Cloudberry pilot report

BioCAM – Big Data Router for Real-Time On-the-fly Business Intelligence and Analysis

Mikael Sundström, Josephine Åhl: Oricane AB, *{micke, josephine}@oricane.com*

Anders Torger, Johan Karlsson Rönnberg: Xarepo AB, *{anders.torger, k}@operax.com*

Olov Schelén: Luleå University of Technology, *olov.schelen@ltu.se*

Introduction

Analysis of business data has been vital for decades but the volumes of data available today are growing on an unprecedented scale. Turning large and complex sets of data into meaningful information can be time and resource consuming, limiting the possibility to act fast.

As an example, in the retail market most of the reporting systems currently used are designed to analyze *historical data* stored in data warehouses. Data from cash registers at thousands of retail stores spread across different continents, representing sales up to millions of different products is typically analyzed weekly, or in the best case on a daily basis.

Common analysis tools interfacing directly to the data warehouse are limited by its I/O performance, making real-time analysis nearly impossible. However, the transmission of data to the data warehouse is analogous to Internet packets sent through a router or a firewall, where the data can be intercepted for real-time analysis, resulting in aggregated data records.

Reporting in real-time enables instant reaction to different events such as micro trends, marketing and campaigns in order to immediately adapt production after demand. As an example, there may be a certain product where the demand suddenly increases unexpectedly – an event more common today with the broad use of social networks. In that case, the supplier either has to over stock in order to be ready for the unexpected or may run out of supply with the risk of losing revenue. By having access to real-time data, the supplier can stock up in a minimalistic fashion and also be able to ramp up production when an increase in demand is detected.

This report presents our work on an in-memory data collection and analytics solution called BioCAM, which includes support for persistent real-time data storage (RTDS), and a proof of concept application (demonstration) in a large scale retail scenario. The work on integration and proof-of-concept development has been carried out as a pilot project within Cloudberry Datacenters at LTU. The basic technologies of BioCAM and RTDS were developed previously to the project.

Motivation and Related Work

Distributed big data analytics is an established field of research rich of previous work. A common approach is to use MapReduce (e.g., Hadoop), which for each batch task activates one-master and multiple distributed slaves. An objective is to increase performance and scalability by organizing computations into long I/O streaming operations that make the best of the total bandwidth of many disks in a cluster. Such approaches are primarily designed for batch processing over large data sets collected beforehand. In real-time big data systems the overhead and delay of first collecting and distributing the data to a cloud repository and then re-distribute some centrally triggered analytics task across the network for execution at a number of slaves, would not be an adequate solution. There are a number of solution proposals that make the MapReduce approach more suitable for real-time processing. However this is primarily to improve the performance slightly, while still using well known traditional big data analytics techniques. The solutions still do not meet all requirements for real-time analytics.

To fill this gap, many solutions have evolved to leverage real-time computation on big data streams [3]. Yahoo S4 [6] , Aurora [10] and Storm [8] are similar solutions that tackle the streaming big data problem. While there are some minor technical and naming differences, their approaches can be narrowed down to one common approach. Data is fed into the system as an infinite sequence of tuples which are transferred to processing nodes (worker nodes). The processing nodes can then consume the data tuples and publish some results or produce more data streams to be processed and/or aggregated by other nodes. In addition, data tuples from intermediate streams are usually buffered in the distributed memory of the worker nodes for minimum latency during data access. Probably the most popular package in this category is Apache Storm [http://storm.apache.org/]. It is a generic package with many use cases (e.g., real-time analytics, on-line machine learning, continuous computation, distributed RPC). Storm is in a benchmark clocked at over a million tuples processed per second per node, and is scalable and fault-tolerant.

The objective of our work has been to address the solution space of real-time analytics and data collection on the fly by using a new approach. Important requirements have been to be able to insert the solution with minimal integration. It is also a well known fact that in real-time systems you should avoid network delay and round-trips in order to improve response times. We have therefore explored how to co-locate collection and analytics at the same node, while maximizing performance in each single node. With this approach, in some cases it is possible to handle an unprecedented number of operations for collection and analytics in a single node. We prove in our solution that it is possible to reach such performance that analyzing all data from a super size business scenario can be done in real-time by using one single business analytics router. Although large scale analytics can be provided in a properly located single node in our solution, we are not at all against distribution. On the contrary, distribution should as always be used as a scalability option, for redundancy/availability, or because retrieving the data in one point is simply not feasible. Our solution can therefore be scaled out by performing real-time analytics locally and then streaming the results on the fly from local nodes to nodes that perform meta analytics (a k a higher order analytics). An important part of our work has been to support our in-memory data engine with persistent real-time storage. Considering the typical data speed that can be handled by the in-memory engine, this is a quite challenging task.

BioCAM big data analysis methodology

The purpose of data mining for business intelligence applications is to analyze collected data from business transactions to achieve an understanding of what has happened in the past. There can be many reasons for performing this kind of analysis and some examples are: (a) To investigate the impact on sales after launching a new commercial, (b) To investigate the difference in sales across product segments and geographical markets, (c) To investigate variations in sales over different seasons.

For a small business operation (with low number of transactions and/or small number of different products) this kind of analysis can be done quite easily using brute force methods but for large business operations, large (and costly) appliances are typically required and still it is quite common to perform this kind of analysis at a low time granularity (weekly basis).

Problem Definition

Data collected typically consists of records where each record have a number of fields. Some of these fields contain real number values (can be integers as well), some fields contains text values, and some fields contain time stamps. In addition to these basic field types we have *selector fields,* which are *lists* of records of the same type from which we can select individual sub-records referred to by a unique *tags*. There could be other fields as well but these major types are sufficient to understand the problem solved by BioCAM.

Across these three types of fields we can define what we call *class fields* which can be used to separate records into different *classes*. There are three kinds of class fields, *explicit, implicit* and *synthetic*.

By *explicit class field* we mean a field where the value stored in the field can be used directly for classification. For example *color* (red, blue, white, yellow), *country* (Sweden, Finland, Norway), and *vehicle type* (bicycle, motorcycle, car, airplane). Explicit class fields are typically associated with text fields.

