesis.

2.3 Updating the AM

In this section we illustrate the procedure we follow for updating the AM, in case an event arrives at the SEP system. Retrieving the record of the subscriber that generated the event is the very first step of algorithm 1. Then, we update all the attributes forming the AM record one after the other. For doing so, we make use of the current event's attributes value as well as information linked to every AM attribute. The properties (value and window type) characterizing an attribute are essential for its proper update. At that point, the updated AM record of the underline subscriber is held. Following algorithm 1, we try to write the newly updated record back to the storage layer[8]. We use a conditional write for this operation, which returns 'false' if the record was modified by another thread/process in the meantime. This way we assure that updates are performed only on the most up-to-date subscriber's record. If a write failure occurs, we repeat the entire procedure. Otherwise, the algorithm terminates returning the record.

Algorithm 1 Update the Analytics Matrix

```
1: function UpdateAM(Event)
2:     record ← storage.read(event.callerId)
3:     for each attribute in: AM record do
4:         attribute.update(event)
5:     end for
6:     result ← storage.write(event.callerId, record)
7:     if result = false then
8:         goto2
9:     end if
10:    return record
11: end function
```

[8] RamCloud and local hash table can be interchangeably employed for maintaining the AM. Both engines implement the same API.

3 Campaigns

In this section the reader finds a detailed description of campaigns. As mentioned before, a campaign is defined by a pair (rule, reward), and it is only activated when a subscriber profile meets the condition implementing the rule.

3.1 Properties

Every Campaign is characterised by a set of properties we depict in figure 4. A comprehensive description of all these properties follows.

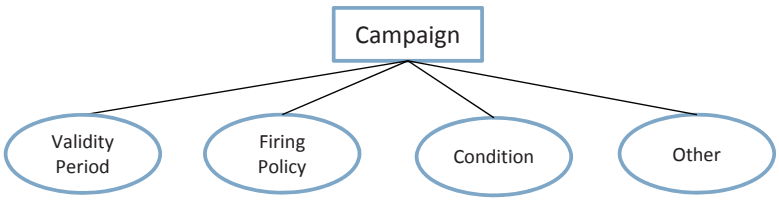

Figure 4: Campaign properties

3.1.1 Validity Period

Validity Period corresponds to the time interval within which the campaign is active. A range [valid from, valid to][9] suffices for declaring this kind of information.

3.1.2 Firing Policy

This property indicates whether a campaign can fire multiple times (no firing restrictions), or there exist constraints enforcing it to fire once within a time period. Moreover, two sub-properties compose the Firing Policy property. The first one is called Firing Interval and defines the interval in which a campaign can fire (e.g. 0 = always, 1 day, 1 week, etc.), whereas the second (named Firing Start Condition) declares if this interval moves based on a fixed or a sliding manner. The following examples demonstrate how these sub-properties affect a campaign's behavior:

- **If** Firing Interval: 1 day and Firing Start Condition: fixed
  then campaign can fire each day only once between 00:00 and 23:59
- **If** Firing Interval: 1 day and Firing Start Condition: sliding
  then campaign can fire if it did not fire within the past 24 hours

[9] Both bounds have Timestamp (combined Date and Time information) data type.

Firing Policy property has only been modelled and not yet implemented. For the time being, we assume that all the campaigns have no firing restrictions, meaning that they fire as long as their condition is fulfilled.

3.1.3 Condition

This property denotes what has to be fulfilled for a campaign to fire. The Condition can be considered as a boolean formula consisting of predicates on AM attributes, event attributes and static subscriber data. The following examples give the reader an idea of different Conditions:

- [number of calls today >= 2] OR [longest call this week > 100 minutes AND event.place-of-caller like "ZUR%"]
- [number of long-distance calls this week > 5 AND total-cost today > 4] OR [number of local calls this week > 20]

The Condition property will be thoroughly explored in Section 3.2. There, the reader can find examples illustrating the way we have modelled this property as well as the functionality we support in the current version of the system.

3.1.4 Other

Other corresponds to a category rather than a single property. It wraps properties of campaigns that are not taken into consideration in this thesis. We have stumbled upon two of these properties:

- Action
- AM attributes initialization

The former refers to the consequence of a campaign that fired. For instance the subscriber can get a discount of 10% for the next call. Regarding the latter, different options exist. We can either employ the "historic", or the "fixed" initialization. Based on the "historic", every time a new campaign is registered to the system the values of the AM attributes are decided using the subscriber's history. On the contrary, if "fixed" initialization has been chosen, default values are assigned to the attributes, e.g. we set "Cost this week" to 0.

3.2 Implementation

In this section, the condition property, which apparently constitutes the most fundamental part of a campaign, is examined in detail. As explained in 3.1.3 this property determines under which circumstances a campaign can fire. The way we model condition and the data structures we make use of are studied in 3.2.1 and 3.2.2 respectively.

3.2.1 Representation

Having in mind that the evaluation of a campaign condition should be immediate, we choose to model it as a tree-like structure being in Disjunctive Normal Form[10] (DNF) [15]. What is more, at the root of this tree we place the OR clause, which can have one or more children. At depth one we have AND clauses with one or more children too. Finally, leaf nodes are designated to store predicates. As a result, a campaign condition is formed of a number of Conjuncts (i.e. AND clauses) joined with OR clauses, and each Conjunct accommodates a number of predicates.

