CUBE DATA MODEL FOR MULTILEVEL STATISTICS COMPUTATION OF LIVE EXECUTION TRACES

Naser Ezzati-Jivan *Michel R. Dagenais*

n.ezzati@polymtl.ca michel.dagenais@polymtl.ca

Department of Computer and Software Engineering
Ecole Polytechnique de Montreal
Montreal, Canada

ABSTRACT

Execution trace logs are used to analyze system run-time behaviour and detect problems. Trace analysis tools usually read the input logs and gather either a detailed or brief summary of them to later process and inspect in the analysis steps. However, continuous and lengthy trace streams contained in the live tracing mode make it difficult to indefinitely record all events or even a detailed summary of the whole stream. This situation is further complicated when the system aims to compare different parts of the trace and provide a multilevel and multidimensional analysis.

This paper presents an architecture with corresponding data structures and algorithms to process stream events, generate an adequate summary -detailed enough for recent data and succinct enough for old data- and organize them to enable an efficient multilevel and multidimensional analysis, similar to OLAP analyses in the database applications. The proposed solution arranges data in a compact manner using interval forms and enables the range queries for any arbitrary time durations. Since this feature makes it possible to compare of different system parameters in different time areas it significantly influences the systems ability to provide a comprehensive trace analysis. Although the Linux operating system trace logs are used to evaluate the solution, we propose a generic architecture which can be used to summarize various types of stream data.

Index Terms— stream processing, multilevel analysis, OLAP analysis, trace abstraction, Linux kernel.

1. INTRODUCTION

Many applications such as network monitoring, web log analysis and stock exchange analysis tools provide different statistics over data streams [1, 2, 3]. In these applications, a system administrator or an automated program monitors the statistics of different system parameters (e.g., the usage of different system resources) to detect any possible problems, patterns or attacks. For instance, monitoring and counting the number of half-open connections in a short duration and comparing it to a predefined threshold (or to the number of completed connections) may help to detect a denial of service attack.

In a previous work [4], we presented a framework to store a history of (offline) trace summary in the trace reading phase to compute and provide the different statistics of system parameters in the analysis phase. However, since the trace stream size, despite the offline trace, is considered unlimited, it is not possible to store a complete history of the stream. Thus, heuristics are needed to select and store only the parts of the input data that are enough to provide accurate statistics. In other words, a trade-off between the size of the summary and the accuracy of the query responses is necessary. But in general, to guarantee the scalability of the solution the size of data structures should be small, somewhat independent of the length of the input trace stream or at most poly-logarithmic to that.

The query response time is also an important factor in the aforementioned stream data analysis applications. The system should provide a fast response time, facilitate the interactive use of the system and satisfy the real-time constraints of the streaming applications. The other factor is the processing time of the input stream. Since a new event may arrive at any arbitrary time, per event data processing rate should be efficient so that the analysis system can operate without congestion or having to drop input events.

Another challenge is providing a multidimensional and multilevel analysis. Analysts usually wish to perform the multidimensional analysis of the input trace stream on an expressive abstract level, including some multilevel exploration operations like drill down or roll up to get more or less detailed information [3]. Trace events are multidimensional in nature and usually represent interactions of different dimensions. For instance, a ”file read” trace event may contain information from the running process, the file that has been read, the current scheduled CPU for this operation and the return value (i.e., the number of bytes read by the operation).

In this paper, we contribute data structures and corresponding algorithms to construct stream cubes and provide OLAP [1] analyses over stream trace data. The solution incrementally constructs a compact and scalable data store from the input data, records it in the main memory (and possibly in the disk) and provides an efficient query mechanism for any flat or hierarchical queries over a system parameter or a group of them. Using this approach, users will be able to compute statistics of multiple system parameters, at different granularity levels, and for any arbitrary time ranges of the system execution.

Another contribution is that the proposed solution supports efficient range queries over the time dimension. Other approaches that support range queries usually work by storing a solid value of the data and counting or summing up the values for the queried range. However, this method could be a time-consuming task, especially when the selected range is relatively large. By storing the summary data as intervals, our solution provides an efficient query response time for range queries, regardless of the size and position of the given range.

The rest of the paper is organized as follows: first, after looking at the related work we present the architecture of the solution, the data structures and the techniques used. Second, we describe the different query types that the system supports. Then, we discuss the evaluation and experimental results of the proposed method. Finally, we conclude by outlining specific areas of investigation for future enhancements.

2. RELATED WORK

Stream analysis has many applications in network monitoring, web logs and click stream analysis, call records analysis, stock exchange and bank transaction analysis, medical records monitoring, weather monitoring, etc [1, 5]. Several research studies have been conducted in the literature on the stream data management [6, 5], OLAP analysis over stream data [3, 7, 8] and data mining [9, 10]. These studies present interesting ideas and results on stream data analysis to extract changes, trends and detect problems.

Several data structures have been proposed to store a history of stream data. Using a modified version of H-Tree, a stream cube [3] is proposed to perform a multidimensional and multilevel OLAP analysis over data streams. They use different time granularities for recent and decent information and a tilted time frame [11] to compress the data over the time dimension. They avoid recording information of all levels and only store the information along the critical paths. With this technique, the information that is not stored directly requires on-the-fly processing to be extracted. Even though our method uses a time frame similar to the presented tilted time frame, it uses different data structures and organizations to manage the stream cube. In our method, all items of the same time points can be extracted synchronously with a single query. Moreover, in our method the range query over the time dimension is supported directly and efficiently.

Patroumpas et al. [12] propose a stream management approach that uses different window sizes. In their technique, different windows are filled simultaneously. The problem with this solution is that they duplicate data in the different windows for the same time. In our approach, however, we avoid duplicating data at different windows while we support different windows and time granularities. Users can retrieve information for the last n milliseconds, seconds, minutes or even for any coarser granularities.

The fixed and moving sliding window methods are used in the literature to analyze the stream data [1, 13, 12, 2]. In these techniques, a recent window of items is kept, processed and used for extracting the desired statistics. The fixed sliding method refers to the case where a window (fixed or variable size) is kept or monitored for fixed durations in the past (e.g., each 1 second, starting from 2:00 PM yesterday), while the moving window refers to non-fixed start and end points that move with the time and is measured using the current time (e.g., every last 10 seconds) [1]. In our approach, we support both the variable-size fixed and moving sliding window queries over the stream cube. At each point, user can extract multidimensional statistics for the last n time units or can compute statistics for a fixed window in the past.

3. PROBLEM STATEMENT

In this section, we describe formal notations and definitions required to present the problem.

3.1. Preliminary Definitions

A dimension schema D is a tuple $\prec Name, L_D, \preceq \succ$ where: $Name$ represents a unique name for the schema, L_D denotes a set of levels, representing the multiple granularity levels of a dimension, and $\preceq$ represents a partial order between elements of L_D that is a graph or a hierarchy of dimension member items. Each level contains a set of members. A dimension instance d_i is defined as a set of members d_m from all levels. Figure 1 shows an example of four dimension schemas: Operation, Machine, Process and File.

In the example shown in Figure 1, the dimension instance Process is defined as $\{$(System $\preceq$ Process Group), (Process Group $\preceq$ Process Name), (Process Name $\preceq$ Process ID)$\}$. Process Group 1, Apache, Firefox, etc. are also member items of the Process dimension, shown in Figure 2.

A trace stream is defined as a sequence of timestamped events $\dots$, e_i, $\dots$,e_n, in which, e_n is considered the most recent event. Each event e = $(d_{m1}, ..., d_{mk}, r_1, ..., r_m)$ represents an interaction between a set of dimensions $(d_{m1}, ..., d_{mk}$, i.e., a set of system resources such as CPU, process ID, filename, disk block number that results in one or more return numbers, $r_1, ..., r_m$. For instance, a file open event (open,

[1]Online Analytical Processing

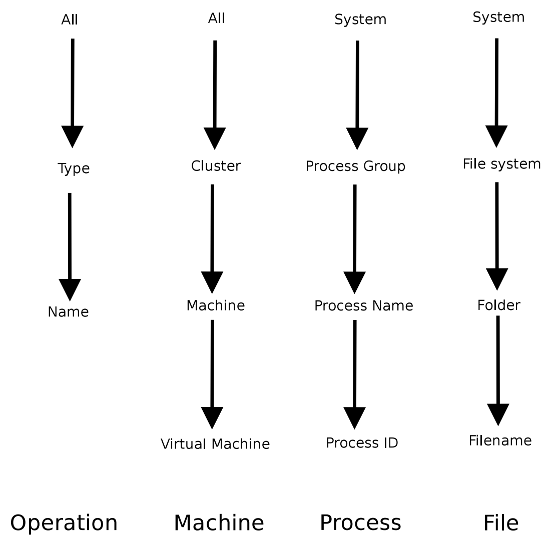

Fig. 1: Examples of dimension schemas.