By *implicit class field*, we mean a field which is typically defined by either a time stamp or a value and where the class is defined by a range. For example all transactions that occurred during the same day is in the same class or all transactions where the sales price is in a certain range is in the same class.

By *synthetic class field*, we mean a field which is not present in the record originally but rather derived from values in other fields. For example a product which is sold in four colors and the records keeps track of sales across individual colors, then the *most sold color* can be generated as a synthetic class field.

In addition to class fields we can use selector fields to separate *partial records* into different classes.

We refer commonly to fields that are neither class fields nor selector fields as *value fields*.

The primary purpose of business intelligence tools is to have the data broken down in aggregated form and being able to generate comprehensible summary reports rather than looking at each individual record. We define a *breakdown* as a layered tree structure where each layer corresponds to a class field. Each class field can occur at most once in each breakdown and not all class fields needs to be present.

While the breakdown defines a hierarchy of class fields, or *classes,* it does not in itself constitute sufficient information to represent a summary report. The first layer of the breakdown represents the *root node* in a multi-branch tree structure and each sub-tree directly below the root node represents a value of the class field, or *class*, of the class field associated with the root node. The breakdown is defined recursively in this fashion until reaching the *leaves* associated with the last class field.

The records themselves constitute the bottom of the breakdown and are located below the leaves. For example, if we use the *vehicle type, country,* and *color*, previously mentioned, as breakdown, we will have a root node with *bicycle*, *motorcycle, car*, and *airplane* as children. Each node at the second layer will have *Sweden, Finland,* and *Norway* as children and each node at the third layer will have *red, blue, white*, and *yellow* as leaves. The sequence of classes encountered when traversing from the root to a leaf is called a *path* and the set of paths of a given breakdown corresponds to a partition of the set of records. That is, each record is accessible via exactly one path. For example, the path {car, Finland, blue} reaches all records representing blue cars in Finland.

A breakdown in itself is does not make sense without any aggregates. To obtain an *aggregate,* or aggregated data, in the leaves of the breakdown some aggregate function is applied to all records associated with the leaf and the resulting aggregate is stored in the leaf. Similarly, to compute the aggregate of a node some aggregate function is applied to all aggregates of the children nodes and the resulting aggregate is stored in the node itself. To compute all aggregates this is, in effect, performed bottom-up recursively until the root aggregate has been computed.

The *aggregate functions* used can be complex and involve several fields from underlying records or very simple and only include a single field. An example of a simple aggregate function is to just accumulate the values of a certain field in the parent and then, for example to generate sales reports broken down in different ways.

The BioCAM Engine

Having described the overall framework it is now time to describe the three main problems that needs to be solved and how these are solved by BioCAM in detail.

Data Record Representation

The first main problem is to efficiently represent the records in a way that minimizes storage of redundant information and at the same time enables extremely efficient construction of breakdowns.

Basic Representation

In the basic representation, we use the same number of bits (typically 32 or 64) to represent each field. This means that we can represent 32/64-bit integers (signed or unsigned) and 32/64-bit real numbers. Time stamps are essentially represented as the number of time units elapsed since a certain "start of time" and are mapped to integers. In Unix Time, a time stamp is represented as the number of seconds that have elapsed since midnight January 1, 1970.

Text fields can vary quite heavily in length and in many cases text fields represents some kind of property, and thus corresponds to a value of an enumerated type in a programming language, whereas some text columns (typically only one or a few) represent, or corresponds to, an identifier of the record itself. Replications are extremely common for property text fields and these must therefore be stored in a dictionary and represented by an integer in the records. For simplicity, the same approach is used for identifier text fields since it is not necessarily known before hand which kind it is. As an option, the text fields can be compressed further using some available text compression algorithm to obtain a dictionary of compressed/encoded strings, as opposed to clear text strings, thus reducing memory requirements for the dictionary data structure. Any type of dictionary data structure can be used to represent the dictionary but we prefer some kind of compressed trie to achieve fast access and low memory footprint.

In addition to the above mentioned, there can also be text fields containing free text which may be arbitrarily large. When present, such text field are compressed using some available text compression algorithm and represented in the record by a reference to the compressed text.

Advanced Representation

In many cases, the actual number of bits required to represent different fields varies a lot. This can be exploited to obtain a reduction in memory consumption, increase locality of accesses and improve the speed of computation.

To achieve the best result, we combine two different approaches where the selected approach depends on whether the field is a class field or not.

For non class fields (i.e., numbers and timestamps), there are some variations with respect to how many bits that are actually necessary to represent the number. From a point of view of compression, we would like to use exactly as many bits as required for each field. On the other hand, number fields are heavily accessed during computations and should therefore be nicely aligned to reduce computation overhead. As a compromise, we use a scheme where number fields are represented by 8, 16, 32, or 64 bits. We have not yet encountered a use case where it is necessary to extent this beyond 64 bits but extending to any multiple of 32 or 64 bits, depending on hardware architecture, is trivial. The non class part of each record is represented by a number of arrays of fields stored in a *packet array* that occupies consecutive memory locations starting with the array of fields that requires the largest representation, followed by the second largest and so on. In order to keep track of how each number/time stamp field is represented and how to access each individual field, the we record for each field which group (8-bit, 16-bit, etc.) it belong to and also the index of the field within its group. By also keeping track of the number of fields in each group we can easily compute the location of an individual field of a record as