At that point let us explain how a predicate looks like. A Predicate is defined by its three children and denotes a comparison between the Attribute and the Constant:

- Attribute: AM attribute or current event attribute
- Operator: <, <=, =, >=, >, LIKE[11]
- Constant

The following campaign condition example, along with figure 5 provides a better understanding of our condition modelling.

Condition: [number of calls per week > 1 AND total money spent per day >= $30] OR [number of non local calls per month > 2]

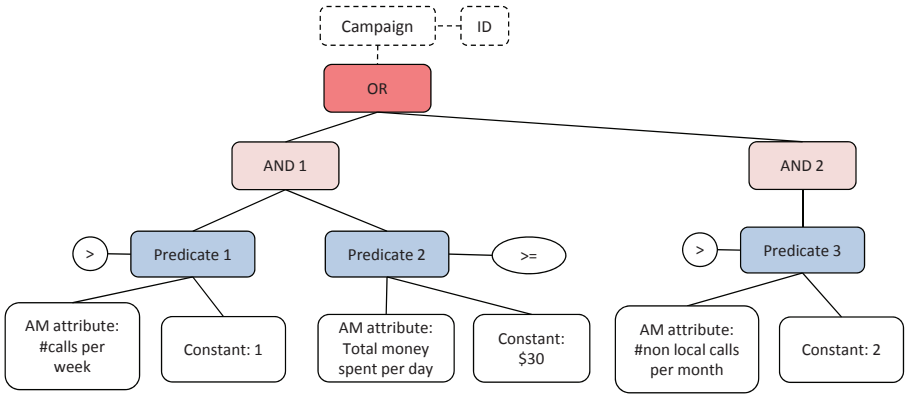

Figure 5: Tree representation of a Campaign Condition

Considering Condition as a statement in DNF aids us in speeding up the evaluation procedure. To be more precise, given the fact that this statement is

[10] We do not allow for negations in our DNF modelling.
[11] Currently, we only support the operators {<, <=, >=, >}.

a disjunction of conjunctions, the first time a Conjunct evaluates to true[12] we can exit evaluation, as the campaign condition has already been met. This way we need not examine all the other Conjuncts. Additionally, as we mentioned before, a Conjunct is a sequence of Predicates combined with AND clauses, and thus, Conjunct evaluation aborts when a Predicate's comparison results in false. In addition to Predicates, a Conjunct also stores the identifier of the campaign it belongs to. This way when a Conjunct evaluates to true, we know which campaign has been triggered.

3.2.2 Data Structure

In section 3.2.1 we elaborated on the reason we follow the DNF representation for the campaign condition. According to this modelling, a condition can be considered as a list of arrays, where every array stands for a conjunct, and it contains an arbitrary number of predicates.

We assume that a predicate may belong to more than one conjuncts, and as a consequence to multiple campaigns. Because of this, predicates are represented globally in the system. In other words, we maintain an array containing all the predicates existing in a deployment of the system. Therefore, every conjunct holds references to predicates, rather than actual predicates. Figure 6 provides a thorough description of this global representation for two conditions having a number of predicates in common to check. There, one can easily notice that conditions have the shape of lists, and that they share predicates (i.e. predicates with identifiers: 3, 4).

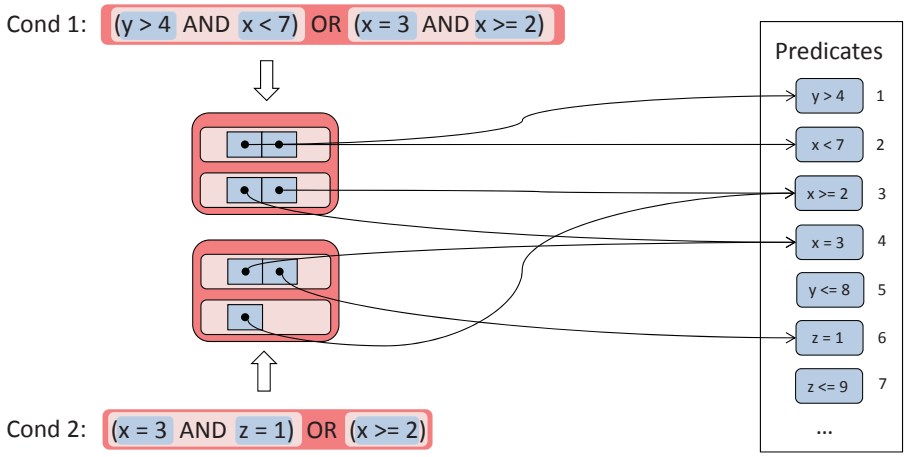

Figure 6: Global Representation of Predicates

[12] A Conjunct evaluates to true when all its Predicates result in true.

4 Campaign Index

The second crucial component of the "Materialized View" approach is the so-called Campaign Index. This index is campaign-driven, meaning that it does not exist prior to campaigns and it is built with the aim to facilitate their evaluation. Ideally, Campaign Index will assist us in finding matching campaigns in $O(log(n))$, rather than $O(n)$, where n is the number of campaigns.

