Moirae: History-Enhanced Monitoring

Magdalena Balazinska[1], YongChul Kwon[1], Nathan Kuchta[1], and Dennis Lee[2]

[1]Department of Computer Science and Engineering
University of Washington, Seattle, WA
{magda,yongchul,nkuchta}@cs.washington.edu

[2]Marchex Inc.
Seattle, WA
{dlee}@marchex.com

ABSTRACT

In this paper, we investigate the benefits and challenges of integrating history into a near-real-time monitoring system; and present a general purpose continuous monitoring engine, called Moirae, that supports this integration. Moirae is designed to enable different types of queries over live and historical data. In particular, Moirae supports (1) queries that look up *specific historical information* for each newly detected event and (2) queries that complement new events with information about *similar past events*. Moirae focuses on applications where querying a historical log in its entirety would be too slow to meet application needs, and could potentially yield an overwhelming number of results. The goal of the system is to produce the most relevant approximate results quickly and, when necessary, additional more precise results incrementally. In this paper, we discuss the challenges of integrating history into a continuous monitoring engine, present the design of Moirae, and show how our proposed architecture supports the above types of queries.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing*

General Terms

Design, Algorithms

Keywords

Stream processing, continuous processing, stream archives, history

1. INTRODUCTION

Monitoring applications enable users to continuously observe the current state of a system, and receive alerts when interesting combinations of events occur. Monitoring applications exist in various domains, such as sensor-based environment monitoring (*e.g.*, air quality monitoring, car-traffic monitoring), military applications (*e.g.*, target detection, platoon tracking), network monitoring (*e.g.*, intrusion detection), and computer-system monitoring. Although the current state of the system is the focus of monitoring applications, when events of interest occur, *historical information* is usually necessary to obtain further information about these events, explain them, and determine appropriate responses.

In this paper, we investigate the benefits and challenges of integrating history into a continuous monitoring system and we propose the design of a new engine, called Moirae, that supports such integration. We focus on continuous *event-detection systems*, where the goal of continuous queries is to detect interesting events. However, our definition of an event is broad and includes any output result produced by a continuous query.

Today, Stream Processing Engines (SPEs)[1] [1, 2, 11, 16, 17, 20, 32] support monitoring applications by providing efficient near-real time processing of information that "streams-in" from the monitored environment. SPEs focus on processing data directly as it arrives without storing it first. As such, they offer high-performance and low-latency continuous processing, but limited or no integrated access to history. Some SPEs have started to recognize the need for exploiting history while processing new data and have started to propose techniques to achieve certain forms of integration [1, 11, 20]. Our goal is to go much further than previous schemes. In Moirae, we treat integrated processing of live and historical data as the fundamental query model in the system. We strive to offer support for a variety of queries over both types of data, and we explore novel ways in which history can enhance continuous monitoring.

There are many scenarios where historical information is a useful component of continuous monitoring. As an example, consider a researcher conducting long-running experiments on a shared testbed such as PlanetLab [35]. This researcher may want his experiments to migrate automatically from one node to another upon failure or overload. However, he may want the experiments to migrate to nodes that are lightly loaded and are likely to remain in that state. He may thus want to run a continuous query of the form: "If a node running my experiment fails, find another lightly loaded node that is running only experiments that have used few resources in the past." Upon detecting a node failure, the query produces the set of currently lightly loaded nodes. Additionally, for each lightly loaded node, the query looks up historical information about the experiments running at that node. This example illustrates the most basic type of integration between real-time events and historical data: an arbitrary SQL query that executes over the data archive as part of the continuous event-detection query. We call this type of integrated query, a *standard hybrid query*.

In addition to standard hybrid queries, exploiting data archives opens the door to new types of queries. As an example, consider an administrator of the PlanetLab testbed who receives alerts when

[1]These engines are also called data stream management systems (DSMS) [2, 32] or continuous query processors [11].

3rd *Biennial Conference on Innovative Data Systems Research (CIDR)* January 7-10, 2007, Asilomar, California, USA.

servers under her control fail. To determine the cause of a failure, the administrator may want to see, with each alert, *similar* alerts that occurred in the past, the context of these alerts, and the solutions that were applied to these failures. Similar alerts could be those where the type of failure was the same and the state of the system was similar (*e.g.*, a subset of the same users were logged-in and a subset of the same processes were running, even though the failure occurred on a different server or even on a different subnetwork). By comparing the *contexts* of the current and past alerts, the administrator may quickly determine which process and user are causing the problem and how to fix it. Furthermore, if the number of similar past events is large, the administrator may want to see only the k most similar ones. This example illustrates a different use of historical data. In this case, the historical query is not a standard query. Instead, it exploits the context of the newly detected event to find the k past events with the most *similar* contexts. We call these queries, *contextual hybrid queries*.

These two examples illustrate the opportunity for different types of integration between continuously streaming data and data archives. Standard hybrid queries provide flexible access to *specific* past information. Contextual hybrid queries explore the use of event context to reduce the volume of relevant historical data by identifying *similar* past events. The goal and innovation of Moirae is to support these different types of queries in a single framework.

In this paper, we present the principled design of Moirae, which integrates several techniques. To compare the similarity of different event contexts, Moirae leverages techniques from information retrieval (IR). To avoid overwhelming users with excessive numbers of results, for each newly detected event, Moirae produces only the k most similar past events according to a *context similarity* metric that we define. To maintain low-latency processing and produce the most important data first, Moirae partitions history and processes recent partitions sooner and with higher priority than older ones. Moirae refines historical information incrementally until it fully answers a query or until the user explicitly indicates that an event is no longer interesting. Moirae's modified stream processing operators support incrementally produced history, enabling historical data to be further combined and correlated with live streams. When new queries enter the system, Moirae pro-actively preprocesses the recent past, to further ensure timely retrieval of recent historical data. Finally, Moirae ensures that all ongoing events benefit from at least some historical information, even when they must contend for systems' resources.

Moirae is currently being developed at the University of Washington. It is a general-purpose engine that does not rely on any domain specific knowledge or models. Moirae uses the Borealis [1] stream processing engine (SPE) for continuous monitoring and PostgreSQL [45] for the historical log. Moirae modifies and tightly integrates both engines.

The rest of this paper is organized as follows. In Section 2, we present the challenges of exploiting history during continuous monitoring and overview existing techniques. We describe the types of queries that Moirae is designed to support in Section 3 and present Moirae's system architecture in Section 4. We present related work in Section 5 and conclude in Section 6.

2. CHALLENGES

The challenge in exploiting historical information as part of continuous stream processing comes from the sheer volume of historical data. Monitoring systems can easily produce gigabytes and even terabytes of data every day. Some processing engines use history to build a model of the monitored environment [26] and, at runtime, compare the current state of the monitored environment to the model. These engines, however, do not support arbitrary queries over historical data.