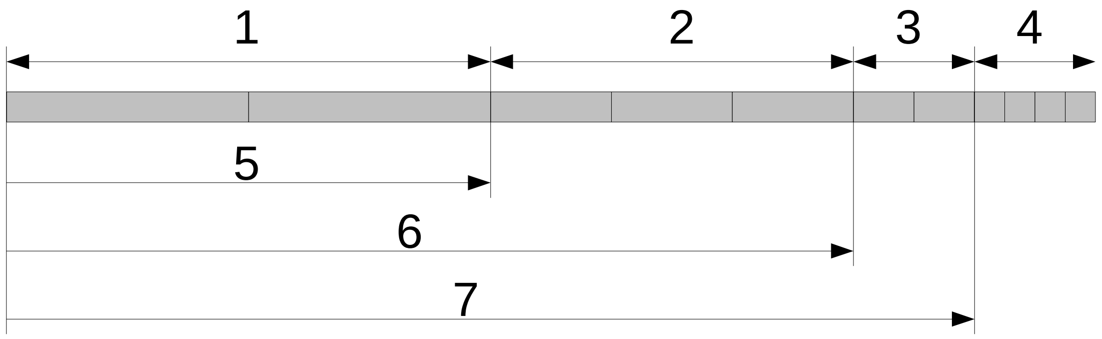


Figure 1

an offset from the start of the array containing the fields that requires the largest representation. While this book keeping may appear complex at a first glance it is not required for each individual record since all records are represented and mapped in memory in the same way. Hence there is a global scheme for mapping records to memory across the whole data base.

In *Figure 1* we show an example of a packed array containing group fields of different sizes. There are two 64-bit fields (1) at offset 0 bits (offset is with respect to the base of the array), three 32-bit fields (2) at offset 128 bits (5), two 16-bit fields (3) at offset 224 bits (6), and four 8-bit fields (4) at offset 256 bits (7).

Figure 2 illustrates the global record mapping scheme for the packed array from *Figure 1*.

64	2
32	3
16	2
8	4

Figure 2

For class fields we perform an analysis for each individual field to find the minimum number of bits required for representing the field. This is achieved by counting the number of unique values that occurs in that field and taking the ceiling of the logarithm with base two of that number. For example, if there are 75 different values we must represent the field using 7 bits (0..127). Depending on the expected dynamics of the data base we may or may not choose to use one or a few extra bits to handle an increase in variety without having to reconfigure the field representation.   The class field values of each record are stored tightly packed in memory and constitute a *master key* for said record. While the master key is preferably embedded in an unsigned integer if it is reasonably small, it is essentially an array of bits and can thus be stored in a quite flexible manner. However, similarly to non class fields, there need to be a global scheme for mapping records to individual parts of the master key that also takes into account the storage of the master key.

In *Figure 3* we show a 64-bit master key representing a 17-bit class field (1), an 11-bit class field (2) and a 16-bit class field (3). The layout of the class fields takes into account the machine word (4) size, which in this example is 32-bit, to avoid that any class field is stored across a machine word boundary

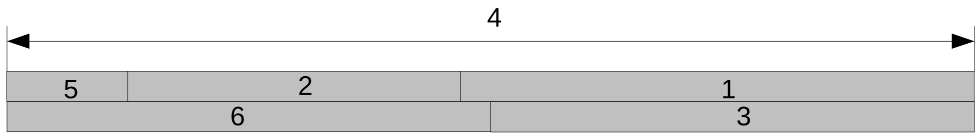


Figure 3

as this could potentially lead to a performance penalty during computation. Also note that there are four (5) and sixteen (6) unused bits in each machine word respectively.

Breakdown Representation

The second main problem is to efficiently represent breakdowns with minimum memory overhead and at the same time facilitate efficient traversal of the tree structures represented to enable fast generation of reports.

A breakdown is essentially built from an ordered list of class fields where the first field represents the top level, the second field the second level of a tree structure and so on. Since the records may be quite large and there may be several breakdowns concurrently in use it is not possible to move records around when constructing breakdowns. Therefore, we start the breakdown construction by constructing an array of handles. For each record there is a handle and the handle contains a reference to the record it is associated with. Furthermore, the handle contains a *slave key* which is a subset of the master key containing only the class fields included in the breakdown. The slave key is represented as an unsigned integer large enough to contain all fields required for the breakdown. Since the slave key is crucial for breakdown construction we map the fields included in the breakdown such that the last field occupies the least significant bits of the slave key, the next last field occupies the next unused least significant bits and so on. If there are unused bits of the slave key when all fields have been stored these are zeroed. By this key mapping we can start the construction of the tree structure by sorting the handles according to the slave keys. This will achieve a sorted array of handles where handles with identical slave keys are grouped together. If is necessary to preserve the order between handles with identical slave keys the sorting algorithm has to be chosen accordingly and if some articular order needs to be imposed an additional class field selected to impose said order can be added as the last class field of the breakdown configuration.

Figure 4 (a) illustrates the records stored in consecutive memory locations (1), the references/pointers

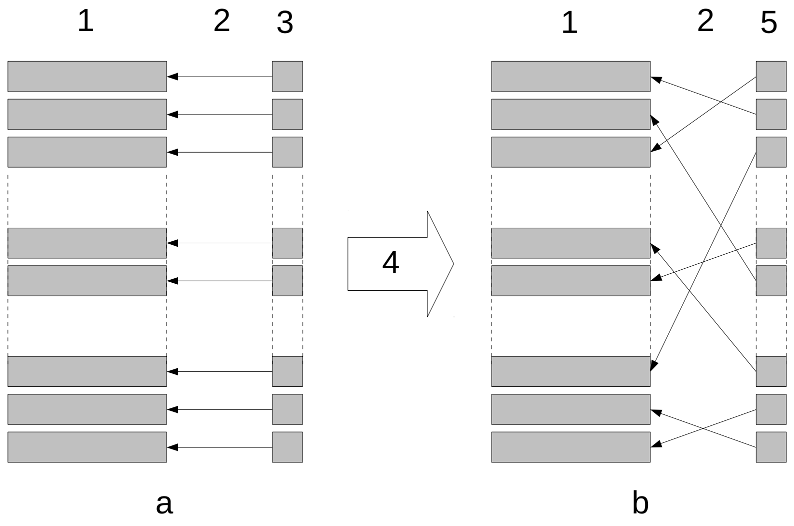


Figure 4

from handles to records (2) and the corresponding unsorted array of handles (3). In *Figure 4* (b) we show the resulting sorted array of handles (5) after sorting (4) the handles in increasing order with respect to the slave keys (4).