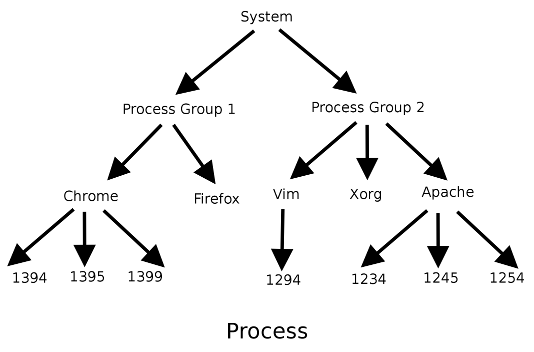

Fig. 2: An instance of the Process dimension schema.

CPU1, process2, file0, 3) represents a file open operation that is performed by a process process2 run by CPU CPU1 to open a file file0. The result, number 3, shows the output of the operation: the assigned file descriptor value.

One mandatory member of each multidimensional event is the timestamp field $t_i \in T$ that is used to order the events. T, the time dimension, is the set of Natural numbers: T= $\{t | t \in \mathbb{N}\}$. In this domain, a time interval $[t_1, t_2]$ is defined as $\{t \in T | t_1 \leq t \leq t_2\}$. We also assume that the timestamp values are distinct and any two events e_i, e_i have different values, $t_i \neq t_j$. Figure 5 represents a set of trace stream events gathered by the LTTng kernel tracer. LTTng [14] is a low-impact and lightweight open source Linux tracing tool, and provides detailed execution logs of operating system and user space applications.

We define the term trace stream cube as a collection of cuboids constructed over a trace data stream. Each cuboid represents a possible group-by of a measure over a set of dimensions. The most specific cuboid, the base cuboid, contains items from all dimensions. It can be used to gather the statistics for any dimension combinations,e.g., return the IO throughput for a particular process over a text file in a given virtual machine. The most generalized cuboid that is also called an Apex cuboid, contains the total value of a measure for all dimensions, e.g., whole CPU utilization of the system. Exploring downward from the apex cuboid to the base cuboid is called drilling down and the opposite operation, going upward from the base cuboids to the apex cuboid is called rolling up.

Example. Having three dimensions D1, D2, D3 and a measure M, the apex cuboid is (*, *, *, M), while the base cuboid is (D1, D2, D3, M). The combination of all possible cuboid forms a cube: (*, *, *, M), (D1, *, *, M),(*, D2, *, M),(*, *, D3, M),(D1, D2, *, M),(D1, *, D3, M),(*, D2, D3, M),(D1, D2, D3, M).

The term metrics is used to represent quantities to compute, monitor, compare or evaluate the usage or performance of the different system parameters at different levels of abstractions. For instance, CPU utilization and network throughput are examples of these metrics. We refer to these quantities using both the terms ”metrics” and ”measures” in the remainder of this text.

Having defined these terms, we seek to perform a multidimensional analysis: we will efficiently extract and compute the different system statistics from the input trace stream for not only the different single time points but also for any arbitrary time ranges at the different levels of granularity. For example, one might require the input/output (IO) throughput of the whole system, a specific virtual machine, process or a file, for a specific exact time, e.g., at 3:36’. or for the last 30 minutes.

3.2. Statistics to Monitor

Different types of statistics are supported in our approach:

1- Statistics such as sum, count and average are supported for any combination of the defined dimensions. Typical examples are the IO throughput of all files in a particular folder, the count of specific event types, or the average usage of a specific CPU. Using these queries, it is possible to provide frequency counting, or top-k elements, for any time range. One obvious application of frequency counting is to detect whether the statistics values exceed the predefined threshold values. Similarly, top-k queries can be used to identify the users, processes or applications that consume the majority of the system resources.

2- Range queries (for the time dimension) are supported for different time points and periods. The selected time range could be a time duration completely in the past, e.g., retrieve the desired statistics between 2 PM on February 3 to 3 AM on February 4, or a range between a time in the past and now $[t-\tau, t]$, e.g., retrieve the desired statistics for the last 30 minutes. In other words, moving sliding windows [1] are supported, in addition to the fixed sliding window where the user seeks statistics per fixed time units, e.g., each 5 minutes.

3- Different time scales are supported in this method. The selected time range could vary from milliseconds to days, weeks or even months. For instance, users may ask queries like ”return the desired statistics for the last n milliseconds, seconds, minutes or any coarser granularity”. However, for the earlier times we use a larger time granularity to extract the finer grain statistics. In other words, for the most recent time, any time range from the millisecond scale is supported, while for time periods in the past, the precision is decreased and coarser grain times are supported (i.e., hours or days instead of seconds and minutes). This consideration is normal in many applications [11], as users usually seek highly precise data for the more recent times and possibly coarser time ranges for the more distant times. In short, the proposed solution guarantees that different time scales (larger than the base time granularity) are supported and users can extract the desired statistics for different time scales and ranges.

4- Hierarchical operations like drill down, roll up, slice and dice are supported in this design, similar to the offline OLAP systems. Users may query the system to get a higher level aggregate value or may ask for more detailed statistics values. For example, having total IO throughput of a virtual machine, one may wish to access the detailed throughput of its processes separately, or having the network traffic of each IP address separately, one might see the aggregated traffic for a range of network addresses in the past days.

For all of the above problems, the compactness of the data structures and the construction and query performance of the method are considered the main requirements.

4. ARCHITECTURE

A high-level view of the architecture is shown in Figure 3. In this architecture, the trace reader reads and processes the input data, extracts the required summary and records in a data store named cube data model. Cube data model, in turn, contains different structures to organize and manage the data summary. Query engine is responsible to handle and respond queries received from the users. These modules will be explained in detail separately in the following sections.

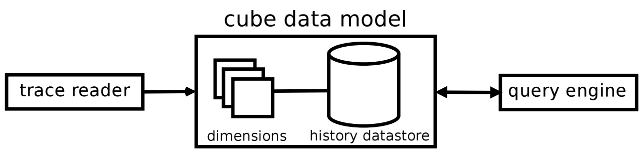

Fig. 3: A high-level view of the architecture.

4.1. Trace Reader

The first step in trace analysis is to extract the required data from the input trace events. Trace events usually describe the system in a very low-level form (Figure 5), and useful and synthetic information (e.g., the statistics data we are looking for) are usually hidden behind these low-level events. Some data analysis steps are required to extract the desired high-level information from the original trace data. To do so, for each statistic metric, a set of events is registered and monitored. For instance, socket-based events like socket connect, socket send, socket receive, etc. are registered and monitored to collect statistics about the network traffic. When one of the registered events arrives from the input stream, it is analyzed and the desired information is extracted and stored for future use.

Since the observed trace events are too low-level, some high-level statistics measures are not obviously recognizable and may require a more sophisticated analysis of the input stream. For instance, calculating the number of file downloads (as a high-level measure) is not obvious, as it is necessary to integrate firstly some specific low-level events [15] to generate the higher-level data to be able to compute their high-level statistics.

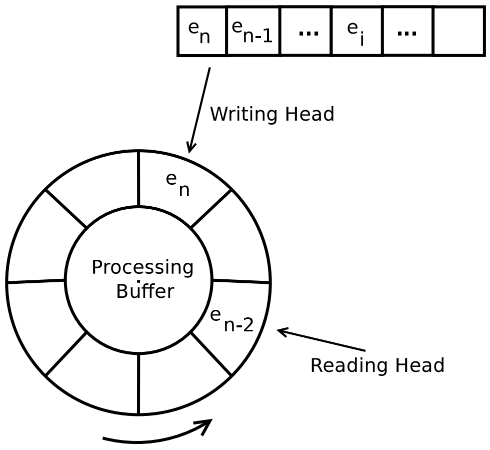

Fig. 4: Circular buffer to read and process the stream events.

One issue in the stream events data extraction is that the distance of consecutive events could be less than the time required to process them. Indeed, this may happen when the events are too low-level (as shown in Figure 5) and must be abstracted out to higher levels before extracting the statistics information. In our implementation, we use a separate thread, rather than the thread used to read the stream, to process the incoming events. To do so, a circular buffer is initialized and preliminary information gathered from the events are copied into that buffer (as shown in Figure 4). The processing thread then operates on the buffer, gathers the statistics information, and stores it in the history data store.