TelegraphCQ [10] and HiFi [20] are continuous monitoring engines with support for integrated queries over streaming and archived data. TelegraphCQ refers to continuously arriving data as *live* data and to historical data as either *historical* or *archived* data [10]. Queries over both live and historical data are called *hybrid* queries. We use the same terminology in this paper. TelegraphCQ supports different types of integrated queries, as we discuss in Section 5, and proposes techniques for sampling streams during archival or retrieval. When an arbitrary query executes over the archive, TelegraphCQ assumes the data is properly indexed and the query can execute sufficiently fast to keep-up with live data. Frequently, however, historical queries are diverse and it may not be possible to maintain necessary indexes for all queries; queries that do not match an index will take a long time to execute. Additionally, for contextual hybrid queries, querying the entire archive may produce an overwhelming number of results. With Moirae, we propose to push the integration much further and effectively support diverse hybrid queries, including the novel queries that exploit context similarity to extract relevant past information. More specifically, Moirae addresses the following challenges:

Responsiveness and fairness. In a naïve implementation, queries against a growing data archive quickly become slow and either slow-down live stream processing or produce results long after they are needed. For example, if the system scans a massive archive to extract historical information about processes running on lightly loaded nodes, a user's task may not migrate to another node for minutes or even hours after a failure. Instead, the system should produce approximate results quickly and, if necessary, additional, more precise results incrementally. This approach would enable the task to migrate quickly to a good candidate and, if further processing uncovers that the candidate is not quite good enough, to migrate again to a better node. To achieve this goal, we can leverage techniques from online query processing [23, 24, 25, 36, 40]. These techniques sample stored data instead of scanning it. They incrementally produce increasingly more accurate and more complete results. Online query processing can produce at least some historical data sufficiently fast to correlate it with live data. The question is what to do with subsequent updates. Normally, stream processing operators perform their computations over windows of data that slide with time. By the time updates to historical data arrive, operators have completed processing the corresponding windows. Techniques exist for operators to process updates by retrieving their earlier state from upstream or downstream connection points [37] or rebuilding their state from a checkpoint [8]. Because each initial tuple is always followed by a group of revisions, Moirae uses instead a simple technique where operators keep tuples in their state until it becomes known that they will no longer be modified. Moirae also offers users the flexibility of joining updated historical data with either new or old live data, depending on query semantics.

One component of responsiveness is fairness. At any time, multiple continuous queries execute within Moirae, producing requests for historical data. Ideally, each request should receive timely historical information. To achieve this goal, Moirae uses a flexible scheduler that allocates resources among all ongoing historical queries in a manner that ensures the timely extraction of at least some historical information for each newly detected event.

Relevance. One approach to producing partial results is to uniformly sample historical data. All historical data, however, is not uniformly relevant to an event. For example, a node running experiments that recently used few resources is likely to be a better candidate for migration than a node running experiments whose re-

source consumption has been approximated with a small uniform sample over the past two years of measurements. In general, we observe that recent historical data is more important than older data and should be returned first. Recent data is also more frequently requested by historical queries. The system should thus put more resources into ensuring that recent data can be retrieved more effectively. To achieve this goal, we optimize Moirae's design to prioritize recent historical data. We incorporates techniques from multi-level data management [33, 43] and partial indexing [38, 39, 42]. These schemes let a DBMS selectively index data or keep subsets of data in memory. Moirae keeps current data in memory and pushes older data to disk. Moirae, however, ensures that recent data on disk is accompanied by materialized views and indexes. Older data is also on disk but any associated views and indexes may be out-of-date with respect to the current workload. During query execution, Moirae processes data one chunk at a time in reverse chronological order. Within a chunk, it processes data in order as stream processing operators rely on that order.

Similarity. A large fraction of relevant historical information for an event corresponds not just to recent data but to those times in the past when the state of the system was the same or similar to the state at the time of the event. Of course, we want to compare only those parts of the state that are relevant to the current event (*e.g.*, the list of logged-in users and the list of running processes). We call this part of the state the *context* of the event. Moirae supports complex context definitions, involving multiple relations, by allowing users to specify *a set of queries* that together produce the set of tuples forming the context of an event. Moirae compares event contexts using techniques adapted from information retrieval (IR).

Because the historical log is large, when looking for similar past events, the size of the query result can be overwhelming to applications. For example, large numbers of similar failures can occur in a given year. To avoid overwhelming the user, Moirae's goal is to extract only *a small set of k most similar* events and their own contexts . These types of queries are often called k-NN queries as they retrieve the k nearest neighbors of an object. Here the object is the current event and its context. The k nearest neighbors are the k past events with the most similar contexts. Supporting such k-NN queries is challenging. Because the historical log is large, the straightforward solution of computing all past events with their contexts either during query execution or when a query is first deployed would impose a large runtime overhead. On the other hand, we argue that accurate results are not necessary. For many applications, rapid access to k events among the most similar and most recent ones is more important than an exact set of k most similar events returned with low latency. Moirae supports *approximate* k-NN queries that return the best results among the most recent ones by exploiting the partitioned history and query execution described above. If necessary, Moirae further improve results incrementally.

Overall, Moirae uses a set of different techniques that together form an integrated system for effectively exploiting historical data during live stream processing. In Section 4, we describe Moirae's design and discuss how it addresses the above challenges. But first, we present Moirae's queries in more detail in the next section.

3. TYPES OF QUERIES

In this section, we present the types of queries that Moirae is designed to support. We first describe how applications specify continuous monitoring queries, which we call *event-queries*. We then show how event-queries can be extended with subqueries that extract historical information. The extended queries correspond to what we call standard hybrid queries. Third, we turn toward the novel types of queries that extract similar past events for each newly detected event. We show how applications specify the context of an event by specifying what we call *context-queries*. We also present the resulting contextual hybrid queries. Finally, we discuss techniques for computing the similarity between two event contexts.

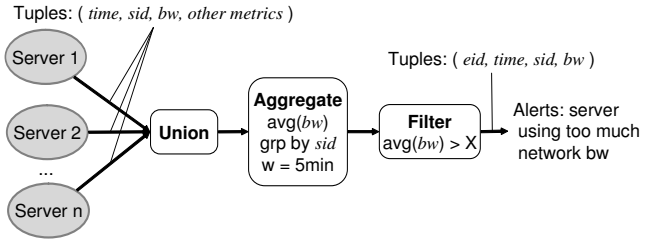

Figure 1: Example of continuous query (event-query).

3.1 Event-Queries

We define an event as a tuple in a stream that takes the form: `(eid, timestamp,` a_1`, ..,` a_n`)` where `eid` is an attribute that uniquely identifies the event, `timestamp` is the time when the event occurred, and $a_1, \ldots, a_n$ are the other attributes of the event.