After completing the sorting step, the rest of the breakdown is built in the same array as the handles after the handles, which represents the bottom of the breakdown, such that all leaves are stored in the array locations directly following the handled, all parents nodes of the leaves are stored in the array locations immediately following the leaves and so on until reaching the last location in use which contains the root node.

The construction can be explained as follows. We keep track of the current *depth*, which initially equals the number of levels of he breakdown, the *current offset*, which initially refers to the offset of the first handle, and the *next* and *free* offset which both initially refer to the offset of the first leaf to build. Across the construction, next offset refers to the first location of a node to be constructed that is not part of the current level under construction whereas free offset refers to the location where to construct the next node. Nodes are represented by a chunk structure that contains a *base*, which is the offset to its first child (or first handle if it is a leaf), key, which is a copy of the slave key, *size*, which is the size of the node measured in number of children (or handles if it is a leaf), and *density*, which is the number of handles in the sub-tree rooted at the node. Each chunk also has a reference to the offset of its parent node. It is assumed that handles can be accessed in the same fashion as chunks. The algorithm consists of three nested loops where the outermost loop carries out the entire construction bottom up level by level, the intermediate loop executes construction of one level by looping through the nodes of that level, and the innermost loop performs construction of a single node (or leaf) by processing the children nodes (or handles) and attaching the parent node to these while updating the *size* and *density* fields of the parent. Upon initiation, the handles occupies locations 1 to *number of handles*, and *next* and *free offset* are both set to *number of handles* + 1 whereas the *current offset*, which is the offset of the current child/handle to be processed, is set to 1. In the outermost loop, the *depth* is decreased for each round and *next offset* is set to *free offset*, after completing the round, and the loop carries on until the last round, where the depth is zero, is completed and *next offset* - 1 is the location of the root of the breakdown. In the intermediate loop, *free offset* is the location of the node to be constructed whereas *next offset* represents the first offset of the node at the level above the current level in construction. Hence, the last children node/leaf/handle of a node at the current level under construction is located at *next offset* - 1. Before entering the innermost loop, the *current offset*, which is the offset of the first child, is stored in the *base* field of the current node, located at *free offset*, and the *key* of the first child is copied to the *key* field of the node. After exiting the innermost loop, *free offset* is increased by one. To determine when to exit the innermost loop, the *key* field at the *current offset* is compared to the previous key field while taking into account only the parts of the keys that are relevant for the current level (relevant parts of keys decreases as the construction commences upwards in the breakdown) as well as the *current*, *free*, and *next offset* variables.

As for the comparison between keys with respect to depth we use a mask to omit parts of the key corresponding to levels already constructed in the comparison. That is, for the bottom level, we use the whole key. For the next level we use a mask that zeros out the parts of the key that contains the last field of the breakdown and so on. Preferably, these masks can be prepared in advance and stored in an array where the depth is used directly to access the correct mask during comparison thus improving performance.

In *Figure 5* we show an example of a breakdown consisting of four levels. To easily distinguish between the levels we use a darker shade of gray for the root node and then lighter shades of gray as we move down the tree towards the leaves. The references/pointers from parent nodes to the first children node of the respective parent is shown in (1) whereas the references/pointers from each node, except the root, to its parent node is shown in (2). In (3) we show the *size* of each node measured in number of children. References/pointers can be stored in nodes as either memory addresses or integer offsets. Also note that this information is sufficient to determine for any given node whether it is a leaf or not and whether it is the root or not. Furthermore, if the node has siblings it is straight forward to determine if it is the first or last sibling and if not move to its previous or next sibling respectively. If the node has children, we can directly access a children node with a given index (assuming it is in range with respect to the node size). Finally, since the representation is *implicit*, or *in-place*, it is extremely conservative with respect to memory consumption and also supports fast traversal and processing.

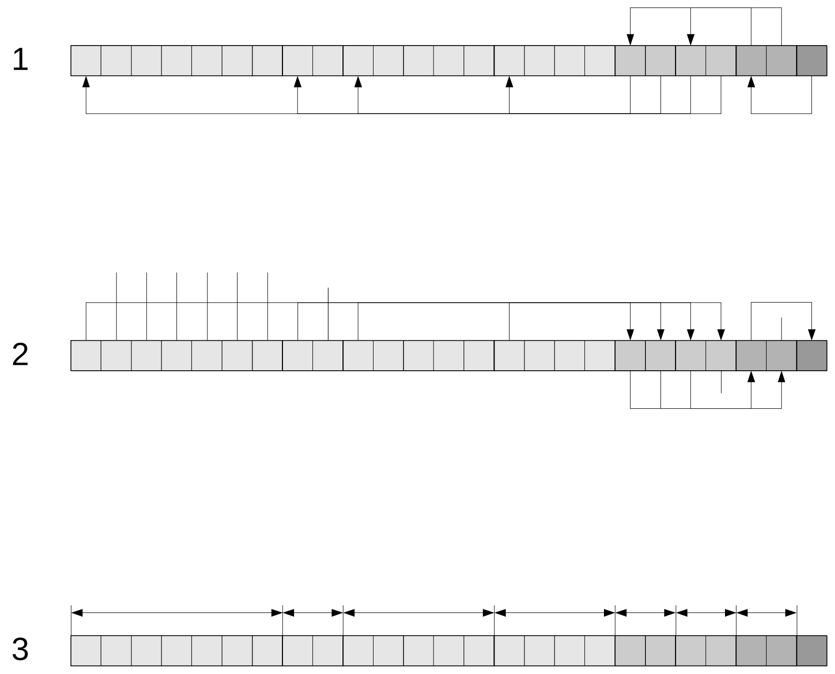


Figure 5

Variations of the above scheme includes saving memory by not storing pointers to parents in children and instead recomputing the aggregates for the whole tree after updates as well as emulating slave keys by using a function to extract fields directly from records instead of using the preprocessed slave key for sorting.

Update Management

The third main problem is to manage update of records to minimize the impact on existing breakdowns as well as minimize the computations required to update reports to reflect the changes after an update.