4.1 Motivation

The Stream and Event Processing system this thesis describes resembles a publish/subscribe system as it manages a stream of incoming user-generated events and a set of rules that must be evaluated against this stream. As [9] proposes, grouping subscription predicates based on their common attributes is advantageous, since many subscription predicates can be evaluated at the same time.

We assume that there exist AM attributes (e.g. number of calls this week, total cost today, etc.) that appear in conjuncts, and thus in campaigns very frequently. Moreover, the assumption that only a few of these attributes exist in the system is made. We name these few attributes Entry Attributes and the predicates involving them Entry Predicates. Following the idea of [9], we believe that building indexes (called Entry Indexes) on Entry Predicates is beneficial for the campaign evaluation procedure, since these indexes function as a filter pruning the campaign space.

Regarding the selection of Entry Predicates, we would like to point out that finding a set that is minimal and at the same time it covers as many conjuncts as possible is not trivial. One option is to fix the number of Entry Attributes and then try to find the best coverage. That is, locate the desired number of AM attributes participating in the most conjuncts. A different way to come up with a set of Entry Attributes is to specify the percentage of conjuncts that must be covered (i.e. contain an Entry Attribute) and afterwards find a minimal set that reaches the specified coverage. For the time being, we simply define this set in advance and we build our campaigns in such a way that most of their conjuncts contain Entry Attributes.

4.2 Building the Index

The following sections clarify the notion of indexing a predicate and present the data structures we have thought about, respectively.

4.2.1 Indexing a Predicate

In section 3.2.1 we mentioned that a predicate is a triple consisting of an AM attribute, a constant and an operator { $<$, $<=$, $=$, $>=$, $>$, LIKE}. Consequently, different kinds of predicates (range, equality and LIKE predicates) exist in the system. Nevertheless, as of this moment, we only consider range predicates[13].

[13] We will study the other types of predicates in a future phase.

Dealing with range predicates enables us to use an ordered tree data structure being able to hold intervals. For this reason we decide to use 1-dimensional R-trees (Interval Trees) [10] for indexing entry predicates appearing in the system. This data structure allows one to efficiently find all intervals that overlap with any given value. As far as the R-tree's implementation is concerned, we use a cache-conscious, extremely space efficient, packed, static, 1-dimensional R-Tree provided by the Crescando[16] team.

Indexing a predicate boils down to finding the interval for which the range predicate results in true, and to inserting it, along with the respective value into the entry index. More precisely, we use the interval's bounds as the key of the record we index. Furthermore, the value being associated to this compound key ([lower bound, upper bound]) is the conjunct containing this predicate (recall that every predicate is part of a conjunct). Bearing in mind that a predicate may belong to multiple conjuncts, we opt to store a list of conjuncts instead of a single conjunct per key. So, given a value an entry index reports all the lists of conjuncts whose keys overlap with this value.

4.2.2 Data Structures

The proposed Campaign Index is composed of two parts:

- Entry Indexes
- Unindexed Conjuncts

Regarding the former, we maintain an array of entry indexes having as many places as the number of entry attributes in the system[14]. Each entry index is dedicated to one entry attribute and it is responsible for indexing (as described in section 4.2.1) all predicates containing this attribute. The Unindexed Conjuncts part refers to an array of conjuncts, where we store conjuncts involving non entry predicates. This conveys that a conjunct is either placed at a leaf of an entry index, or it exists in the Unindexed Conjuncts.

During campaign index building a campaign is decomposed into its conjuncts and every conjunct is in turn broken down to its predicates. In case of an entry predicate, we extract the matching interval (the interval for which the predicate's comparison is true), and after pairing it with the conjunct this predicate is part of, we insert them into the entry index. Otherwise, the conjunct is pushed back to the Unindexed Conjuncts.

For the scope of this Thesis we consider the Campaign Index component as immutable, in the sense that it is built offline (before the system starts serving events) and no extra campaigns can be added, or existing campaigns can be removed[15].

A complete example showing a snapshot of the Campaign Index for a set of campaigns is presented in Figure 7. With the aim to keep the example simple, we choose to have only one entry attribute (i.e. attribute x), and hence, one

[14] We do not build indexes for entry attributes being involved in no predicates.

[15] The possibility of adding campaigns dynamically will be introduced at a later stage.

entry index (built on top of attribute x). The reader can observe how the contents of this entry index are structured. To be more specific, there are three entry predicates (P1, P2 and P3) participating in two, one and one conjuncts, respectively. Furthermore, we have two unindexed conjuncts with one predicate each in that set-up.