An event-query is a stream processing query expressed in the language of an SPE. For a stream processing query to be an event-query, the only requirement is that it produces tuples with unique `eid`'s and `timestamps`. These attributes are defined by the user as part of the query. For the continuous stream processing part of the system, we use the Borealis SPE [1]. In Borealis, applications express continuous queries with a boxes-and-arrows data flow, where boxes represent operators and arrows stand for streams. We thus express event-queries as boxes-and-arrows diagrams. Figure 1 shows an example event-query. This query produces an alert when the 5-minute average network traffic generated by a server exceeds a pre-defined threshold, X. In this example, `eid` is the unique event identifier, `time` is the event timestamp, `sid` denotes the server identifier, and `bw` is the bandwidth utilization of the server.

3.2 Standard Hybrid Queries

A standard hybrid query intertwines an event-query with one or more historical queries, which are standard SQL queries over the data archive. Figure 2 shows an example of standard hybrid query based on the event-query from Figure 1. For each event produced by the event-query, the historical query looks up the average and standard deviation of the historical network utilization of the server. The hybrid query then filters out alerts where the resource consumption is within some historical norm. The query over the data archive is encapsulated in a new operator that we call *Recall*. The Recall operator is analogous to Aurora's Read operator [7]. For each input tuple, e, Recall executes a pre-defined SQL query. However, as we discuss later, the Recall operator has different properties than a standard Read operator. In the example, the Recall operator could be executing the following SQL query:

```
SELECT h.sid, avg(h.bw), stddev(h.bw)
FROM History h
GROUP BY h.sid
HAVING h.sid=e.sid
```

where `stddev(bw)` is a user-defined function that returns the standard deviation of all values for a given attribute, `History` is the relation corresponding to the raw stream archive, and `e.sid` is the `sid` of the server identified in the input event `e`. Historical queries can execute over the raw stream archive, or over other currently-defined intermediate streams. These intermediate

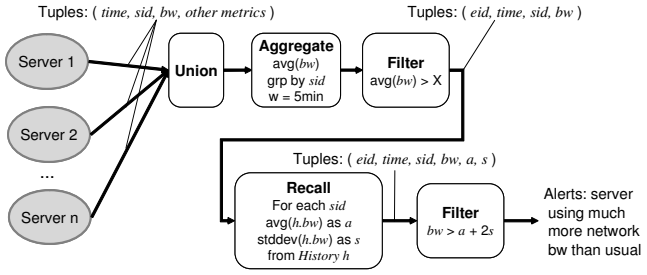

Figure 2: Example of standard hybrid query.

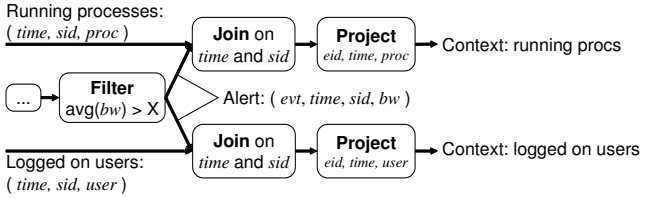

Figure 3: Example of query extracting the context of an event (context-query).

streams must explicitly be archived by inserting *Archive* operators into the query diagram.

Standard hybrid queries are thus simply event-queries that include one or more Recall operators. Their output is a stream that can further be combined and correlated with other streams. Note that the system does not know nor care whether a stream processing query is an event-query until the output of that query is connected to the input of a Recall operator. At that time, the user must specify which attributes serve the role of event identifier and timestamp.

3.3 Context-Queries

We define the context of an event to be *a set of tuples*, each one of the form: `(eid, timestamp,` c_1`, ...,` c_n`)`, where `eid` and `timestamp` are the identifier and timestamp of the event, and $c_1, \ldots, c_n$ are the attributes of the tuple in the event context. Different tuples that belong to the same event context can have different attributes.

To specify the context of an event, applications submit one or more *context-queries* that join each output tuple produced by an event-query (*i.e.*, each event) with tuples on other streams that fall within the same time-window. Using terminology from temporal databases [41], these other tuples are basically all *valid* tuples at the time when the event happens (*e.g.*, the users that are logged on, the system load at the time of the event). Context-queries can also include arbitrary SQL queries over small static relations. For example, a context-query can look up the specifications (CPU, memory, etc.) of the server experiencing the failure. A subset of attributes of the event itself, such as the name of the failed server, can also be part of the event context. Context-queries can even be standard hybrid queries. For a stream processing query to be a context query, the only requirement is that it produces tuples of the form described above. The union of all tuples that satisfy a group of context-queries and share an `eid` forms the context of the event with the given `eid`. The context of an event can be the empty set. Figure 3 shows an example of context defined as the set of logged-in users and running processes.

In most cases, applications will also need to specify additional queries for information surrounding events other than context-queries. Typically, users will request the sequence of events preceding or following each alert. Such additional information can be treated in the same manner as the context, with the only exception that the resulting tuples are not used in the similarity comparison.

Figure 4: Similarity Recall operator.

Finally, in some applications, the context of an event, or at least some components of that context, can be independent of the event itself. For example, the context of a server failure could include the overall request-rate on the system, the time of day, the time since the release of a new feature, etc. Using explicit continuous queries to compute the context of an event is sufficiently flexible to enable such context definitions as well.

3.4 Contextual Hybrid Queries

A contextual hybrid query comprises the following subqueries (a) an event-query, as described above; (b) one or more context-queries, as describe above; and (c) a new operator, called *Similarity Recall*. Similarity Recall takes an event-query and context-queries as input. For each newly detected event, it retrieves from history all events of the same type along with their contexts. It then compares the context of these old events with that of the new event and returns past events ranked by context similarity. Figure 4 shows a Similarity Recall operator. The events and contexts output by Similarity Recall or of the same type as its inputs with one exception: each output tuple also carries the identifier of the newly detected event that it complements.

As above, the system does not know nor care that a query is a context-query unless its output is connected to the input of a Similarity Recall operator. At that time, the user must specify which attributes serve the role of the event identifier and timestamp.

We present the Recall and Similarity Recall operators further in Section 4.5.

3.5 Computing Context Similarity

As described above, the context of an event is a set of tuples coming from one or more streams and relations. Therefore, computing a similarity score for two contexts corresponds to computing a similarity score for two sets of tuples.

Different techniques for computing context similarity are possible. We propose to use a technique from information retrieval. We consider each context as a document, where the tuple attribute-values correspond to terms.[2] We measure the similarity between contexts by measuring their cosine similarity [6]. The cosine similarity metric from information retrieval has successfully been used in the past for ranking query results in a database [3]. As we discuss in Section 5, the main difference with our approach is that we compare groups of tuples together rather than comparing individual tuples to a query. As such, our problem is even more closely tied to information retrieval.

With the cosine similarity metric, two contexts are similar to each other if they contain a larger number of the same "terms"

[2]Each term is a concatenation of the attribute name and the attribute value. With this technique, if two different attributes have the same value (*e.g.*, a user name and a process name), they still remain distinct terms.