Most updates does not affect class fields. Therefore we optimize the update management algorithm to handle updates (changing absolute value, decreasing, increasing) of value fields that does not affect class fields. If there are no active breakdowns, an update simply means to change the contents of a record and nothing else is affected by the update. However, if there are active breakdowns, changing value fields typically affects one or more aggregates further up in the tree if any of the altered value fields subject to aggregation.

One strategy to minimize the computational overhead resulting from updates is to delay re-computation of aggregates as much as possible to the point when there is a request for reading the current value of the aggregate and then re-compute the aggregates of the nodes that are absolutely necessary. The advantage of this strategy is that the maintenance work resulting from a single update is minimized and independent of the number of breakdowns/aggregates affected by the update. However, the disadvantage is that whenever an aggregate is re-computed, all records involved needs to be accessed resulting in essentially the same amount of work as reconstruction of the aggregates from scratch.

Another extreme strategy is to update all aggregates affected by an update immediately after the update. This approach is very good if the frequency of reporting (reading aggregates) is extremely high compared to the frequency of updates. However, it requires that the path from handle to leaf and then all the way to the root node is updates for all affected aggregates, causing a very high computational overhead on each update.

To avoid the drawbacks of these two extremes, we propose a strategy where the each update also updates the parent of the handle (i.e., the leaf of the implicit tree structure for each affected breakdown/aggregate). For each field, it is therefore necessary to keep track of the breakdowns affected by changing the field and it is also necessary to maintain a mapping from record and breakdown to the handle of the breakdown associated with the record. Otherwise, the leaf node to be updated, when updating the record, can not be located. The leaf node is then inserted into a task queue data structure associated with the aggregate if it is not already present in the task queue. When issuing a report which requires re-computation, nodes from the task queue are dequeued and their parent nodes are updated and inserted (if not already present) in the same task queue. The re-computation of the aggregate is concluded when the root node is extracted from the task queue.

In the rare cases when an update affects one or more breakdowns, all affected breakdowns are marked invalid and re-constructed from scratch the next time a report is requested which is associated with an invalid breakdown.

A proof of concept in real-time business analytics

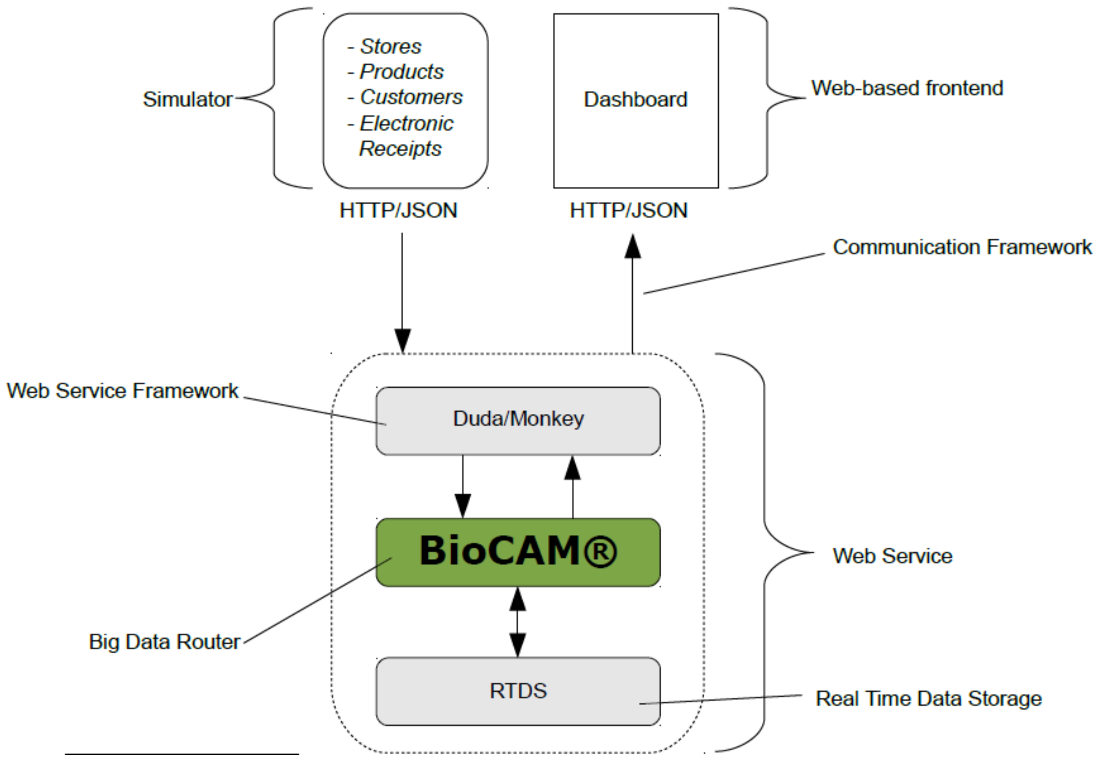


Figure 6

Our Proof of Concept (PoC) [4] was built to provide a platform for technical evaluation of the solution and a demonstrator suitable for user evaluation. The back-end is based on the real technical solution for deployment in various commercial cases, while the front end and the GUI has been defined for a particular simulated scenario. We have built a simulator that simulates a global retail chain with cash registers located in thousands of stores across the globe. The simulator produces actual cash register records in real-time, which are collected and reported to the BioCAM engine every second, and analyzed on the fly with intervals of 10 seconds. To facilitate analysis in our PoC, the BioCAM big data router engine is embedded in a lightweight high performance HTTP server and made available through a JSON web service. A persistent storage API (RTDS) is integrated to provide the BioCAM with persistent storage to make stateful restarts possible, and in the longer term high availability with hot standby.

A JSON web service is a trade-off between efficiency and ease of integration using typical high level development frameworks. While a custom binary protocol would be more efficient, it would also require more from the integrators. A JSON web service is easy to understand and integrate while being relatively compact and efficient.

To simulate a global retail market use case a simulator is built than feeds the BioCAM with live real-time sales information via the JSON web service. Real-time reports are then ordered and fetched from BioCAM (through the same JSON web service) and presented on a live dashboard web interface.