However, for the time periods in which too many events are received, e.g., when the system is too busy and the tracer module generates many events, the buffer may overflow due to the slow computation. In this case, two following solutions may solve the problem. First, by increasing the gaps between the processing and database updating steps, the statistics can

```
kernel.syscall_entry: 509.147214537 (kernel_1), 1697, 1697, lttctl, 1, 0x0, syscall { 0x7f0b0eb96cbd, 0 [syscall 0] }
kernel.sched_schedule: 509.147214752 (kernel_7), 0, 0, swapper, 0, 0x0, syscall { 48, 0, 1 }
fs.read: 509.147220219 (fs_1), 1697, 1697, lttctl, 1, 0x0, syscall { 8176, 21 }
kernel.syscall_exit: 509.147220604 (kernel_1), 1697, 1697, lttctl, 1, 0x0, user_mode { 241 }
fs.write: 509.147227093 (fs_5), 568, 538, rs:main Q:Reg, 1, 0x0, syscall { 66, 4 }
kernel.syscall_exit: 509.147227571 (kernel_5), 568, 538, rs:main Q:Reg, 1, 0x0, user_mode { 66 }
kernel.syscall_entry: 509.147234027 (kernel_1), 1697, 1697, lttctl, 1, 0x0, syscall { 0x7f0b0eb96cbd, 0 [syscall 0] }
kernel.syscall_entry: 509.147235150 (kernel_5), 568, 538, rs:main Q:Reg, 1, 0x0, syscall { 0x7fcb30146c5d, 1 [syscall 1] }
net.socket_recvmsg: 509.147236434 (net_1), 1697, 1697, lttctl, 1, 0x0, syscall { 0xffff880188c90580, 0xffff880199fe1d70, 4096, 64,
fs.read: 509.147237217 (fs_1), 1697, 1697, lttctl, 1, 0x0, syscall { 4096, 3 }
kernel.syscall_exit: 509.147237564 (kernel_1), 1697, 1697, lttctl,1, 0x0, user_mode { -11 }
kernel.syscall_entry: 513.772101451 (kernel_4), 2334, 2334, /opt/google/chrome/chrome-sandbox, 2037, 0x0, syscall { 0x7fccc73b0007, 2 [sys_open+0x0/0x30] }
fs.open: 513.772106530 (fs_4), 2334, 2334, /opt/google/chrome/chrome-sandbox, 2037, 0x0, syscall { 3, ”/etc/ld.so.cache” }
kernel.syscall_exit: 513.772107125 (kernel_4), 2334, 2334, /opt/google/chrome/chrome-sandbox, 2037, 0x0, user_mode { 3 }
```

Fig. 5: LTTng trace events for common files accesses.

be computed and the database updated less frequently (e.g., update the database every 10 seconds instead of every 1 second). The other solution is to disregard the extra events and not process them until the system returns to its normal state. In the experimental results section, we discuss an evaluation of the distance between the incoming events and the time required to process them. However, it is outside the scope of this research to discuss all possible cases and the detailed solutions for each. We focus here on the data structures and the way we manage the processed data in a long, continuous stream data.

4.2. Cube Data Model

The proposed method works by extracting the preliminary and punctual statistics from trace events and storing them in a disk-based data structure. It incrementally builds an efficient history of data, so as to be readily retrieved when needed. The overall view of the cube data model is shown in Figure 6.

As shown in Figure 6, the cube data model contains two main structures: dimension tree and history data store. Dimension tree models the different system dimensions and parameters and acts as a set of key references for the history data store. History data store is the real storage of the data in which the summary of input trace is stored. This history data store contains different cubes for different time frames (Figure 6). The cubes in turn are implemented by tree structures. In other words, each cube corresponds to a time frame (range) and contains an interval tree that stores the information of that time frame. Each interval has a key from the dimension tree, a value that is gathered from trace, a start time and an end time. More details will be provided for each structure in the following subsections.

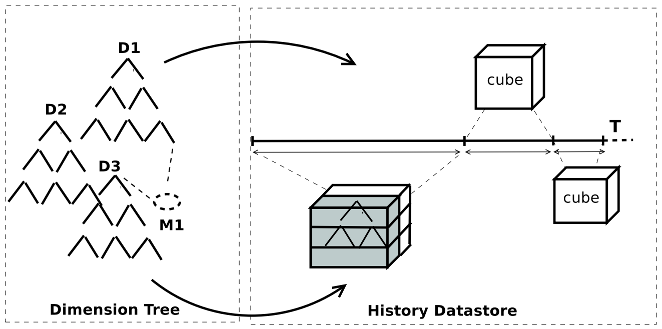

Fig. 6: Two internal structures of the cube data model.

h contains different parts that are explained in this section.

Dimension Modeling

One may wish to gather statistics about several types of system resources (e.g. memories, processes, files, devices, etc). In our model, each resource (i.e., dimension) is structured as a hierarchy and the metrics of interest are defined between these hierarchies. Figure 7 shows two dimensions -Process and File- and the IO usage metrics that are defined between these two resources.

In this organization, that is called dimension (metrics) tree, it is possible to define the metrics between any set of resources at any granularity. For example, as shown in Figure 7, one may define the metrics node between the process and file dimensions to compute the IO throughput of a particular file/-folder performed by a specific process, e.g., to place the desired metrics between the Chrome and /Home nodes to compute the number of bytes read or written for all files in the

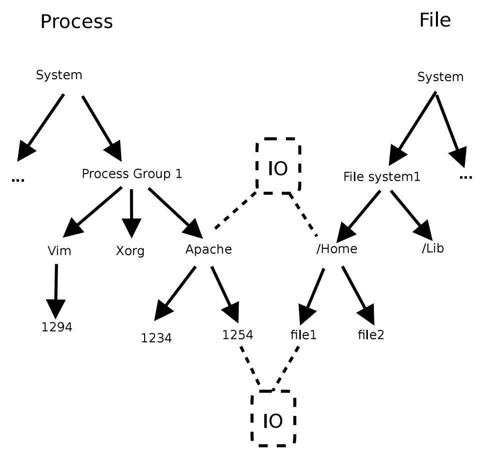

Fig. 7: Dimension hierarchies and metrics.

/Home directory by the "Chrome" process.

As explained, an important feature of this tree is its ability to define the metrics at any granularity levels. For instance, some users may only be interested in a specific high level granularity (e.g. the IO throughput of a whole virtual machine, and not their containing processes), thus the system can avoid storing finer or coarser values, thus saving a significant amount of storage. We will further study the gains of this design in the Experimental Results.

In the aforementioned dimension (metrics) tree, the metric nodes (indicated by dotted rectangles) comprise a link to the corresponding value records in the history data store. They also keep track of pointers to the first and last occurrences of the metrics value in the stream, representing the operational scope of the metrics. These pointers increase the speed of queries for the points that are beyond the operational scope.

History Data Store

For any predefined metric, we keep a history of its values during its whole trace lifetime. To do so, when a registered event arrives and the corresponding statistic value is changed, we create and store a summary of that change in the history data store. We model this change as an interval record and store an interval instead of two single records. For each interval, the bounding points (start time and end time), a key and a value are all stored. The key is a combination of a set of dimensions and a measure referring to the metric nodes (dotted rectangles) in the dimension (metrics) tree. The value represents the statistics value for the chosen metric in this time interval. For example, suppose a "file read" event is registered in advance for the IO throughput metric. Then when a file read event (e.g., process p1 read 400 bytes of file f1 in time t2.) is arrive, and changes the value of a corresponding metric (i.e., the IO throughput), we create and store an interval record for this change: {IO throughput of process p1 and file f1, 400 bytes, t1, t2}. This record actually shows that the IO throughput of process p1 that is read/written from/on file f1 between time t1 and time t2 is 400 bytes. Here the time t2 is the current event timestamp and t1 is a time that a previous registered event (read event or write event or etc.) is seen and processed.

Using the above technique, we store statistic summary as interval values instead of single values which enables the range queries. Hence, to store the interval values, any interval container, such as an R-tree [16], the SLOG2 file format [17] or a State History Tree [18] can be used.

Significant space is required to store the summary of a stream in this way, which means after a short while, both the main memory, and finally the disk will be filled. To solve the problem, we use some heuristics to solve this problem. One heuristic uses a proper granularity degree (GD) [4], and stores a cumulative interval instead of each single interval. To do so, for any k (i.e., the GD value) consecutive intervals, we store a single record representing the statistics value between the time ranges of those intervals. In the example shown in Figure 8, for each 7 operations, only one data interval is created and stored (instead of 7 separate interval records). A small GD value increases the history storage space while a larger GD value reduces the precision of the statistics. Thus, a careful consideration of the proper GD value is important and can be achieved through balancing the importance of the metrics and the available storage space.

Another heuristic that can be used to alleviate this problem is using the so-called Tilted Time Frame technique [11] to compress the time dimension. The idea is to use a coarser granularity degree (GD) for the older history, yet a finer value for the most recent history.

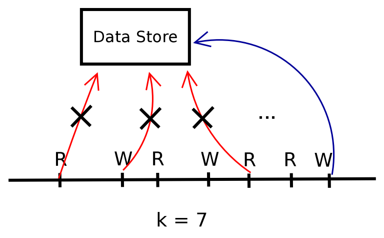

Fig. 8: Efficient updating the history data store.

Time Frame

In the trace analysis applications, the most recent history is more interesting than older history [3]. Thus, for the long term history, we can use coarser granularity degrees. For example, as shown in Figure 9, for the last 5 minutes of the trace a minimal granularity degree (e.g., 1 second) is used, and for the last 24 hours range, a larger granularity degree (e.g., 5 minutes) is used. With this method, after reading any 5 minutes of the input stream, aggregate statistics of the last 5 minutes are calculated and passed through the cube in the other level. After 60 minutes, another aggregation is calculated and stored in another cube level.