(same attribute-values in our case). If two contexts contain the same *rare* term, that term is weighted more heavily. Intuitively, in our computer-system monitoring scenario, if a rare process is executing when a given type of failure occurs, then past events with this same rare process should be considered as more relevant to the newly detected event. We compute the weight of each attribute-value as its TF-IDF product [6]. TF denotes the term frequency, or, in our case, the number of times an attribute-value appears in the context of an event. IDF denotes the inverse document frequency, or, in our case, the frequency of event contexts containing that attribute-value. We use the following two standard formulas:

$$IDF_k = \log\left(\frac{E}{C_k}\right) \quad \text{and} \quad w_{kc} = \mathrm{TF}_{kc} \cdot \mathrm{IDF}_k$$

where IDF_k is the inverse document frequency of attribute-value k, E is the total number of events in the log, C_k is the number of event-contexts containing attribute-value k, TF_{kc} is the frequency with which attribute-value k occurs in context c, and w_{kc} is the resulting weight of attribute-value k in context c. To increase the accuracy of the IDF_k values, we compute these values separately for each *type* of event. By doing so, values that are infrequent in the context of a certain type of event are weighted more heavily even if they appear frequently in the context of other events.

The cosine-similarity score between two contexts is the value of the cosine angle between the vectors of weights of the two contexts, where each attribute-value corresponds to one dimension. The cosine-similarity of two contexts C_1 and C_2 is thus given by:

$$\mathrm{SIM}(C_1, C_2) = \frac{\left(\sum_{k=1}^{k=n} w_{kc_1} w_{kc_2}\right)}{\left(\sqrt{\sum_{k=1}^{k=n} w_{kc_1}^2}\right)\left(\sqrt{\sum_{k=1}^{k=n} w_{kc_2}^2}\right)}$$

where n is the total number of attribute-values in the two contexts.

Figure 5 shows an example of similarity computation. In this example, the contexts of events e_2 and e_3 are most similar as they share a common logged-in user and running process, and all contexts are roughly of the same magnitude.

The TF_k for each attribute-value k in a context can be computed once and stored with the context. Computing the IDF_k score for each element of the domain and for each event type requires scanning all events of that type. Instead, we propose to compute these values incrementally. As a new event occurs, we increment the total number of events, E, and, for each attribute-value k in the event, we update the total number of contexts, C_k, containing k. We then recompute IDF_k for each k.

For continuous-domain attributes (such as temperature readings), we use the "generalized IDF similarity for numeric data" metric defined by Agrawal *et al.* [3]. They use kernel density estimation techniques to define extensions to the cosine similarity metric for numeric attributes. They derive the following equation:

$$\mathrm{IDF}_k = \log\left(\frac{n}{\sum_{i=1}^{n} e^{-\frac{1}{2}\left(\frac{k_i - k}{h}\right)^2}}\right)$$

where n is the number of tuples in the database, the k_i's are all the values of the attribute that appear in the database, and h is a bandwidth parameter [3]. They also weigh the similarity of two attribute-values by a factor that captures a notion of "distance" between the compared values:

$$e^{-\frac{1}{2}\left(\frac{k_1 - k_2}{h}\right)^2}$$

where k_1 is the value of the attribute in context c_1 and k_2 is the value of the attribute in c_2. These extended metrics nicely capture similarity as a function of the numeric distance between values. The only problem is that computing IDF_k for a new numeric value k requires scanning all earlier events. Since we do not need exact values, however, we can maintain a synopsis structure over the historical data [5] (*e.g.*, a histogram) and use that structure to compute an approximate value of IDF_k.

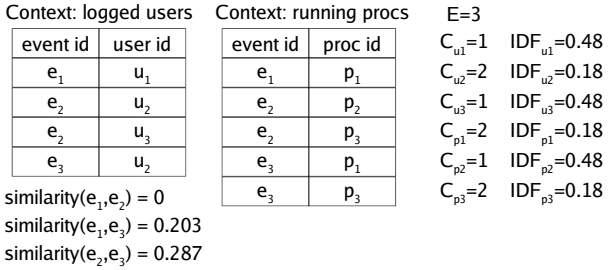

Figure 5: Example of similarity computation for three event contexts.

We thus compute the similarity of contexts by treating them purely as bags. Bags with more common categorical values, especially rare values, or numerically closer attribute-values have higher similarity scores. With this approach, we do not exploit any possible time-series structure of event contexts. Indeed, we consider that events contexts will typically not have any such structure. We envision that an event context will contain the attribute-values of various entities involved in the event or present in the environment at the moment when the event occurred. Such contexts will be simple bags of attribute-values without structure.

There are other possible techniques for comparing and exploiting event contexts. We plan to investigate such techniques in future work.

4. Moirae DESIGN

In this section, we describe Moirae's design. We present an overview of Moirae's system architecture, shown in Figure 6, and discuss the details of the different components.

At a high level, Moirae is a continuous query processor that can handle hybrid queries. It consists of an SPE for continuous stream processing and an RDBMS for the storage of historical streams, events, and contexts. Moirae, however, integrates and extends both engines in several ways.

Moirae's users submit hybrid queries in the form of Borealis boxes-and-arrows diagrams as described in the previous section. We assume that users know about Recall operators and explicitly place them in appropriate locations in the query diagram.

To support hybrid queries, Moirae adds a new *Archiver* component to the storage manager of the RDBMS. This component enables archival and retrieval of streams in a manner that effectively supports hybrid queries. The Archiver is essentially an access method. We present Moirae's overall history-processing technique and the Archiver module in Section 4.1.

Inside the SPE, when queries enter the system, the *Deploy Manager* controls their deployment and later their tear down. Moirae modifies this component in three ways. First, the new Deploy Manager splits SQL query templates specified in Recall operators into pieces to facilitate independent processing of different parts of historical data. Second, it infers which raw streams and intermediate streams must be archived, inserts explicit Archive operators into the query diagram, and lets the Archiver know about these streams. Third, the Deploy Manager inserts event and context materializa-

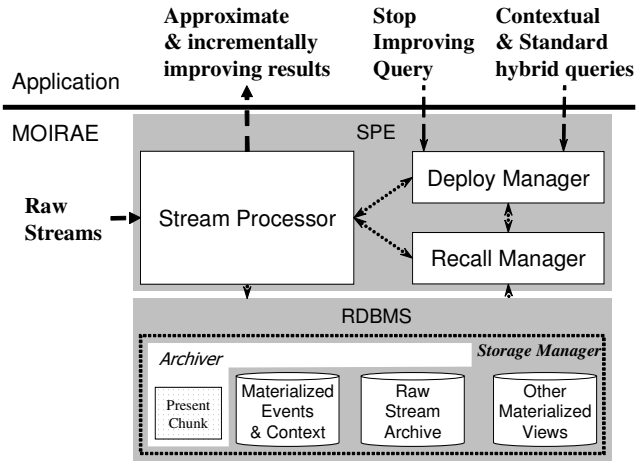

Figure 6: Moirae System architecture.