The system demonstrates a realistic scenario of how real-time analysis of big data flows can be presented, using simple standard protocols for integration and requiring minimal hardware resources.

Figure 6 shows the PoC setup.

Monkey/Duda web service framework

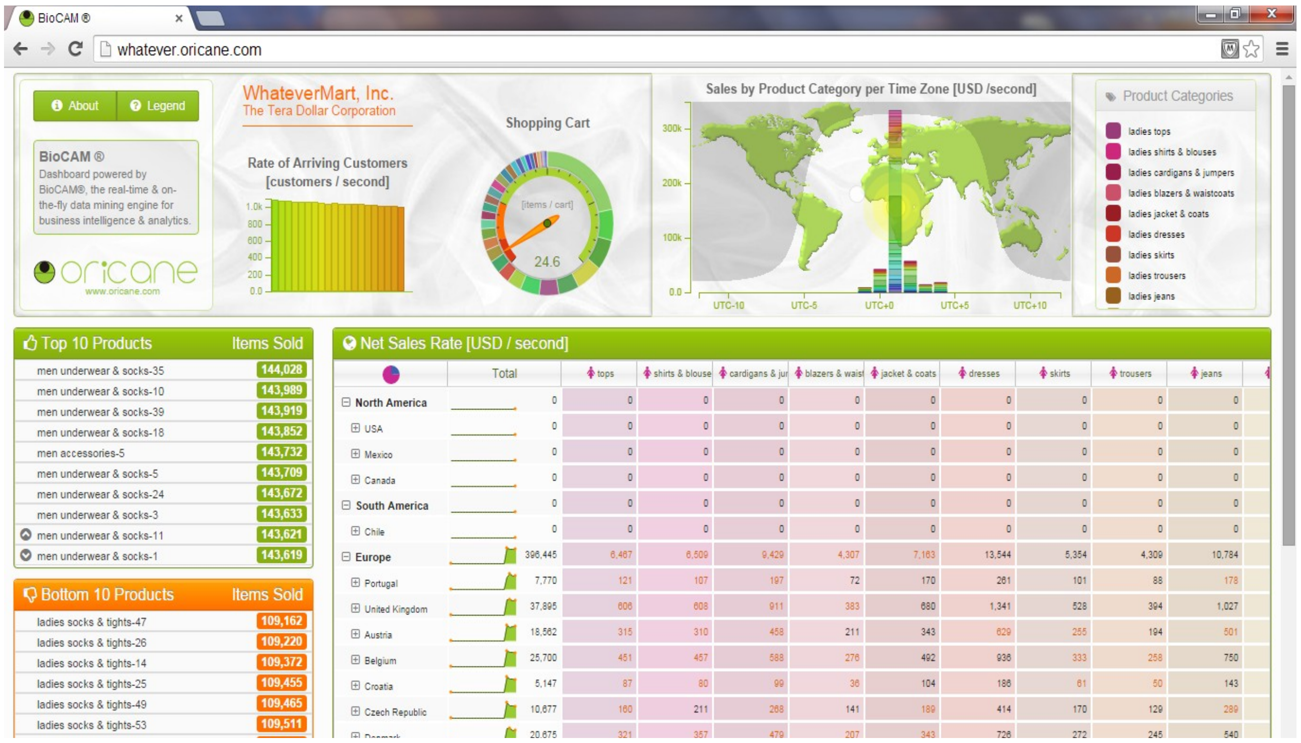
Monkey is an open-source HTTP server designed to be highly scalable while having low memory and CPU consumption. As such it is a perfect match for the energy efficient software concept BioCAM represents, and was thus chosen as the server for exposing the BioCAM web service.

Duda is an extension to the Monkey web server which provides a C API to make embedded web services. This was used to create the BioCAM web service.

When integrated the BioCAM engine is loaded at start-up as a dynamic module and runs in the same process as the Monkey web server. This tight native integration makes a small energy-efficient footprint.

BioCAM web service

The web service is made as a thin layer on top of the BioCAM native API, meaning that it provides access to all its features. Custom databases can be created and filled with data, and then reports can be generated and fetched. This means that the web service can be used in any type of analysis scenario where the BioCAM feature set is suited, not just the PoC scenario.



BioCAM ®
Dashboard powered by BioCAM®, the real-time & on-the-fly data mining engine for business intelligence & analytics.

Product Categories
ladies tops
ladies shirts & blouses
ladies cardigans & jumpers
ladies blazers & waistcoats
ladies jacket & coats
ladies dresses
ladies skirts
ladies trousers
ladies jeans

Top 10 Products	Items Sold
men underwear & socks-35	144,028
men underwear & socks-10	143,989
men underwear & socks-39	143,919
men underwear & socks-18	143,852
men accessories-5	143,732
men underwear & socks-5	143,709
men underwear & socks-24	143,672
men underwear & socks-3	143,633
men underwear & socks-11	143,621
men underwear & socks-1	143,619

Bottom 10 Products	Items Sold
ladies socks & tights-47	109,162
ladies socks & tights-26	109,220
ladies socks & tights-14	109,372
ladies socks & tights-25	109,455
ladies socks & tights-49	109,465
ladies socks & tights-53	109,511

Net Sales Rate [USD / second]

	Total	tops	shirts & blouse	cardigans & jur	blazers & waist	jacket & coats	dresses	skirts	trousers	jeans
⊟ North America	0	0	0	0	0	0	0	0	0	0
⊞ USA	0	0	0	0	0	0	0	0	0	0
⊞ Mexico	0	0	0	0	0	0	0	0	0	0
⊞ Canada	0	0	0	0	0	0	0	0	0	0
⊟ South America	0	0	0	0	0	0	0	0	0	0
⊞ Chile	0	0	0	0	0	0	0	0	0	0
⊟ Europe	396,445	6,467	6,509	9,429	4,307	7,163	13,544	5,354	4,309	10,784
⊞ Portugal	7,770	121	107	197	72	170	261	101	88	178
⊞ United Kingdom	37,895	606	608	911	383	680	1,341	528	394	1,027
⊞ Austria	18,562	315	310	458	211	343	629	255	194	501
⊞ Belgium	25,700	451	457	588	276	492	936	333	258	750
⊞ Croatia	5,147	87	80	99	36	104	186	61	50	143
⊞ Czech Republic	10,677	160	211	268	141	189	414	170	129	289
⊞ Denmark	20,675	321	357	479	207	343	726	272	245	540

Figure 7

Currently there are no security or audit features implemented (user authentication, access lists etc), which would be required in a production environment.