A high-level view of the time frame is shown in Figure (10). In this method, for the duration of each time unit, a separate stream cube is constructed and materialized. The cube is actually implemented using the dimension (metrics) tree and the history data store. In other words, to manage and store the cuboids we do not use any external database applications, as is common in the OLAP applications, but instead we implement each cube using two tree-based structures, the one is used to mange the dimensions (dimension tree) and the interval tree that contains the real data for the current time frame (history data store) (Figures 6, 14).

Each cube is used to answer the queries inside the corresponding time duration. After passing this time yet before dropping the cube, an aggregate of this cube is computed and stored in the cube at one step coarser.

Using this method, one can store a history of the statistics values for a long time and utilize a small amount of storage space. We will now explore the storage space this method requires to keep track of the intervals.

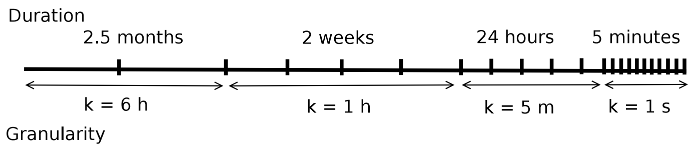

Fig. 9: Different granularity degrees for different time durations.

Let us assume there exist n metric nodes in the dimension (metrics) tree. We also assume that all metrics have changed each second and a new interval must be created for each metric and each second. For the recent 5 minutes, $5 \times 60 \times n = 300n$ record spaces are required to store all the interval values for these n metrics. In the same way, the other 24 hours require: $(24 * 60) \div 5 \times n \approx 300n$. Similarly, for two weeks and 2.5 months, $(14 * 24) \div (1 \times n) \approx 300n$ and $(75 * 24) \div (6 \times n) \approx 300n$ respectively are required. Thus, for a 2.5-month period, $300n + 300n + 300n = 900n$ record spaces are required which is 1/10000 of the case that uses a uniform time frame (i.e., $900n \div (75 * 24 * 60 * 60 * n \approx 6480000n) \approx 1/10000$).

Cubes Construction

Using the tilted time frame, different trees (cubes) correspond to different time units and one must pass a set of aggregated values to a granular tree (cube) when the time unit is changed. In other words, after crossing the first tf_1 time units (e.g., the first 5 minutes), this method aggregates the statistics and inserts these aggregated values into the tree of another time unit, such as tf_2 (Figure 11). This also occurs after crossing any

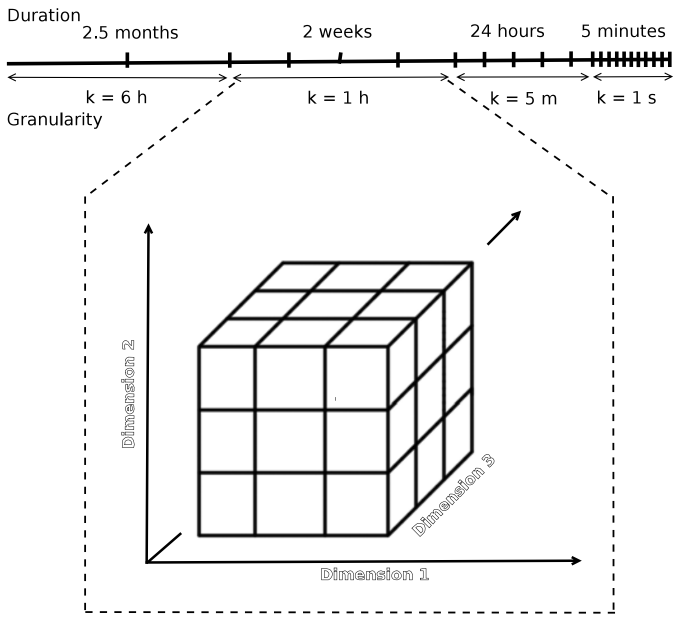

Fig. 10: A separate sub-cube for each time unit.

other tf_1 time units (e.g., each 5 minutes). In the same way, after crossing tf_2 time units (e.g., 24 hours), one must perform another aggregation of the values of tf_2 time units and insert into the coarser time unit tree. This process is repeated after each time unit changes.

As shown in Figure 11, a separate cube corresponds to each time unit, one for the area with minimum time scale tf_0, one for the area with the minimum time scale tf_1 and so on. After passing the first tf_1 time, an aggregation process is called and the aggregated records are inserted into the other level tree. This process is repeated when it passes the tf_2 time. Let us now explore the costs of these aggregate updates.

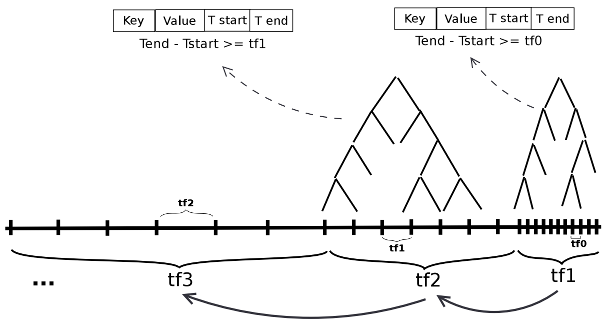

Fig. 11: Moving the aggregated values from one tree to another.

Again, let us assume that there are n metrics in the metrics tree and the statistics values for all metrics are changing at each time unit. After crossing the tf_1 time, we must aggregate all values of the tf_1 time period and insert them into the cube corresponding to the tf_2 period (Figure 11). Since there are n metrics in total, each metric requires one record; thus, n records all metrics together. Each record shows the statistics value of the corresponding metric in the completed time range. Thus, at each step any time unit change requires n aggregate updates into the higher-level cube. Therefore, using the parameters shown in Figure 11, formula 1 can be used to calculate the number of updates from one tree to another.

$$\begin{aligned}&\psi_{all} = \\ &(tf_2 \div tf_1) \times n + \\ &(tf_3 \div tf_2) \times n + \\ &\cdots + \\ &(tf_n \div tf_{n-1}) \times n.\end{aligned} \tag{1}$$

The above formula calculates the required number of aggregate updates from one tree to a coarser level tree. The resulting value is a very small portion of all whole tree insertions:

Let us calculate the total number of insertion operations in all trees and compare that to the number of aggregate update operations. Suppose that for each tf_0 duration (the smallest time unit in the proposed time frame for which values can be gathered directly from the input trace events), the values of all metrics are changed. Thus, we will have n updates for each tf_0 duration. For the larger level, tf_1 duration, the number of insertion and update operations in the history will be:

$$\Phi_{tf_1} = (tf_1 \div tf_0) \times n. \tag{2}$$

Similarly, the number of insertions for each tf_2 duration:

$$\begin{aligned}&Val_{AB} = Val_B - Val_A = x1 + x2 + x3 \\ &\Phi_{tf_2} = [(tf_2 \div tf_1) \times ((tf_1 \div tf_0) \times n)] \\ &= (tf_2 \div tf_0) \times n.\end{aligned} \tag{3}$$

And:

$$\begin{aligned}&\Phi_{tf_m} = [(tf_m \div tf_{m-1}) \times ((tf_{m-1} \div tf_0) \times n)] \\ &= (tf_m \div tf_0) \times n.\end{aligned} \tag{4}$$

Totally:

$$\begin{aligned}&\Phi_{all} = \Phi_{tf_1} + \Phi_{tf_2} + ... + \Phi_{tf_m} \\ &= (tf_1 \div tf_0) \times n + (tf_2 \div tf_0) \times n + \dots \\ &+ (tf_m \div tf_0) \times n \\ &= \frac{\sum_{i=1}^{i=m} tf_i \times n}{tf_0}.\end{aligned} \tag{5}$$

Therefore, the fraction of the aggregate updates with respect to all insertion and update operations equals:

$$Portion = \frac{\psi_{all}}{\Phi_{all} + \psi_{all}}. \tag{6}$$

For the example shown in Figure 9, this proportion is 0.00006 = 0.006 %. In other words, the aggregate update cost is a very small proportion of all operations and is not an issue. The challenging issue is the time required for the tree construction, which will be investigated in the Experimental Results section.

We use another heuristic to reduce the number of tree insertions: when a metrics value is unchanged in two or more consecutive time units, no update is required in the tree. To do so, we store the current value of the metric in a temporary structure and wait for a change. After the first change, a node is inserted in the tree representing the value of the metrics for all unchanged time durations. This technique reduces the number of insertions in the history data store.

Sliding Window

One use-case of this research is to support the sliding window queries, both the fixed and moving sliding windows. In this subsection we propose a technique to support the sliding window queries.