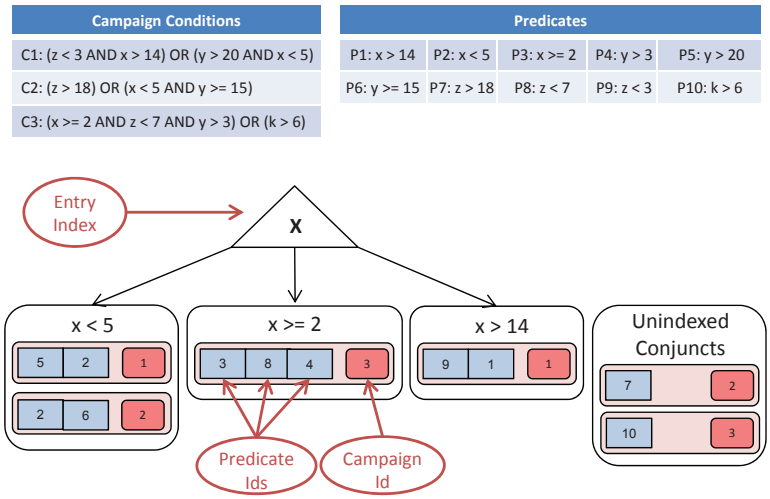

Figure 7: An Instance of the Campaign Index

4.3 Probing the Index

This section presents the algorithm (i.e. algorithm 2) that is used for finding relevant campaigns that potentially fire as a result of an event. The illustrated function is invoked for every incoming event and it takes the updated subscriber profile (AM record) as an argument. Additionally, the function returns the identifiers of the activated campaigns. One can easily observe that Algorithm 2 comprises the following two steps:

1. Find Candidate Conjuncts
2. Evaluate Candidate Conjuncts

Firstly, all entry indexes are probed using the AM record (profile) of the subscriber that incurred the event. Specifically, every single index is queried with the value of the AM attribute this index is built upon. This way conjuncts including entry predicates that do not match are ruled out without further examination[16]. By the end of Step 1 a list containing the Candidate Conjuncts is

[16] In case of an unsatisfied predicate, the conjunct holding it is not satisfied too.

obtained. A conjunct holding an entry predicate is labeled as candidate when its entry predicate matches. Moreover, since we have no information with regard to unindexed conjuncts, they are always part of Candidate Conjuncts.

In the second step of the algorithm we examine all the Candidate Conjuncts consecutively to find out which of them evaluate to true. The outcome of this procedure is the matching campaigns[17].

Algorithm 2 Find Matching Campaigns

```
1: function MatchCampaigns(Profile)
2:     CandidateConjuncts ← Unindexed Conjuncts
3:     for each index in: Entry Indexes do
4:         CandidateConjuncts ← index.getCandConjs(Profile)
5:     end for
6:
7:     Matching ← ∅
8:     for each conjunct in: Candidate Conjuncts do
9:         result ← conjunct.evaluate(Profile)
10:        if result = true then
11:            Matching.append(conjunct.CampaignId)
12:        end if
13:    end for
14:
15:    return Matching
16: end function
```

Figures 8 and 9 visualize the first and second part of the Matching Algorithm. In the former the set of Candidate Conjuncts is acquired (i.e. dotted rectangles), whereas in the latter their evaluation takes place. In Figure 8 we probe the entry index with the value x = 3. What is more, two entry predicates are satisfied with this input value resulting in three conjuncts that must be further examined. Clearly, unindexed conjuncts must also be evaluated. Therefore, we should check five conjuncts in total. Given the values of the other AM attributes and what rules the predicates specify we end up with two conjuncts evaluating to true. In the end, we report the campaigns with identifiers 2, 3.

[17] Recall that if a conjunct is true, the campaign including it triggers.