Retail market simulation client

The retail market simulator simulates actual cash register records in real-time located in thousands of stores spread over the globe.

When connecting to the BioCAM web service, it sets up a database to store records (if not already existing), and then starts to simulate sales by inserting updates accordingly. The database columns are product name, category, size, color, shop identifier, country, country region, time zone, price, number of items sold and number of items still available in the shop.

The simulator has an inventory of realistic products that can be found in a typical global retail market. The default simulation is about 1500 shops world-wide with a product inventory of 2200 different products in about 30 categories, which is based on actual public data from a well-known fashion retailer, but all these can be tuned with parameters.

The simulation client is implemented in C, employing a multi-threaded architecture for increased scalability on multi-core platforms. As it keeps state of all the shops to generate realistic traffic it is memory-intensive but does not require much processing resources to generate its traffic. Thus it can run on any modern desktop computer, preferably with at least 8 gigabytes of RAM.

Dashboard

The web-based dashboard (Figure 7) connects to the BioCAM web service and specifies which real-time analysis reports that should be generated, then continuously retrieves the results and presents them in a live dashboard.

It is implemented with web development techniques such as Ajax to get live updates and a modern desktop application feel.

Real-time Data Storage (RTDS)

The RTDS is a persistent storage engine designed for telecom grade high availability and performance requirements. Rather than running as a stand-alone process it is linked together with the user application. It streams memory state updates to a self-rotating transaction log, which can optionally be mirrored to a remote node to support immediate fail-over (hot standby). It is not by itself a database, but can for example serve as a persistent storage back-end to an in-memory database such as BioCAM.

Rotating transaction log

The high availability requirements include supporting hot standby and robustness so it can recover if failure or power loss occurs in any state. A proven design to achieve this is to store data in a transaction log; instead of just dumping a raw image of the current memory state, the engine writes a log file with all changes that are made to the data (i.e., insert, delete and modify of objects) in RAM. This transaction log can then be replayed continuously in the standby node producing an exact copy of the state up to the last transaction, ready for immediate fail-over.

The inherent problem with a transaction log is that it will grow indefinitely and you can thus run out of space on disk. Traditionally this is solved by running a separate process that traverses the transaction log, builds a copy of the state and then dumps a “checkpoint”. This process is however problematic from a scheduling and memory consumption perspective.

RTDS employs a new solution, by having an internal background thread that traverses all state in memory and writes update operations into the log in parallel, which means that when you have a full traversal in a log file you can restore the full state from that single log file and you can start on the next file. This way RTDS does not need more than two log files, one with the complete state and one current which contains the changes on top of that complete state. When traversal is finished the files can be rotated.

For this to work during heavy load RTDS has a carefully designed throttling mechanism to avoid starvation of the traversal thread, while at the same time minimize latency. The traversal thread runs at highest real-time priority (will preempt other threads on a single-core platform) but only in very short bursts.

To restore state at startup, the two logs are traversed. For large and/or fast-changing states startup time will be time consuming as the logs may be large. For a high-performance system with high loads startup time can be in the 10 second range. Restoring from a plain memory dump would be much faster, but you would then not have the transaction safety or data readily arranged for mirroring. In an active+hot standby arrangement the peer node will be active during the startup period.

High availability features

RTDS supports the typical high availability use cases. For example you can replace both hardware nodes in a running system without downtime. Shut down standby node, replace hardware, start up and RTDS will synchronize with the active node (while the active node is servicing requests), and when finished synchronizing you can switch over and service the other.

RTDS also supports data format changes in runtime, that is when you upgrade the software which may have new object formats, and then run and synchronize with an older version of the software. Certain rules in format extensions much be obliged though.

All these features are designed for supporting a software with very high availability.

Real-time features

RTDS is intended for “soft real-time” requirements and to run on a standard operating system such as Linux. Soft real-time means that low latency is a requirement, but an occasional hiccup does not mean failure. Note that the standard operating systems themselves cannot meet “hard real-time” requirements, so you cannot claim to have hard real-time support on these platforms. Response time variations and latency increase considerably when running in a virtualized environment too of course.

The real-time design in RTDS is manifested through many lock-free or contention free solutions inside. For example, getting and committing a buffer to the transaction log is lock-free, so multiple threads in the user application can get buffers in parallel without blocking.

Today the low-latency aspect of real-time is overkill for most applications, and often the application is deployed on a virtual platform which in itself has high latencies. However, the other aspect of real-time is to keep track scheduling to avoid starvation or uncontrolled queue growth, which still is important. This is called “overload control” and in RTDS this means not generating more transaction log than the disk (and possible standby node) can receive, and make sure transaction log can rotate before growing too large.

Naively designed overload control with simple blocking at queue full can lead to queue flapping (queue is either full or empty) which in a networked environment can propagate and make the whole system unstable. RTDS instead monitors queue length and introduces tiny sleeps at suitable intervals (with a random factor) to make a soft queue filling behavior, while minimizing the introduced latency.

Energy efficiency

RTDS is more lightweight than a typical storage back-end and is thus more energy efficient (i.e., less CPU cycles required to store and mirror data). Another aspect is that through its real-time properties and overload control a system can be dimensioned with less hardware and still provide safe stable behavior during periods of high loads.

Data layout

RTDS is designed to support applications that store their data in RAM in possibly millions of smaller objects, which can be linked to each-other like a network database. It is not intended to support larger objects, because this is not wise from a latency standpoint, as committing an object that occupies multiple megabytes would fill large parts of the transaction queue and could thus cause longer latencies and impair overload control.