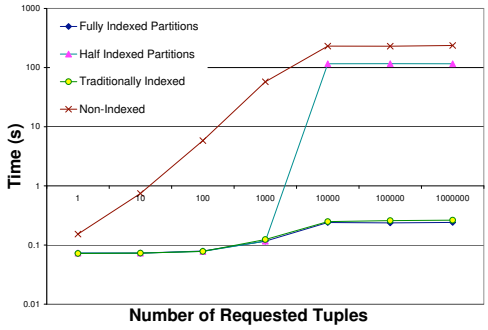

Figure 7: Illustration of performance of partitioned indexing. The "traditionally indexed" and "fully indexed partitions" curves overlap.

tion queries to precompute a subset of past events with their contexts and speed-up their retrieval at runtime. We further discussed why and how the Deploy Manager performs all these tasks in Section 4.2.

Once a new hybrid query is deployed, its event-query runs in the SPE like an ordinary stream-processing query. When events occur that require historical data, Recall operators generate Recall tasks and submit them to a new component, called the *Recall Manager*. This latter component controls the execution of all historical queries requested by Recall operators. A Recall task consists of a parametrized SQL query template and the query parameters, *i.e.*, the associated event and context. For each Recall task, the Recall Manager issues queries to the RDBMS and delivers the results to the originating Recall operator. We discussed the details of the Recall Manager and the Recall task execution in Section 4.3.

While executing hybrid queries, Moirae allows users to tell whether they want higher quality results or whether they are no longer interested in an event. If the user signals that an event is important, the Recall Manager puts more resources into processing Recall tasks related to that event. More importantly, if a user indicates that an event is no longer interesting, the Recall Manager cancels all related Recall tasks and reclaims any associated resources. We discuss the scheduling of Recall tasks and user interactions with the system in Section 4.4. We also provide further details on how Recall operators process the historical data they receive in Section 4.5.

Finally, because Recall operators produce historical data incrementally to ensure responsiveness, stream processing operators downstream from Recall must be modified to support such incrementally improving data streams. We discuss these modifications in Section 4.6.

4.1 Partitioned History

Moirae archives three types of streams: raw streams that enter the system, event and context streams that feed Similarity Recall operators (because these streams will later be reprocessed when looking for similar past events), and any other streams explicitly specified by the user.[3] For the first two types of streams, the Deploy Manager inserts special Archive operators at the appropriate locations into the query diagram. These Archive operators simply forward their input streams to the Archiver. For the last type of archived streams, the user inserts explicit Archive operators into the query diagram.

Raw streams are archived continuously. Intermediate streams, however, can only start to be archived when the user submits a query that defines them. At this point, no history exists for these streams even if the raw stream archive is large. If an event occurs, the lack of history either causes old events to be ignored or imposes a high overhead during query execution because the raw streams are reprocessed to rebuild past events. An alternate technique is for the Deploy Manager and Archiver to reprocess the entire history every time a new intermediate stream starts to be archived. This approach, however, imposes a high runtime overhead every time users define new streams to archive. Clearly, a good strategy lies somewhere between these two extremes.

Furthermore, a naïve approach to exploiting archived data would be for the Recall Manager to execute queries requested by Recall operators over the entire archive at once. Because of the size of the archive, this approach would lead to slow response times for all Recall operators. It could also lead to an overwhelming number of results for Similarity Recall operators.

To address the above challenges, Moirae partitions the archive into three types of chunks: (1) Present chunk: the most recent chunk of the historical log, typically incomplete. The Archiver keeps the present chunk for all archived streams in memory to ensure fast retrieval. (2) Recent chunks: a small set of relatively recent chunks frequently searched when looking for relevant historical data. These chunks are on disk but, for these chunks, Moirae pre-processes and stores all intermediate streams currently archived by continuous queries. This materialization ensures fast historical query execution over these chunks. Moirae can further index these chunks for an even faster access. (3) Old chunks: rarely accessed older parts of the log. These chunks may be accompanied by some older indexes and materialized views, but the system makes no guarantees: any old materialized views may no longer match the current workload. Any access to these chunks is thus much slower than access to more recent chunks. At runtime, Moirae processes chunks incrementally, prioritizing recent chunks over older chunks. The chunk size is defined by the administrator. For example, each chunk could correspond to a few hundred megabytes of data.

As we discuss in Section 5, multi-level storage, partial indexing, and view materialization are all known techniques. The innovation in Moirae is in the integration of these schemes and in the use of the resulting infrastructure for effective execution of hybrid queries as we discuss in Section 4.3.

[3]The user needs to explicitly archive intermediate streams if she wishes to perform standard historical queries on these intermediate streams instead of raw streams directly as discussed in Section 3.2

To illustrate the goals of this partitioning, partial materialization and indexing, Figure 7 shows the results of a simple benchmark retrieving k tuples from a 1GB log partitioned into 5MB chunks. For small k, indexing only half the chunks yields the same query response times (and even a little faster) than indexing the complete log. When k becomes large, some older unindexed chunks must be processed, and the query response time for the extra tuples increases. Our goal is similar, although our queries are significantly more complex. With the hierarchical partitioning approach, Moirae can efficiently retrieve a small set of recent historical data. Additional data can be retrieved if necessary, but at a greater cost. Furthermore, because the cost of pre-processing and indexing only the most recent chunks is small, it justifies changing views and indexes as the workload changes over time.

4.2 Recent Event Materialization

In Moirae, as in other SPEs, users can submit queries at any time. When a user submits a new query, the Stream Processor module needs to materialize recent history for all newly defined streams that feed Archive operators. To do so, the Stream Processor must reprocess all Present and Recent chunks of raw stream data using the appropriate query diagrams.

To let the Stream Processor reprocess the appropriate historical data, the Deploy Manager replicates each fragment of the query diagram that feeds a new Archive operator along with the Archive operator itself. Figure 8 illustrates the approach for the case of a contextual hybrid query. The original query, `CQ1` feeds a Similarity Recall operator and its outputs are thus archived. The Deploy Manager sets-up a replica of that query, `CQ1-replica1`. The outputs of the replica are also archived. The inputs to the replica are Recall operators that simply replay all historical data within a given time interval. The Deploy Manager sets the initial time interval to match the range covering all Recent and Present chunks at the moment when the query was inserted into the system. As we discuss below, the Recall Manager will nevertheless cause the materialization to occur one chunk at a time in reverse chronological order. We discuss the remaining components of Figure 8 when we discuss query execution.

The Stream Processor can schedule these query diagram replicas with lower priority than other operators since their goal is only to improve the performance of upcoming historical queries.

While materializing past events and contexts, Moirae can optionally index the resulting materialized views for an even faster access. It can also create other materialized views in the RDBMS.