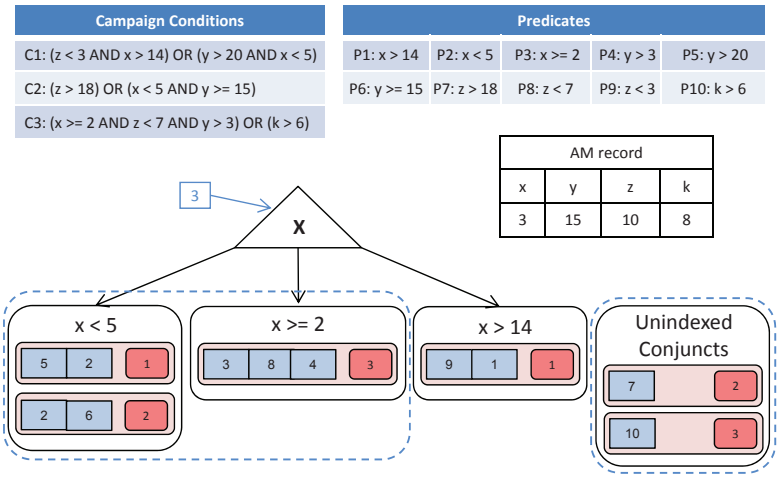

Figure 8: Find Candidate Conjuncts

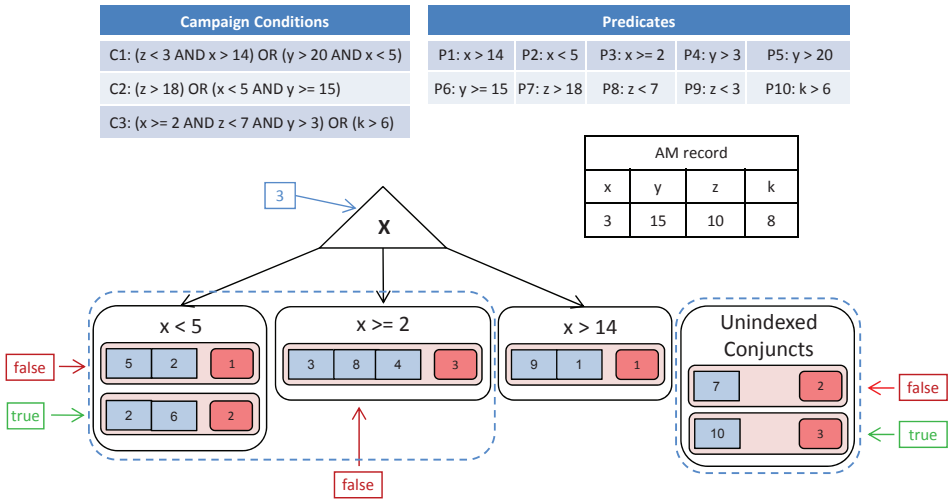

Figure 9: Evaluate Candidate Conjuncts

5 Performance Evaluation

In this section we evaluate the performance of our Stream and Event Processing system and compare the alternative architectures (Separated Storage, Integrated Storage). Section 5.1 describes the benchmark and the workload the system is subject to. Section 5.2 details the system's setup. Finally, Section 5.3 presents the experiments we conducted along with their analysis.

5.1 Benchmark and Workload

We have implemented a workload generator that creates a number of Analytics Matrix attributes and a set of campaigns according to a workload specification. We store both the attributes and the campaigns into a MySQL[19] database (named meta-database) which is loaded at start time and based on this information we set up the data structures being used during the execution of the program. All attributes in the AM have tumbling window type with size of either one day, or one week. Moreover, AM attributes can take all the possible value types (accumulative, selective). Applying all different combinations of window size, metric and aggregation function results in a set of 54 distinct attributes[18]. Attributes such as:

- Number of local calls today
- Cost of long distance calls this week

exist in a deployment of the system. In addition to the AM attributes, the benchmark generates a number of campaigns[19]. The campaigns we are experimenting with are constructed using multiple conjunctions/disjunctions on AM attributes. An example of a campaign condition that could exist in the system is:

- [avg(local cost) $>$ 27.0 $\wedge$ sum(local duration) $<$ 62] $\vee$ [avg(total cost) $>$ 71.0 $\wedge$ max(long distance duration) $>$ 40 $\wedge$ number of calls $>$ 10]

Conditions are generated in such a way that each conjunct most probably (i.e. with probability 90%) contains a pivot attribute. A popular AM attribute occurring in many campaigns is called Pivot. Our proposed campaign index is constructed from scratch by reading the meta-database. This operation is not performance critical, because it takes place before we start serving events. The number of entry indexes equals the number of Pivot attributes, while the number of unindexed conjuncts exclusively depends on the number of campaigns.

As far as the workload is concerned, we assume uniform distribution of events over subscribers. In addition, every subscriber incurs 28.8 billing events per day. An event is composed of a set of attributes such as caller-id, is-long-distance, duration, cost, etc. Due to the huge number of events, we choose

[18] Not all possible combinations make sense, e.g. max(number of calls).

[19] A campaign comprises a number of properties as described in 3.1, but for now we only consider its condition.

to create and emit events on the fly, rather than reading and storing them in main memory[20]. Every thread running on the application server is not only responsible for handling an event, but also for generating it. As a result, no network is involved for an event to enter the system.

5.2 Experimental Setup

We ran all experiments on workstations having two CPU sockets with 4 cores at 2.40GHz per socket, 128GB RAM operating under Linux and communicating via InfiniBand.

The number of AM attributes remains constant (i.e. 486 attributes) throughout the experimental session. This number corresponds to AM records with size of approximately 3KB. To simulate this large number of attributes, we keep each of the 54 unique attributes mentioned before several times (i.e. 9 times). Moreover, 20 pivot AM attributes exist in the system entailing the existence of 20 entry indexes in the campaign index. If it is not stated differently, 300 campaigns are active in the system. Each one of these campaigns has at most five conjuncts and every conjunct in turn consists of at most five predicates.

In this series of experiments we measure the throughput (events/s) and the average response time of the system. We also measure the quality of service, that is, the percentage of events that fulfil the specified time requirements (i.e. latency $<$ 10ms). Timings are taken in microseconds and they are presented in milliseconds. Response time captures the whole event cycle (i.e. event generation, AM record update and campaign evaluation). Regarding experiments duration, we want to point out that all of them ran for 1 hour. During this time period, we first populate the database (RamCloud or local hash table) with a default record[21] per subscriber, and then the system starts serving events.

For all the experiments, but the scalability, one application server is used. As far as the storage layer is concerned, the separated storage approach occupies 3 RamCloud servers. The integrated storage, on the contrary, needs no RamCloud server, as it uses a local hash table.

In the beginning of this section we noted that our workstations have 8 CPU cores. With the aim to avoid contention between threads and to keep latency low, we select to bind each app server's thread to a CPU core. Additionally, for the separated storage, 7 threads[22] in a closed system[18] are running simultaneously, while for the integrated storage all the 8 threads can be exploited.