The application uses RTDS allocation functions (rather than the standard “malloc”) and puts the data in structures directly in this allocated space. RTDS will internally store a copy of the data. That is, RTDS will double the memory requirement of the application, which can be significant. RTDS could be designed to not store a copy, but then it would be much more difficult to use and detect errors. By having a copy RTDS can detect how the data is modified and makes the data much safer from accidental changes. It also makes the background traversal transparent which can be done without locking and without any extra API calls for the user (which otherwise would need to lock the objects before modifying them).

Note that as the copy is stored in the same process it's not fully protected. A pointer bug could cause accidental change of the RTDS private copy and thus causing corrupt data being written to the transaction log. The risk of that type of bug is considered low enough though, and fits the general context of an in-memory database or other applications this type of storage engine would be used.

Alternative methods of persistent storage

A common method to add persistent storage to an in-memory application is to integrate with a standard SQL database. The application will either define its data structures directly in SQL or use some layer on top. While the real-time features comparable to the ones delivered by RTDS cannot be obtained, the high availability aspects can be implemented if the proper database, hardware and configuration is employed. At a lower level these databases usually use a transaction log based design, just like RTDS.

The advantage of this type of standard database integration is that you employ a since decades proven model and get a more flexible data back-end concerning integration with other systems (thanks to SQL). The disadvantages are that it can be more complicated to integrate, costs more (both in terms of hardware and software) and is less efficient in terms of hardware resources (less energy efficient), both due that it consumes more CPU cycles in base load, but also may require more over-capacity to behave well during overloads.

The other alternative for persistent storage is to go the other direction towards even less complexity. The simplest and most lightweight way is to just dump a raw image of the RAM. You can even use memory mapping (mmap() call on Unix platforms) and let the OS handle it automatically. In terms of energy consumption this will be the most efficient way, and you can have very fast startup times.

The drawbacks are that it lacks transactional safety, and data corruption can easily happen with bugs or power loss type of failures. As you do not get updates grouped in transactions it can also be difficult to mirror to a hot standby.

RTDS integration with BioCAM API

The purpose of integrating RTDS with the BioCAM API is to provide a persistent storage option, so a BioCAM-powered application can be restarted and get full state restored from persistent storage. Through the high availability design of RTDS the application will handle unclean shutdowns (crashes, power loss etc) and still restore a consistent state, which would not be the case if a simpler memory mapped backing has been employed.
In the longer term RTDS can also provide mirroring to a hot standby, but those features has not been enabled in the current integration. Note that a true hot standby requires some integration at the application level too, as all data structures need to be kept up to date, not only the persistently stored data.

The BioCAM engine was initially designed for high speed in-memory processing, but it was not designed to compete with traditional in-memory databases that would also have a hook for persistent storage. Consequently, there were at first no hooks for persistent storage. However, the RTDS API is well-suited to handle generic structures with data interconnected with links (pointers), which is the way data is typically natively laid out in an application built with a procedural language like C, and BioCAM is no exception to that. Only a few minor changes in the BioCAM data layout were required to be compatible with the RTDS API. The changes and additions to support RTDS became simple enough so that they were made as a compile-time option (removing the optional RTDS sections from the code). With RTDS disabled, the BioCAM engine runs in the original memory-only mode with no significant changes to data layout.

On top of the persistent data lies the actual BioCAM data structures that organize the data. These structures are not stored persistently, but are instead reconstructed at startup. An alternative would be to also store this persistently, which would likely lead to faster startup times. The data structures are however very complex objects which would make integration with RTDS or any other persistent storage solution complex. Also, while leading to a somewhat quicker startup time the total overhead would be higher as you would need to store both data structure and object data in runtime instead of only the latter.
The BioCAM API could be kept unchanged, except for adding RTDS startup and shutdown calls (returning the loaded BioCAM engines) and one extra parameters to name the BioCAM engine when created, so if there are multiple engines each one can be identified when reloaded.

References

1. WhateverMart, a BioCam Proof-of-Concept http://whatever.oricane.com/
2. Cheng-Zhang Peng; Ze-Jun Jiang; Xiao-Bin Cai; Zhi-Ke Zhang, "Real-time analytics processing with MapReduce," *Machine Learning and Cybernetics (ICMLC), 2012 International Conference on* , vol.4, no., pp.1308,1311, 15-17 July 2012
3. Osman, A; El-Refaey, M.; Elnaggar, A, "Towards Real-Time Analytics in the Cloud," *Services (SERVICES), 2013 IEEE Ninth World Congress on* , vol., no., pp.428,435, June 28 2013-July 3 2013
4. Tao Zhong; Doshi, K.A; Xi Tang; Ting Lou; Zhongyan Lu; Hong Li, "On mixing high-speed updates and in-memory queries: A big-data architecture for real-time analytics," *Big Data, 2013 IEEE International Conference on* , vol., no., pp.102,109, 6-9 Oct. 2013
5. Feng Li; Ozsu, M.T.; Gang Chen; Beng Chin Ooi, "R-Store: A scalable distributed system for supporting real-time analytics," *Data Engineering (ICDE), 2014 IEEE 30th International Conference on* , vol., no., pp.40,51, March 31 2014-April 4 2014
6. L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, “S4: Distributed stream computing platform,” in IEEE International Conference on Data Mining Workshops (ICDMW), 2010, pp. 170–177, IEEE, 2010.
7. M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik, “Scalable distributed stream processing,” in Proc. Conf. on Innovative Data Syst. Res, 2003.
8. M. Nathan, X. James, and J. Jason, “Storm: Distributed real-time computation system.” [Online]. Available: http://storm-project.net/, 2012.
9. Dena, D.; Bucicoiu, M.; Bardac, M., "A managed distributed processing pipeline with Storm and Mesos," *Networking in Education and Research, 2013 RoEduNet International Conference 12th Edition* , vol., no., pp.1,6, 26-28 Sept. 2013.
10. 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”, The VLDB Journal, vol. 12, no. 2, pp. 120-139, Aug. 2003.