4.3 Partitioned Query Execution

At runtime, every time an event requires historical data, the corresponding Recall (or Similarity Recall) operator issues a Recall task to retrieve that data. The Recall task will execute over the hierarchically partitioned data store. As we described above, in this hierarchy, present data is in memory. Recent data is stored on disk, but it is accompanied by various indexes and materialized views that are relevant to current queries. Older data is also on disk, but it may or may not have any relevant materialized views and indexes. Older data may even be missing necessary events and context, requiring the Stream Processor to reprocess the raw data during historical query execution. To benefit from this hierarchical partitioning, the system should thus use different query plans for chunks at different levels in the hierarchy. Additionally, because we consider recent data to be more important, historical data should be retrieved in reverse chronological order.

To achieve these goals, Moirae proceeds as follows. First, the Deploy Manager separates all historical query templates into two

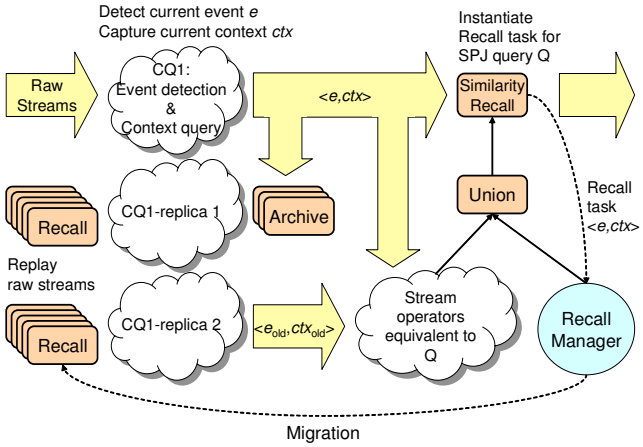

Figure 8: Sample deployment of a query diagram with background materialization of recent historical events and their contexts.

parts: an underlying Select-Project-Join (SPJ) query template and additional operators that must be applied to the result of the SPJ query. For a Similarity Recall, the extra operators correspond to context similarity matching and ranking. For a standard Recall, the extra operators are Group-By and Aggregate. This separation enables Moirae to process different chunks of historical data using different query plans because the results of independent SPJ queries can be merged with a simple union. Recall operators compute their outputs incrementally as the results of the underlying SPJ queries become available. Because their processing is non-monotonic [40], Recall operators output both new tuples and revisions to earlier produced results. We present the processing done by Recall operators in Section 4.5. In this section, we focus on the SPJ queries.

When a new event occurs, the Recall operator constructs a Recall task that it hands over to the Recall Manager. The Recall task contains the SPJ query template and the parameters to instantiate the query *i.e.*, the event and its context. Given a Recall task, the Recall Manager issues *independent* SPJ queries over increasingly older data chunks by adding to the SPJ queries a predicate on tuple archival times. In our task migration example, assuming chunks of size one week, the Recall Manager would first select the per-process resource utilization information for this week, then for last week, then for two weeks ago, and so on. The underlying RDBMS optimizes and executes each such query independently, exploiting the Archiver access method and all other available materialized views and indexes for the given chunk. As queries complete, the Recall Manager issues subsequent queries over older data chunks forwarding the results to Recall operators, which combine and aggregate them incrementally. With this approach, Moirae processes recent data faster and earlier than older data.

One limitation of the approach, however, is that joins can only match tuples within the same chunk (or they can join streams with small static relations that do not have any time attributes). It is often possible, however, to decompose queries that need to join data across chunks into a sequence of Recall operators. For example, in our automatic task migration scenario, the first Recall operator can look up, for each available node, the set of experiments currently running on that node (query over the present chunk only). The following Recall operator can take the resulting set of experiments as input and execute a query that looks up past resource utilization for each experiment (queries over increasingly older chunks).

The Recall Manager uses the above strategy for all Recall operators: Similarity Recall operators, standard Recall operators, and

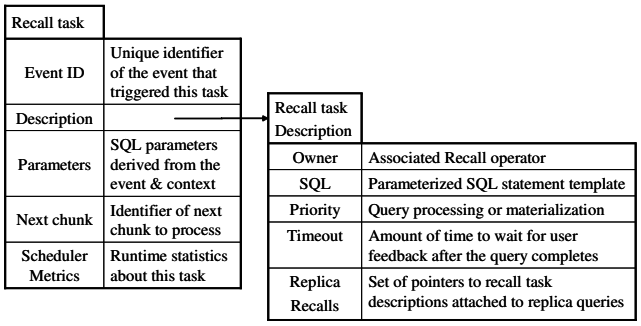

Figure 9: Recall task and Recall task Description structures.

Recall operators that serve to materialize data in Present and Recent Chunks. Hence, all these operators process data one chunk at a time in reverse chronological order. With this approach, the returned results are not only approximate because the system examines only a subset of chunks, but also because events that fall on chunk boundaries may go undetected. To address this problem, one approach is to allow some overlap between consecutive chunks as in LHAM [33]. The overlap must be sufficiently long to reconstruct enough state in stream processing operators to avoid missing events (*e.g.*, using techniques from Hwang *et al.*, [27]). The overlap however, should be small compared to the chunk size and would only be used when re-processing a chunk with the stream processor, not for arbitrary SQL queries.

Similarity Recall operators do not process raw streams. They process intermediate streams of events and contexts output by stream processing queries. Standard Recall operators may also contain SQL queries over intermediate rather than raw streams. In these cases, once all Present and Recent chunks have been processed, if more results are necessary, the Stream Processor needs to re-process older raw data to produce the necessary streams. To achieve this goal, Moirae uses the same strategy as for the initial materialization of data in Present and Recent Chunks: the Deploy Manager sets up replicas of the appropriate query-diagrams. In Figure 8, this replica corresponds to `CQ1-replica2`. This additional query diagram executes only when the Recall Manager notices that the Recall operator needs the additional historical data. As shown in the figure, the outputs of this additional query diagram need not go to disk, they can be streamed directly to the historical query feeding the Recall operator. We expect that few events will require this third level of processing.

4.4 Historical Query Scheduling

During continuous monitoring, multiple events can occur at the same time or quickly one after the other. These events can come from the same query or from different queries executing in the system. Additionally, each event can trigger one or more Recall tasks. Since Moirae tries to behave like an SPE, while providing a better quality of information than an SPE alone, Moirae tries to ensure that each alert and each Recall task is quickly given at least some historical information. To achieve this goal, Moirae must properly allocate resources to the different operators executing concurrently in the system, including the stream processing operators, the Recall operators, and the Archive operators. However, as an initial approach, we delegate stream processing operator scheduling to the Stream Processor and archiving to the RDBMS. We focus only on scheduling Recall operators.