5.3 Results and Analysis

The first experiment evaluates the performance of the system in regard to the number of subscribers. As mentioned in 1.3, one of our goals is to maximize the

[20] Differently, memory consumption would become an important issue.
[21] This record consists of the default value for every AM attribute (i.e. Cost = 0).
[22] One additional thread handles communication with RamCloud.

number of subscribers a machine can handle. The following two graphs[23] illustrate the numbers we attained for throughput and response time respectively. As far as Figure 10 is concerned, throughput remains essentially the same as the number of subscribers increases. Analogously, response time (Figure 11) does not change significantly with the subscribers number. The reason for this roughly constant behaviour is that the number of subscribers do not alter the load that is put to the application server. Specifically, the event arrival rate depends only on the number of threads issuing events. Since, this number does not change during execution, the load arriving to the system does not change either. One can notice that the integrated storage approach outperforms the separated storage one. The fact that a machine demands more CPU cycles to issue an external request to RamCloud, than to the local hash table, explains this difference in performance. In conclusion, both architectures comply with response time and throughput requirements as specified in 1.3, and therefore both can be employed.

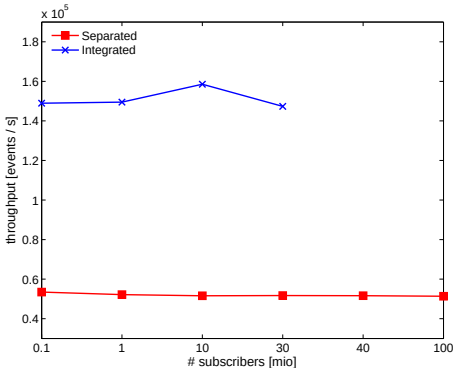

Figure 10: Throughput Comparison

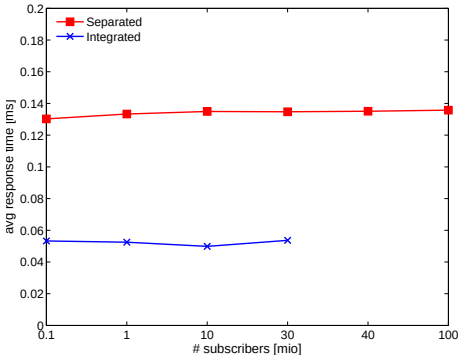

Figure 11: Response Time Comparison

Figure 12 gives the reader an idea of our service quality subject to the number of subscribers. In other words, it demonstrates the percentage of requests that were completed within the specified time limits (i.e. 10ms). Apparently, more than 99.9% of the events issued were answered in time, verifying that adopting any of the two architectures is feasible.

The next two experiments (Figure 13 and 14) depict how the system scales with the number of application servers, across all alternatives. Given that for the integrated storage approach each app server acts individually, adding a new machine does not impact on the performance of the existing machines. This explains why we get linear scale out for this architecture. In the separated storage approach, on the other hand, performance loss appears, due to the fact that every additional machine consumes shared resources, and hence affects other machines' performance. The performance gain of the integrated storage

[23] In the integrated storage, an app server maintains part of the AM. Given that a machine has 128GB RAM and an AM record has a size of 3KB, it becomes clear why we cannot handle more than 30 million subscribers, and that is why we have missing value.

architecture derives again from the little computational effort a request to the local hash table needs.

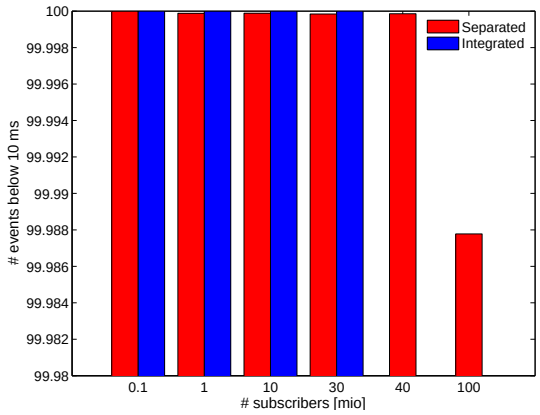

Figure 12: Quality of Service

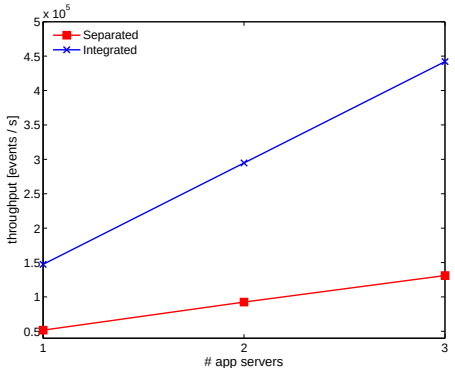

Figure 13: Throughput Comparison

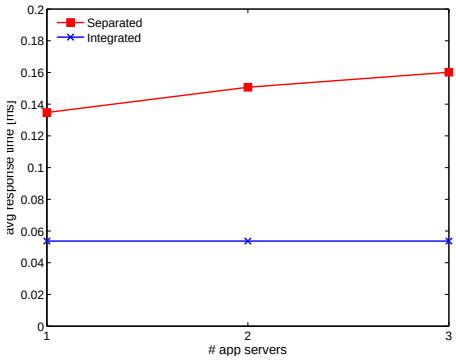

Figure 14: Response Time Comparison

Figures 15 and 16 show that both architectures are sensitive to the number of campaigns. Clearly, these results follow our natural intuition. The more computational work the system must carry out, the longer an event service lasts. Consequently, throughput decreases with the number of campaigns, and correspondingly response time increases.