As explained, at regularly defined time points, the algorithm aggregates the values of the current tree and inserts them into a coarse tree, belongs to a larger time frame.The fixed sliding window is obviously supported, because the values of any previous time ranges are available in the data structures. But to support the moving sliding window one inefficient way is to update the history for all new trace events. In other words the algorithm should update all trees at any granularity levels for each new event, after passing the first tf_0 time (e.g., a second). However, it would be too costly to update the tree structures, remove the old entries and insert a new one, for each new time unit tf_0 (e.g., a second).

The way we support the moving sliding window is by delaying the aggregate moving from one tree to another tree. In other words, we do not immediately move the aggregate value to another tree, but wait for another tf_1 time and then aggregate the tree and move to the coarser tree. Figure 12 depicts this technique (with respect to Figure 11).

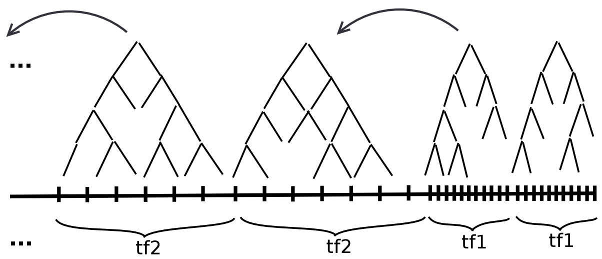

Fig. 12: Supporting the moving sliding window by delaying the aggregate updates.

To illustrate this point, we use the values shown in Figure 9. Instead of aggregating the tree for the last 5 minutes and discarding the detailed tree, we keep these details in the memory and continue for another 5 minutes. At the end of the second 5 minutes, we simply aggregate the first tree, (the tree from the first 5 minutes), move it to a coarser level tree, and discard that tree from the memory. At the query time, when users ask for the last k, say 7, minutes, we can easily use these two trees (the trees of the first and second 5 minutes) to answer the query. As shown in Figure 12, the duplication and delay in aggregate propagation are used in all time frames, enabling the extraction of the desired statistics values for any arbitrary k time units with a varying time precision.

5. QUERY

The proposed method, as explained in the previous sections, reads the input trace events and extracts the statistics from the data. Then, the statistics data is stored in a tree-based history data store. In this history, the more details are stored for recent data and less for older data. Using this configuration, it is possible to reduce the details from the older history, while still satisfying the queries for recent history with a higher precision. In general, different types of queries are supported. We first look at the range queries and subsequently address the other types of queries.

5.1. Range Queries

The records in the history data store, collected from the trace events, represent the cumulative statistics values for the specified metrics and the corresponding time range between the start and end points. Thus, to retrieve the statistics value at any point, one can simply find and explore the corresponding tree and find a node that contains the required point. The extracted value represents a cumulative statistic between that query point and a base time point. In the same way, to extract the statistics values for a time range, one may perform two stabbing queries and subtract the results to yield the desired statistics value. Figure 13 shows an example of a range query.

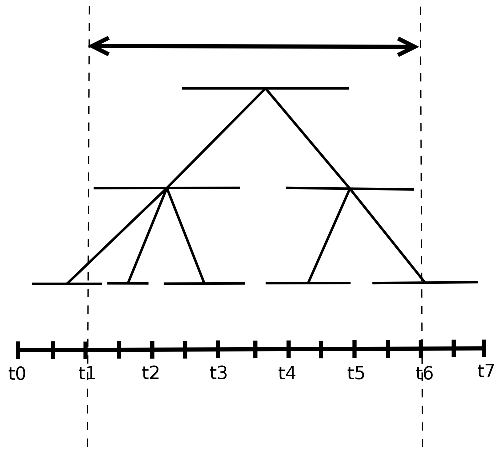

Fig. 13: Performing range query in the stream history.

As shown in Figure 13, to extract the statistics values in any time range, say $[t1, t6]$, two stabbing queries are required, one for $t1$ and another for $t6$. These two stabbing queries return two cumulative values with respect to a fixed start point. Then, the subtraction of the two values will provide the required statistics value (the increment within the interval).

Example: Suppose we have three records in the history: ([0,2), metrics1, 0), ([2,6), metrics1, 20), ([6,10), metrics1, 30). Each record contains a time interval, a key and a value. For instance, the first record shows that the value for metrics1 between times 0, 2 is zero. These records show that we have value changes in times 2, 6 and 10, since we create a new interval record only when a value changes, (here the GD value is 1). Using this history, the metrics1 statistics value between any time range in [0,10), say [3,9], will be the subtraction of the statistics value at 9 and 3 = 30 - 20 = 10, because time point 3 crosses the second interval record and time point 9 crosses the third record.

The stabbing query -finding the intervals that contain a given query point- is shown in Algorithm 1. Since our proposed solution does not force using the interval tree container, Algorithm 1 shows a general stabbing query for any typical interval trees. The maximum number of items in the result list L will be n, the number of metrics. Sometimes it is possible to have less than n intervals in the result list L. This means that some metrics do not have values for the given point, meaning that the corresponding resources were not active at that point, e.g., looking for the IO throughput of a process in a time prior to its starting time.

After performing a stabbing query, the result set L will contain statistic values of all metrics. Thus, one must filter out the result list L to find the value of the desired metrics.

Algorithm 1 Stabbing query.

```
Require: an interval tree v and a query point t.
1: if root node r contains the point t then
2:     add r to the result list L.
3: end if
4: if there is any children for node r then
5:     for any children of v like c(v) do
6:         call the algorithm for c(v) , t.
7:     end for
8: end if
9: return list L as result;
```

As explained earlier, we have different trees for different time points: more details for recent times and less for older times. The proposed stabbing and range queries work for all cases, for a time range within a single tree or several trees. For all cases, we can find the statistic values for the desired time range, by using two stabbing queries for the boundary points of each interval.

The last point in this section relates to calculating the aggregate values of a tree and moving that to a coarser tree. To calculate the aggregated values of all metrics for any time range, one must perform two stabbing queries, one at the start and one at the end point of that time range. Each stabbing query returns the values for all n metrics together, so it is not necessary to repeat this algorithm for each metric separately. Therefore, calculating the aggregate values of a tree is not costly. It simply requires two stabbing queries for the start and end points of the tree.

Top-K Queries

Sometimes it is important to detect when the values exceed a predefined threshold, or find virtual machines or processes which integrate more system resources than others. To support these query types, the system should be able to answer top-k queries, for any arbitrary k. To do so, we first use the mentioned range queries to extract the statistics values of all metrics, and then use an optimal sorting algorithm to find the top-k values from a maximum of n values.

The cost of this algorithm is $O(logm_1 + logm_2 + n \times logn)$, where m is the number of nodes in the history tree and n is the number of metrics. In this equation, $logm_1$ is the time required to perform the stabbing query for the history tree at the start point of the given query interval, $logm_2$ is used to extract the statistics value at the end point of the query interval, and $n \times logn$ is used to sort the n items, the output of the stabbing queries. The cost is obviously dependent on the count of metrics, n, and the depth of the tree, as one or the other may be more dominant depending on the situation.

5.2. Sliding Window Queries

This solution supports both the fixed and moving sliding window queries. As outlined earlier, the techniques used enable us to extract the statistics values for the last k time units for the fixed or moving values of k. An example of a fixed sliding window is reporting the statistics values after each k time units. In this case, after finishing the predefined k time units, say 1 second or 1 minute, the algorithm aggregates and returns the values for the desired time range. For instance, one may wish to retrieve a minute by minute report of the CPU usage. To do so, after finishing each minute, the program aggregates and reports the CPU usage for the preceding minute by summing up the values of small-scale chunks (e.g., by summing up the CPU usages of all 60 seconds of that whole minute).

Sometimes users wish to obtain values for a moving or sliding value of k, by taking into account the precision of the time unit. Suppose that we are in the 6th minute of the trace and the user asks for the CPU usage for the last 3 minutes. If we had aggregated the tree at the 5th minute, we would not be able to answer this query, since we need 2 more minutes from the history. However, we would not have the requested values with the desired precision if these values were already aggregated for that 5-minute interval. To solve this problem, the algorithm delays moving the aggregated values to a coarser unit tree for another time unit (e.g., another 5 minutes). For example, using the tree shown in 12, it is possible to provide values for any previous time range (less than the current time unit) e.g., last 3 minutes, last 58 minutes, last 11 hours, etc.

5.3. Multilevel Queries

Since in many applications, users may wish to perform some operations like group by, drill down or roll up, we investigate this type of queries here.

Due to the required storage space and processing time, it is not possible to generate and store all possible cuboids along the stream. To support multilevel queries, we propose two general solutions: minimal and partial cube materialization.

The first solution, minimal materialization, defines all metrics in the finer level - for the leaf nodes of the dimensions- and only stores the base cuboids (the history values for the leaf nodes of the metrics tree). Any other high-level metrics (none-base cuboids including apex cuboid) are computed on the fly using these low-level metrics. For instance, in the above example, it is possible to compute the IO throughput of virtual machines by performing aggregate functions (i.e., sum) over the low level history values, i.e, by summing up the IO throughput of all processes belonging to each virtual machine. Figure 14-1 shows a view of this solution.