When an alert triggers a Recall operator, the operator generates a Recall task that it submits to the Recall Manager. Figure 9 shows the main attributes of the structure that represents a Recall task and its associated task description. The Recall Manager creates a Recall task Description for each new Recall operator that is deployed. Recall operators create one Recall task for each new event. The Recall Manager schedules all submitted Recall tasks. As shown in Figure 9, each task includes the identifier of the chunk that must be processed next. When the Recall Manager selects a task to run, it issues the corresponding SQL query over the next chunk, and updates the chunk identifier for the task. The SQL query that the Recall Manager submits to the RDBMS includes a predicate on the time interval that matches the timespan of the chunk. We call these individual SQL queries *history-queries* of the Recall task.

To ensure all Recall tasks quickly produce some results, we propose to use the stride scheduling algorithm [49]. This technique is a variant of lottery scheduling. Each Recall task receives *tickets* which can be added or deducted dynamically. Recall tasks with more tickets are scheduled more frequently. Every time a Recall task is scheduled, it executes only one history-query, *i.e.*, it processes only one chunk. After the query completes, the scheduler removes from the task a number of tickets proportional to the number of results produced (thus increasing the stride of the task). Because Recall tasks are scheduled based on the number of tickets they hold, this approach ensures that Recall tasks with few results are scheduled more frequently. The Recall Manager also ensures that no more than a fixed number of historical queries execute within the RDBMS at any time. The threshold is set by the administrator, but could be adjusted automatically based on load.

By processing one chunk at a time, Moirae manages multiple concurrent queries in a manner that ensures all Recall tasks produce at least some historical data. However, it does not prune query processing for non-interesting events and does not dynamically prioritize interesting ones. As in other online query processing systems [24, 25], only the user can tell which events are interesting or non-interesting. The system thus needs to support dynamic inputs from users. Moirae allows users to cancel historical query execution at any time. When a user submits a "cancel request" for an event id, the Deploy Manager forwards the request to the Recall Manager. The Recall Manager in turn cancels all Recall tasks with the given event id and reclaims all associated resources. The user can also submit an "accelerate request" for an event id to indicate that the given event is particularly interesting. To increase the priority of the identified event, the Recall Manager increases the number of tickets associated with that event.

Some Recall tasks need to process only a bounded amount of history (*e.g.*, select the average resource utilization this week) while others do not have any explicit time bound (*e.g.*, select the k most similar past events). To avoid executing the latter queries over the entire archive if possible, Moirae periodically suspends each Recall task and waits for user feedback. Based on the intermediate results, a user can decide to refine the result or cancel the event. The feedback frequency is set by the user when submitting the hybrid query. If Moirae receives no feedback from the user within a given time period, it discards all Recall tasks associated with the event.

When a Recall task runs out of materialized events to process, the Recall Manager triggers the replica of the query diagram that can produce these events. It does so by migrating the corresponding Recall task (with its tickets) to the Recall operators in the query diagram replica. The new Recall tasks thus run with the same priority as the original one.

In addition to ticket based scheduling, Moirae has at least two different priority levels for Recall tasks: a higher priority level for ordinary query processing and a lower priority level for materialization. Moirae enforces ticket-based scheduling among tasks with

the same priority. It schedules lower priority tasks only when the system is lightly-loaded.

In addition to the cardinality of their result so far, the Recall Manager keeps other runtime statistics about each task. These additional statistics can also serve to adjust the number of tickets for each task. In future work, we plan to investigate more sophisticated scheduling algorithms taking into account the quality of results returned so far, the total execution time of queries, or scheduling together queries that need to process the same historical chunks.

4.5 Recall Operators

As historical data streams into Recall operators, the latter must perform additional processing before outputting results. However, to ensure timely processing, they cannot wait to receive all historical data; they must produce results incrementally. The user specifies the desired trade-off between result frequency and latency, by specifying the period between result updates by Recall operators. In this section, we present the additional processing performed by Recall operators.

4.5.1 Recall

As it processes incoming historical data, a standard Recall operator may have to perform additional grouping and aggregation. To avoid delaying historical data, aggregation should be performed online [25]. The challenge, however, is that incoming data is not a uniform sample of the historical period of interest. Instead, the operator receives the most recent history first and older data incrementally, one chunk at a time.

To produce approximate results over the entire time-period, we propose that the operator extrapolates the values received so far. For example, for a sum or a count operation, if a user requested the aggregate value for this week, but the operator only received the values for today, Recall multiplies the current sum or count by seven, approximating the values for earlier days with the value for today. Average, min, max, and count distinct operations require no adjustment. The approximate value for the recent data is used as the approximate value for the entire interval. The semantics of approximate results are thus different than with traditional online aggregation because of the emphasis that we put on recent history. A better technique would be to keep summary structures, such as histograms, to approximate historical data and use them in addition to the recent data to produce more accurate estimates.

To indicate the inaccuracy of output results, the Recall operator annotates the results with the time interval used to compute them.

4.5.2 Similarity Recall

A Similarity Recall operator takes as input a newly detected event with its context and a stream of past events of the same type and their contexts. Similarity Recall compares the contexts of past events with that of the newly detected event.

The query plan for producing the approximate set of k most similar events is relatively straightforward. The streams holding event contexts are already sorted by increasing event identifiers, as this order follows from the sequential processing of the streams. These streams can thus easily be joined together as they are replayed. For each resulting *group of tuples*, the Similarity Recall operator computes the similarity score between each old event and the new event, using the technique from Section 3.5. For each chunk of historical data, the Similarity Recall sorts the resulting events on their scores and outputs the top-k events. As it processes increasingly older chunks, the Similarity Recall operator filters out events less relevant than the ones already produced, by keeping track of the top k events produced for each alert.

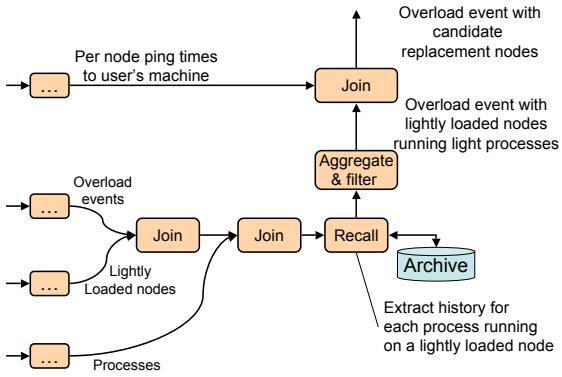

Figure 10: Example of standard hybrid query where the output of a Recall operator is processed further.

4.6 Incremental Stream-History Processing

Because a Recall operator produces historical data incrementally, operators downstream from Recall must be able to handle out-of-order and incrementally improving input streams.