Being interested in comparing our alternatives to a baseline, we consider a system configuration in which no entry indexes exist. So, in that case, our campaign index comprises only the unindexed conjuncts component. Regarding the probing procedure, all conjuncts must be evaluated one after the other for finding the matching campaigns. This brute force approach eliminates the advantage that entry indexes give to the system, as no conjunct can be excluded

from evaluation[24]. Tables 2, 3 compare the throughput we measured for the different configurations subject to the number of campaigns. The last column (i.e. Diff) of both tables shows how much better than having no index, the proposed campaign index performs.

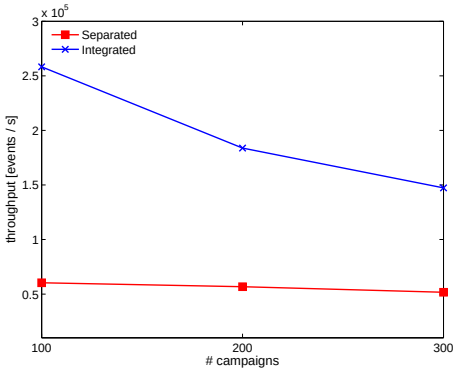

Figure 15: Throughput Comparison

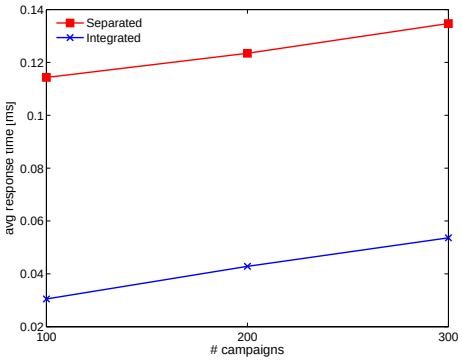

Figure 16: Response Time Comparison

Campaigns	Separated with Index	Separated no Index	Diff [%]
100	60,392	61,387	-1.62
200	56,747	57,496	-1.30
300	51,710	51,438	0.53
1,000	36,473	37,108	-1.71
10,000	8,050	5,244	53.49

Table 2: Throughput (events/s) Comparison for the Separated Storage

Campaigns	Integrated with Index	Integrated no Index	Diff [%]
100	258,268	269,479	-4.16
200	183,829	195,160	-5.81
300	147,353	152,105	-3.12
1,000	58,988	57,889	1.89
10,000	9,418	5,802	62.31

Table 3: Throughput (events/s) Comparison for the Integrated Storage

[24]Conversely, in case of entry indexes, we only evaluate conjuncts whose entry predicate matches.

Furthermore, Figure 17 compares average response times for the various configurations. It turns out that for small number of campaigns (up to 1,000) the brute force approach performs slightly better than the one with the entry indexes, but from a certain point onwards the campaign index pays off (up to 62.31% better for the Integrated Storage architecture). This performance difference (for large number of campaigns) is explained by the fact that the system is obliged to evaluate all the existing conjuncts for the brute force configuration, while, in the case of campaign index, a significant number of them is not examined. For the use case[25] that inspired this thesis it seems that evaluating all campaigns sequentially is sufficient (since we deal with a relatively small number of campaigns) and no extra space consumption (for building the entry indexes) is necessary. Nevertheless, other use cases, where the number of campaigns/rules/subscriptions is considerably larger (in the order of magnitude of tens or hundreds of thousands) than in the underline use case, lead us towards the campaign index direction.

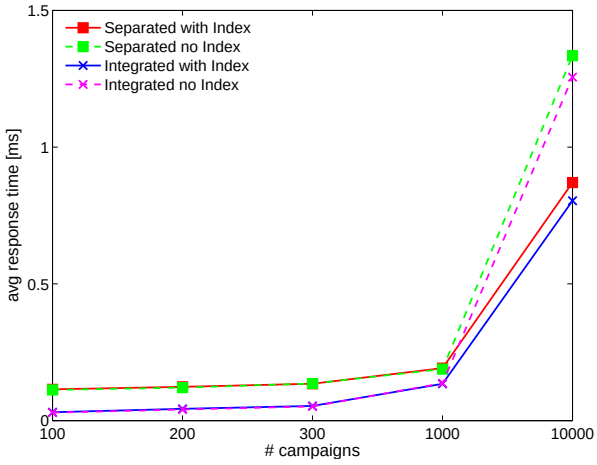

Figure 17: Response Time Comparison

[25] A system encompassing a relatively small number (for instance 300) of campaigns that should be evaluated against a stream of user generated events.

6 Conclusion and Outlook

In this document we proposed a system that can efficiently execute real-time processing tasks on a stream of incoming events. The idea arose from a telecommunication company being interested in managing subscriber-generated events, with the aim to extract valuable information out of them. Our system, named SEP, is responsible for maintaining aggregated information (statistics) deduced from subscribers billing events, and matching this source of information to a number of rules residing in it. We have optimized the mechanism for maintaining the statistics in such a way that information is added incrementally without the need for revising a subscriber's history (past events). As far as the matching task is concerned, SEP implements a memory-based algorithm relying on tree data structures that allow for quick examination of rules.