Partial materialization [3], the second approach, is used when the high-level nodes, for which the analytical data will be queried, are predictable. Therefore, the solution creates metrics for these high-level nodes and keeps track of history values for them as well. For instance, in Figure 7, suppose the system is notified in advance that users wish to retrieve the IO throughput for all virtual machines together in addition to each process separately. In this case, a metric node representing the desired granular level is created in the metrics tree and a history is kept within the history data store. Figure 14-2 depicts a view of this solution.

Depending on the number of high-level nodes, the partial materialization solution may require more storage than the minimal materialization solution to store the data for the coarser or finer granularity scales. However, the later may require more processing time to aggregate and compute the desired high-level statistics on the fly using the low-level information.

A tradeoff between the processing/response time, the storage space and the user and application requirements is generally required to determine which strategy should be used. In some applications, a small number of high-level nodes may be critical to users and therefore data should be kept to directly and quickly retrieve answers. In this case, a partial cube materialization method is used to only materialize the important high-level nodes in addition to all leaf-level nodes.

To extract the high-level statistics (rolling up) for a given time point, when the requested value is not directly in the history, one must perform a stabbing query at the given point,

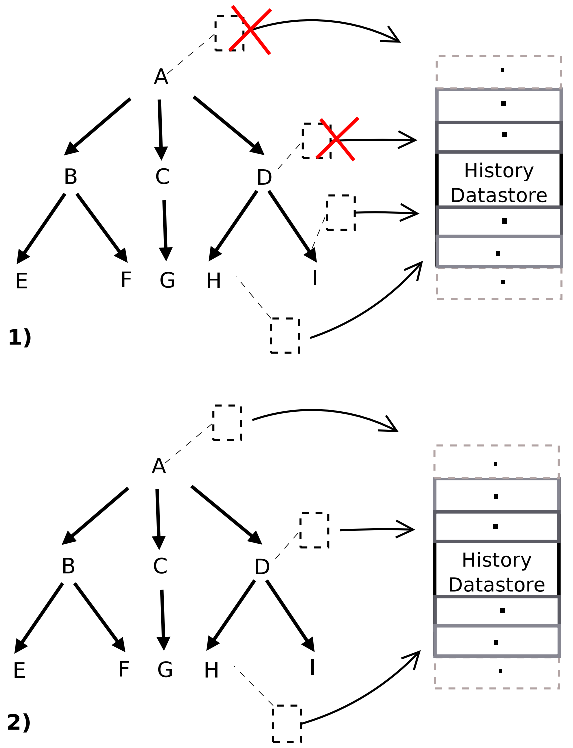

Fig. 14: Hierarchical queries. 1) Minimal cube materialization, using aggregate functions to compute hierarchical values, 2) Partial cube materialization, storing data at the multiple levels.

extract the values of all low-level nodes of the queried dimensions and finally aggregate the results (sum up, count or so on.). Since a single stabbing query will return all values for the given point, the rolling up query requires the same time as low-level queries, except for the extra time required to perform the aggregation over extracted values. For instance, to compute the IO throughput of a folder, one must aggregate the IO throughput of the files inside that particular folder, which can be obtained directly with a single stabbing query.

6. EXPERIMENTAL RESULTS

The experiments were performed on a Core i7 2.80 GHz system with 6GB of main memory, running Linux kernel version 2.6.38.6 instrumented with the LTTng tracer. The algorithms were programmed in Java using the Eclipse plug-in for Java and will eventually be contributed to the free software TMF (Tracing and Monitoring Framework)[2]. The tests were performed with real trace logs gathered from the LTTng kernel tracer. Since the original logs are too low-level, techniques are adopted from [15, 4] to abstract out the raw data to higher level and extract the desired statistics data. We also use the locally developed State History Tree [18] as the interval container for the interval values. With respect to the other approaches in the literature, this format works better for cases in which the input data arrives sporadically (in an unpredictable manner) and cases in which the tree is constructed incrementally. To generate the trace logs, system activity is generated using recursive operations like grep -r, wget -r -l, ls -R, etc.

The experiments will be discussed in three sections: processing time, memory usage and query response time.

6.1. Processing Time

In the first experiment, we aim to investigate the efficiency of the proposed trace analysis module, and whether a trace stream can be processed in real time (events are analyzed in less time than their rate of occurrence).

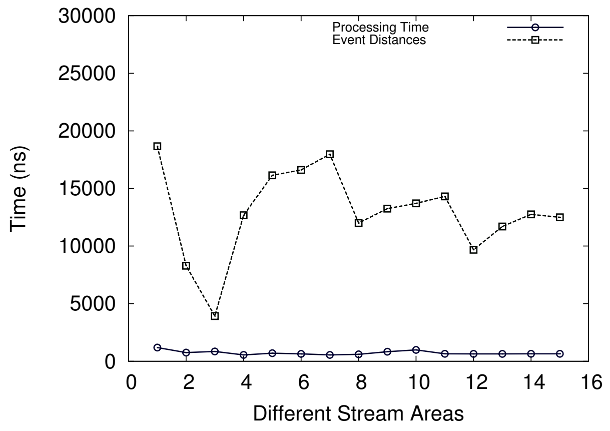

Fig. 15: Delay between trace events and processing time for one event (average over a batch of 10000 events).

Figure 15 shows the average delay between trace events in different areas of a trace. Each delay is calculated as the average time for 10000 subsequent events. Similarly, the processing time to analyze the trace is computed as the average over 10000 events. The analysis time per event does not vary much. The average delay between events varies significantly depending on how busy the traced system is. In our tests, the processing time remains much lower so that the average delay and the analysis is thus efficient enough to accept a streaming trace in real time. In extreme cases, where events are much closer to each another, buffering the events and performing a delayed processing, or even dropping some events, will be required. We may investigate these techniques in more detail in future work.

Figure 16 shows the different times required for reading and processing the trace stream. The first case only reads the trace without processing any data. The other curve shows both the trace reading and simple processing of the events: reading the trace stream, extracting the statistics values and aggregating each base time unit (e.g., 1 second). However, it does not include the time needed to store this data in the

[2]http://lttng.org/eclipse

data structure. Other cases show the time required to store the processed data to the different cubes in the history data store. In each case, the time frame is set so that the requested level cube is constructed. For example, in the case with only the first time unit, one cube is materialized, while in the case with two time units, the first and second levels are materialized and so on. Figure 16 shows that the first cube level requires the longest processing time. This is not surprising since the first cube level is updated for each base time unit (e.g., 1 second), while the others are called at granular time units (e.g., 5 minutes) and require less time.

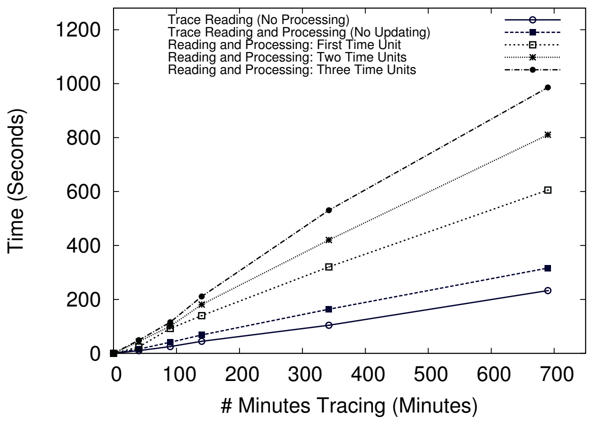

Fig. 16: Processing time for different stream processing steps.

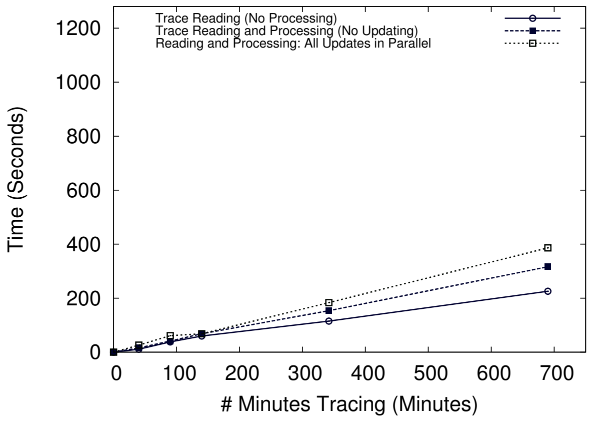

Fig. 17: Processing time for parallel cube updating

Figure 17 shows the same processing steps while performing a parallel analysis, in which a separate thread is assigned to each single cube update. The parallel cube update is possible since different cubes correspond to different areas and the data gathered from unique trace portions can be written to the different cubes simultaneously. In this method, after performing a preliminary analysis over the trace and extracting the statistics data, a separate thread is assigned to each cube and, as a result, updates are done separately yet parallel to each other.

It is important to note that in all the above experiments, 1000 measures are used. As explained earlier, each metric is considered as a measure between two or more dimensions. For instance, the metric CPU usage is defined between the virtual machine, process and CPU dimensions and could also be shown as (virtual machine, process id, CPU number, CPU usage).

6.2. Memory Usage

One of the important aspects of any useful stream processing method is the ability to use as little memory as possible. In this section, we show the memory usage for our proposed method. The history data store, which records the temporary and intermediate values, stores the data on disk rather than in memory, enabling it to store a longer period of abstract streaming data.

As explained earlier, there are three cube materialization methods: full, partial and minimal. In all experiments below, the partial materialization stores 10% of higher-level cuboids in addition to the lowest level cuboids.

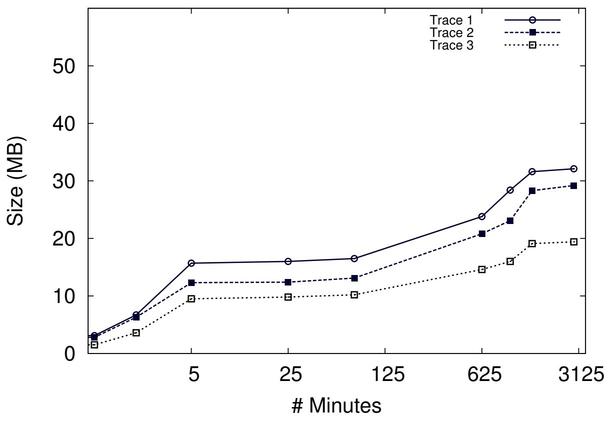

Fig. 18: Memory usage for different trace areas.

Figure 18 shows the memory usage for different areas of the trace stream. Different trace areas may have more or less events depending on how busy the underlying system is. In Figure 18, the curve shown with Trace 1 belongs to a busy area of the trace (where the number of events per second is higher than in other places). The data used for drawing the curve shown with Trace 3 belong to a less active area of the trace, with fewer events. In all three cases, the number of metrics used is again 1000. Additionally, the time units are from Figure 9. In other words, the memory is used to store three levels of cubes: the first level for the last 5 minutes, the second for the last 24 hours and the third for the last 12 days. However, we have used a one-day trace duration to test all three cube levels. The results show that the configuration used is desirable and readily usable. The maximum required memory is approximately 35 MB, which can easily reside in the main memory. However, as will be discussed shortly, the required memory may increase depending on the number of metrics or time units. This low memory usage is achieved

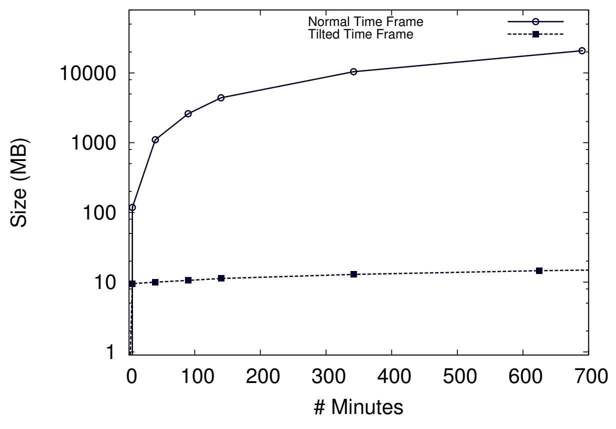

Fig. 19: Memory usages comparison for tilted and non-tilted time units.

at the cost of removing the more detailed history from the most distant time periods. It is obviously not feasible to store all information at the most detailed level for the whole trace duration, unlike for relatively small traces in offline tracing mode [4]. Figure 19 compares a comparison of the memory usage of two methods: storing all history for the whole tracing duration, and removing the old history from the data store (please note that a logarithmic scale is used for the y axis).

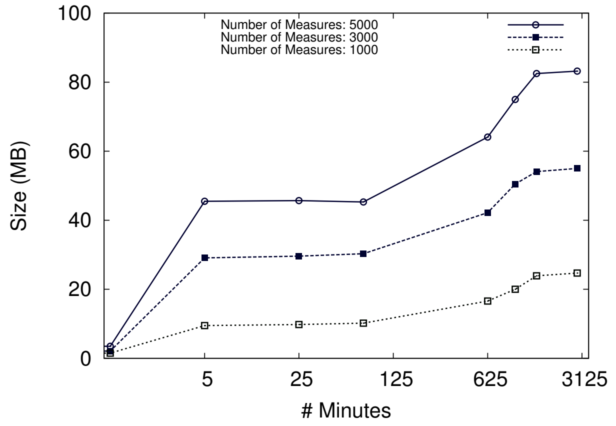

Fig. 20: Memory usages for different number of measures.

Another experiment was performed to see the effect of the number of metrics. It is possible to define different measures in different levels between the existing dimensions. For example, IO throughput can be defined between a file or folder in one hand,and a process or a group of processes in other hand. It is also possible to add a virtual machine to this measure, making it a threefold measure. Figure 20 shows the memory usage for different numbers of measures. In fact, the memory usage depends on both the type and number of measures. Indeed, the count and frequency of the events required to compute the values of a measure is the key factor to compute the memory required to store statistic values of that measure. The number of accesses to a specific web site requires events of type http that connect to the specified web site and can rarely be observed in the events. However, the IO throughput measure is based on the type of events (i.e., read or write events), which occur very often in any system execution, and thus require more storage space. The same metric, IO throughput, defined between three dimensions, (virtual machine, process and file), is used to gather the results in Figure 20. To increase the number of measures, for instance from 1000 to 3000, we have added new tuples (virtual machines, files and processes) to the existing tuple list. The results show that increasing the number of measures has a direct (but not linear) effect on the memory usage.

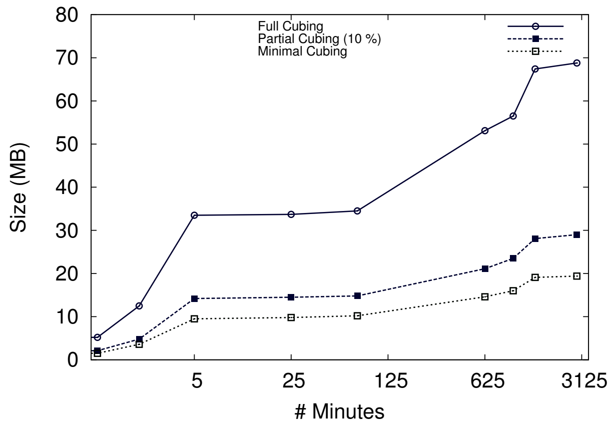

Fig. 21: Memory usages for different cube materialization strategies.

The final experiment was conducted to investigate the different materialization methods. To do so, three modes are defined for 1000 measures: Minimal cubing, in which all measures are defined in the leaf nodes of the metrics tree (dimension tree); Full cubing, in which measures are defined in all levels and the history data is stored for all levels; and Partial cubing, which is something in-between, defining all but 10 % in the leaf nodes. These 10% extra measures which are defined in the non-leaf nodes, can be considered as measures frequently requested by users, thus being less desirable to compute on-the-fly. Figure 21 shows the difference in memory usage for these three methods. Full cubing method demands more memory (three times or more to store the history in each and every level), while the partial and minimal cubing methods act very similar.

The above memory usage experimental results demonstrate that the size of the data store is relatively stable and independent of the stream data size. Indeed, the design is such that it stores only minimal data for the distant history, and the memory usage therefore increases very slowly (logarithmic) with the size of the stream. This is a very important feature of the proposed data store that makes the solution scalable and usable for any size of input stream data.

6.3. Query Response Time

A proper response time is an important factor for most stream processing applications. Indeed, these tools may be used interactively to monitor the system runtime behaviour and track problems. We have performed different analysis tests to measure the response time for different configurations.

In the first experiment, the single point query is examined using 1000 measures and for all three materialization strategies. To obtain a comparable result, the same points are queried in all three cube materialization modes. The results show that the base case is the minimal cubing mode, since in that case the history size is smaller than with the two others. Figure 22 depicts the comparisons of these three methods.

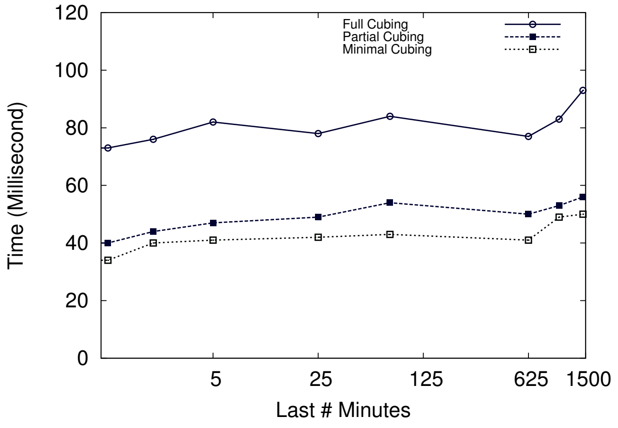

Fig. 22: Response time for single point queries.