In addition to the theoretical analysis, a set of experiments depicting the performance of the SEP system was presented. Moreover, the experimental results act as a proof-of-concept for the feasibility of our architecture. Specifically, the "Materialized View" approach that SEP follows ensures a high throughput (tens of thousands of events per second) and an extremely low response time.

While working on this project a lot of ideas have been arising with respect to future developments. Some of them are in the direction of introducing new functionality to the existing system (for example, other window types), while others propose alternative ways to address the entire problem.

Clearly, the very first thought coming to our minds involves replacing the "Materialized View" approach studied in this thesis with a data stream management system. This decision entails implementing every campaign as a continuous query and modelling events as a data stream. It is still debatable whether the ease of using a stream engine compensates for the lack of memory management control. Stream engines, such as Esper or Storm, provide users with out-of-the-box solutions, but their support comes at a price. No access to internal state and data is allowed.

Migrating the event generation task from the application server to a client layer would be an important improvement to the current system. This amendment will give us the opportunity to measure the real time[26] SEP needs for serving an incoming event, and it will also aid us towards separation of concerns.

Introducing more features to the current SEP system is something we are also aiming at. In this direction there is still a lot of room for improvement. Moreover, in our view, implementing campaign properties (section 3.1) is a high priority goal, since it would bring us a step closer to realistic use cases. Adding constraints would undoubtedly prolong the campaign evaluation time, and would also require auxiliary data structures (e.g. maintain one more table in RamCloud). Other complications should also be taken into account. Matching a campaign condition will not necessarily indicate that a campaign has been triggered. Firing restrictions should also be examined before we report that a

[26] Measurements will include receiving a request and sending back the response via network (TCP/IP).

campaign matches. Further features we could consider are:

- implementing more complicated window types (i.e. stepwise window)
- making the system more dynamic by allowing for new campaigns to be registered, or existing campaigns to be deleted during execution
- enhancing the dynamic nature of the system by modifying (inserting or deleting) the set of AM attributes while the system is up and running
- maintaining subscriber information and using it in campaigns (e.g. LIKE operations)

Going one step further, from our point of view, the ultimate goal is the combination of the SEP and RTA[27] (section 1.2) systems with the aim to build the full-fledged AIM system. This system should comply with the AIM specification and it should be able to deal with the workloads of both sub-systems.

We believe that the system we came up with, lays the groundwork for more complex systems sharing the same characteristics as the ones presented in this thesis. Namely, systems that are fed with intense workloads and must apply simple, yet not trivial, computations on extremely large amounts of data by ensuring high event rates and low latencies. As data has grown in size and complexity, and time restrictions have become stricter new solutions must be employed for dealing with these conditions. What we consider as the most important contributions of this thesis, is the way we treat aggregations of information, and the abstractions we follow for modelling the rules. These general principles can be used for shifting through volumes of data to produce knowledge and detect patterns.

[27] Upon completion of the Real-Time Analytics system.

References

[1] Esper. *http://esper.codehaus.org/*, March 2013.

[2] Storm. *http://storm-project.net/*, March 2013.

[3] H. Carcia-Molina, J. Ullman, J. Widom. *Database systems: the complete book*. Pearson Prentice Hall, 2009, 301-308.

[4] E. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. Park, A. Vernon. Scalable trigger processing. In *Proceedings* of the *Int. Conf. on Data Engineering*, 1999.

[5] E. N. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings* of the *ACM SIGMOD Int. Conf. on Management of Data*, 1992.

[6] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, S. Zdonik. Aurora: A Data Stream Management System. In *Proceedings* of the 2003 *ACM SIGMOD Int. Conf. on Management of Data*, 2003.

[7] S. Babu, L. Subramanian, J. Widom. A Data Stream Management System for Network Traffic Management. In *Proceedings* of *NRDM*, 2001.

[8] M. Aguilera , R. Strom , Daniel Sturman , M. Astley , T. Chandra. Matching events in a content-based subscription system. In *ÎŢighteenth annual ACM symposium on Principles of distributed computing (PODC '99)*, 1999.

[9] F. Fabret, H. Arno Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings* of the 2001 *ACM SIGMOD*, Pages 115-126.

[10] A. Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14 (June 1984), 47-57.

[11] J. Rao , K. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings* of the *25th International Conference on Very Large Data Bases*, 1999.

[12] Michael Stonebraker. The Case for Shared Nothing. *In Database Engineering Bulletin*, Vol. 9, No. 1. (1986), pp. 4-9.

[13] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev. 43* (January 2010), 92–105.

[14] MemcacheD. *http://memcached.org/*, March 2013.

[15] Disjunctive Normal Form. *http://en.wikipedia.org/wiki/Disjunctive_normal_form*, March 2013.

[16] G. Giannikis, P. Unterbrunner, J. Meyer, G. Alonso, D. Fauser, D. Kossmann. Crescando. In *Proceedings* of the 2010 *ACM SIGMOD*, Pages 1227-1230.

[17] InfiniBand. *http://www.infinibandta.org/*, March 2013.

[18] R. Jain. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. *Wiley-Interscience*, April 1991, 556-561.

[19] MySQL. *http://www.mysql.com/*, March 2013.