A similar comparison is undertaken for the range queries. As explained earlier, one of the continuations of this work is to support the range queries without having to count and aggregate the values for the whole range. In our solution, this is achieved by performing a few queries for the start and end points of the range and subtracting the values. Figure 23 shows this comparison. To obtain the comparison results, different time periods are examined within the last n minutes of the test. For instance, the values in time point 5, are obtained by testing time ranges within the last 5 minutes. Similarly, the results for point 25 are obtained by using random time points within the last 25 minutes. Since the time units are chosen randomly, they could occur in one or more time units. Further, the same ranges are used for all three cases. The results

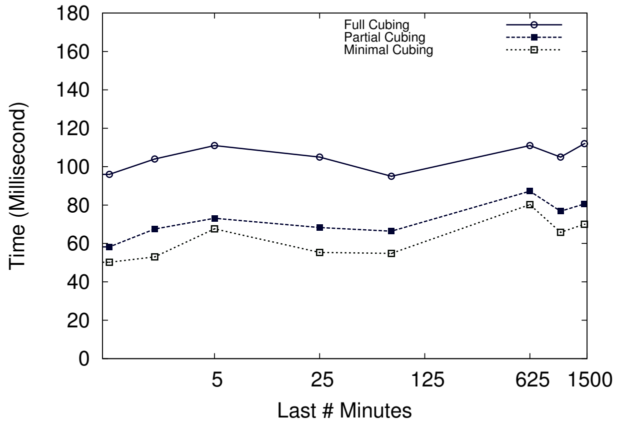

Fig. 23: Response time for range queries.

show that the time required to perform the range queries is related to the number of measures and materialization strategy, not the time interval duration.

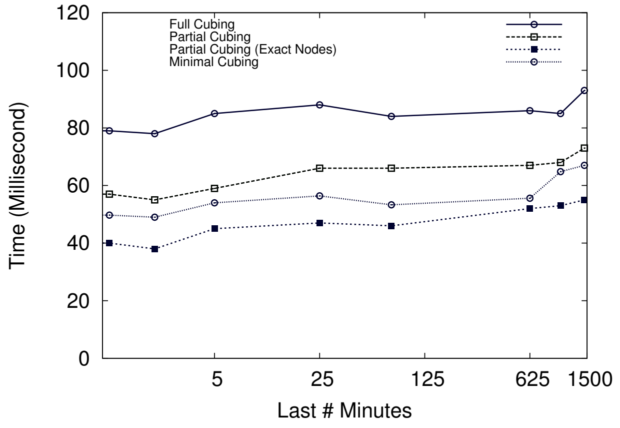

Fig. 24: Response time for roll-up queries.

Figure 24 shows the response time for roll-up queries, to extract a high-level value given the low-level values. In the minimal case, to compute any higher level measure, the values from the leaf level (lowest level) should be extracted from the history data store and then aggregated on-the-fly. For the full cubing, all the values already exist in the history and simply need to be extracted. The partial cubing is somewhere in-between: some non-leaf measures have their own values in the history data store and some must be treated like the minimum cubing. We separate these two cases and consider them separately. Due to the values shown in Figure 24, the best query time belongs to the partial cubing with the 10% measures that data are stored. The minimal cubing method is similar but requires slightly more processing time. The results show that carefully choosing the non-leaf measures is an important factor, and can affect the response time. In this comparison, the same time points are used for all four cases and the results are computed for the single point queries.

In summary, partial cubing with carefully chosen non-leaf measures is considered the best solution. The memory usage for this method is almost equal to the minimal cubing, but the partial cubing has the best response time. However, selecting the potential measures to keep in non-leaf nodes is not an easy task. They can be chosen statically by a system expert or dynamically based on the users' feedback and experience. The memory requirements for the metrics (depending on the associated events) or usage statistics could be two important factors in dynamically selecting the non-leaf measures to materialize.

7. CONCLUSION AND FUTURE WORK

In this paper, a multilevel architecture, and corresponding data structures and algorithms are proposed to construct a cube storage for very large, theoretically unlimited, trace streams to enable different multilevel trace analyses. Reasonable memory usage, efficient response time and support of different query types (single point, range queries, drill-down and roll-up, sliding window queries) are important features of the proposed approach. A customized form of a so-called tilted time frame is used to compress the time dimension. In this configuration, a separate cube is constructed for each time frame, where the cubes for the most recent times are kept more detailed, while the cubes for older times are kept less detailed.

Each cube stores the statistics values in the interval forms (it stores the value and also the time range that value is valid) instead of storing the single values. Storing the data in interval forms enables the time range queries, not only within a cube, but also between the different cubes. This feature supports querying the system for any given time range of the input stream.

We have tested the proposed solution by using a stream of execution trace events gathered by the LTTng kernel tracer. The results show the possibility and efficiency of of performing OLAP-based multilevel multidimensional analysis over a live trace stream. Having this possibility, this technique may be extended to monitor the system runtime behaviour and detect different host and network based problems and attacks.

Several experimental results indicate the memory usage and response times of the proposed method for different cases and configurations. The results generally show that the memory and speed of the proposed method is reasonable and efficient. Indeed, for the range queries of any arbitrary length time ranges, the results show that the response time is unrelated to the size of the range. This achievement is important as the proposed solution enables us to efficiently perform long-lived historical (time-based) queries.

We defined the partial cubing using statically defined metrics. However, a possible future work will be to dynamically choose the non-leaf cuboids to be materialized, or to dynamically switch between two solutions (minimal and partial) based on the users' feedback or the defined queries. Extending the proposed solution to detect system problems and conduct complex analyses with data mining techniques is another possible future work.

Acknowledgment

The support of the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson Software Research, and Defence Research and Development Canada (DRDC) is gratefully acknowledged.

8. REFERENCES

[1] L. Golab and M. T. Özsu, "Issues in data stream management," *SIGMOD Rec.*, vol. 32, pp. 5–14, June 2003.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, (New York, NY, USA), pp. 1–16, ACM, 2002.

[3] J. Han, Y. Chen, G. Dong, J. Pei, B. W. Wah, J. Wang, and Y. D. Cai, "Stream cube: An architecture for multi-dimensional analysis of data streams," *Distrib. Parallel Databases*, vol. 18, pp. 173–197, Sept. 2005.

[4] N. Ezzati-Jivan and M. R. Dagenais, "A framework to compute statistics of system parameters from very large trace files," *SIGOPS Oper. Syst. Rev.*, vol. 47, pp. 43–54, Jan. 2013.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, (New York, NY, USA), pp. 1–16, ACM, 2002.

[6] X. Lin and Y. Zhang, "Aggregate computation over data streams," in *Proceedings of the 10th Asia-Pacific web conference on Progress in WWW research and development*, APWeb'08, (Berlin, Heidelberg), pp. 10–25, Springer-Verlag, 2008.

[7] A. M. N. M. S. Abdellatif, J. Slonim, M. J. McAllister, *et al.*, "Apparatus and method for dynamically materializing a multi-dimensional data stream cube," Sept. 20 2011. US Patent 8,024,287.

[8] Y. Pitarch, A. Laurent, M. Plantevit, and P. Poncelet, "Multidimensional data stream summarization using extended tilted-time windows," in *Proceedings of the*

2009 International Conference on Advanced Information Networking and Applications Workshops, WAINA '09, (Washington, DC, USA), pp. 250–254, IEEE Computer Society, 2009.

[9] L. Tu and Y. Chen, "Stream data clustering based on grid density and attraction," *ACM Trans. Knowl. Discov. Data*, vol. 3, pp. 12:1–12:27, July 2009.

[10] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu, "Mining frequent patterns in data streams at multiple time granularities," 2002.

[11] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang, "Multi-dimensional regression analysis of time-series data streams," in *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pp. 323–334, VLDB Endowment, 2002.

[12] K. Patroumpas and T. Sellis, "Multi-granular time-based sliding windows over data streams," in *Temporal Representation and Reasoning (TIME), 2010 17th International Symposium on*, pp. 146–153, 2010.

[13] A. Arasu and G. S. Manku, "Approximate counts and quantiles over sliding windows," in *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '04, (New York, NY, USA), pp. 286–296, ACM, 2004.

[14] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium) 2006*, pp. 209–224, 2006.

[15] N. Ezzati-Jivan and M. R. Dagenais, "A stateful approach to generate synthetic events from kernel traces," *Advances in Software Engineering*, vol. 2012, April 2012.

[16] A. Guttman, "R-trees: a dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, pp. 47–57, June 1984.

[17] A. Chan, W. Gropp, and E. Lusk, "An efficient format for nearly constant-time access to arbitrary time intervals in large trace files," *Scientific Programming*, vol. 16, pp. 155–165, April 2008.

[18] A. Montplaisir, "Stockage sur disque pour acces rapide d attributs avec intervalles de temps," Master's thesis, Ecole polytechnique de Montreal, 2011.