As an example, consider our sample task migration scenario. In this example, tasks should migrate to lightly loaded nodes running only processes that do not use significant amounts of resources on average. Let's extend this scenario by also requiring that the candidate nodes have good network connectivity (*i.e.*, low ping time) to the user's desktop machine. We can express this extended query with a standard hybrid query as illustrated in Figure 10. First, an event-query detects when overload occurs on a node running the user's experiment. Upon detecting such event, the query also produces the currently lightly loaded nodes along with their processes. Second, a Recall operator retrieves the historical resource utilization of each process running on one of the lightly loaded nodes. Third, aggregate and filter operators process the output of Recall to produce the set of candidate nodes that are both lightly loaded and run historically light processes. Finally, a join operator correlates the stream of candidate nodes with the stream of current ping-time values to produce the final set of candidate nodes.

In this example, the output of the Recall operator may change as it processes more history. Indeed, a process that was recently consuming a lot of resources, could have only been experiencing a short spike. It may not use much resources on average. The recall operator will thus need to produce *revision tuples* [1]: *i.e.*, insertions of previously missing tuples and deletions or updates of previously produced tuples. The downstream aggregate, filter, and join operators must process these updates. The challenge is that most stream processing operators (*e.g.*, aggregate and join) perform their computation over windows of data that slide with time. By the time an update arrives, the window of computation has moved.

The Borealis SPE [1] already supports revision tuples. Borealis operators process revision tuples in one of two ways. They either restart from a checkpoint [8] or they update only the affected windows of computation by retrieving historical state from connection points [37]. Because Recall operators always produce a certain amount of revisions following an initial result set, it is more effective for Moirae's operators to keep more state in memory and process both new tuples and revisions simultaneously.

To truncate old state, Moirae's operators use *punctuation* [46, 47] or *boundary* [8] tuples. Boundary tuples are produced periodically by data sources. As they propagate through the system, they en-

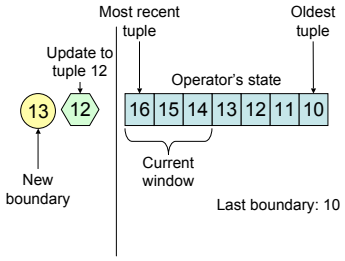

Figure 11: Example of dropping window state using boundary tuples to facilitate revision processing.

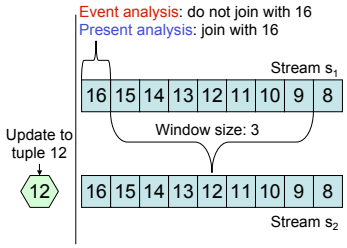

Figure 12: Example of joining a tuple after an update. With event analysis, an updated tuple can only join with tuples within the original window. With present analysis, an updated tuple can also join with more recent tuples.

sures, at each step, that all following tuples will have a timestamp higher than the boundary timestamp (*i.e.*, no more revisions will arrive for tuples with a lower timestamp). Figure 11 illustrates the approach. In the figure, the current window of computation spans tuples with timestamps in the range $[14, 16]$. However, the operator keeps all tuples since timestamp 10, the value of the most recent boundary tuple it received. When an update arrives for tuple with timestamp 12, the operator can readily process the update. When a new boundary tuple arrives with timestamp 13, the operator drops from its state all tuples with timestamps below that value.

Recall operators propagate boundary tuples only after they process an event and all preceding events completely. The timestamp value of the boundary tuple carries the timestamp of that event.[4] All tuples output by a Recall operator that are related to the same event carry the same timestamp value, the timestamp of the event. All these tuples can be pruned once historical queries for the event stop. Since the Recall Manager makes the decision to terminate the historical queries for an event, the Recall Manager produces the boundary tuples and transmits them to Recall operators.

The above approach also enables a new model for processing revision tuples. Revision tuples carry old timestamps equal to the timestamps of the tuples they correct. With existing stream operator semantics, these tuples should join with equally old tuples on other streams. In Figure 12, the updated tuple 12 only joins with tuples in range $[9, 15]$. In our task migration example, as the Recall operator updates its output, a previously unacceptable node can suddenly become a good candidate. When the correction appears on the stream, the join operator will match the updated tuple with the *ping-time value at the moment of the original event* because it joins tuples within the same window. We call this type of processing *event analysis*, because it appears as if time had stopped for the event and all revisions are processed using the valid data at the moment of the event. In some scenarios, this is in fact the desired behavior. For example, this approach is correct if the user wants to study the overlay network and wants to know about nodes that had both light processes and low ping times when an overload event occurred

In our scenario, however, the user does not want to perform event analysis. He wants to take an action (migrate tasks) based on the best data available. In that case, it would make more sense for the join to correlate the newly discovered candidate node with *the most recent ping-time values*. We call this type of processing *present analysis* as the user wants to know about the current state of the system in order to take some action. In the Figure 12, this would mean that the update tuple with timestamp 12 should also join with the most recent tuples with timestamps > 15 on the other stream.

Moirae supports both types of processing by enabling the user to set the desired type of analysis as a parameter to each join operator.

4.7 Additional Techniques

There are several additional issues with integrating historical data into a continuous monitoring system that we did not discuss in the paper. One issue lies in indexing event contexts, where tuples are spread across multiple relations, to speed-up the similarity matching process. As another example, each Recall operator may execute similar historical queries for multiple events that occur in sequence. Caching the results of these different executions could reduce system load. Finally, there exists various sampling and summarization techniques [5, 12] for streams that we could leverage to reduce the size of the log and produce approximate results.

5. RELATED WORK

SPEs [1, 2, 11, 16, 17, 32] focus on efficient, continuous processing of live data. In most systems, queries over historical data are processed independently from queries over live data [4, 28]. Some systems do not focus on stream archives explicitly, but allow joins between streams and stored relations [32] or SQL operations on stored relations as part of the continuous processing flow [7]. They assume, however, that these relations are sufficiently small for the system to keep-up with input data rates. In Aurora [2] and Borealis [1], connection points can store limited stream history, but that history can only be replayed to satisfy ad-hoc queries [2], process tuple revisions [1, 37], or perform time-travel operations [1].

The work most similar to ours is Chandrasekaran's work [10] on supporting *hybrid* queries over live and archived streams. Chandrasekaran supports three specific types of queries: queries that start in the past and continue in the present, queries that access an offset part of the historical log (*e.g.*, compare today's average with yesterday's average), and arbitrary queries over the entire log. For the first two types of queries, Chandrasekaran proposes algorithms for sampling stream data while archiving and retrieving it. For the latter type of queries, his schemes assume the entire log is properly indexed and the query over the log will execute quickly and return a small set of results. With Moirae, we focus on the latter types of queries, but we assume not all queries have a matching index, the log is too large to be queried in its entirety, and most queries will return a large result set.

We reuse several existing database techniques in our work. None of these techniques in isolation suffices to solve our problem. Instead, we must combine these techniques into a novel framework. The techniques include view materialization, which is commonly used to improve query execution performance [22], and partial indexes [38, 39, 42], which enable a database to index only tuples matching a predicate. Materializing and indexing only some pre-

[4]For simplicity, we assume tuples are sorted on their timestamp values on all streams, but this assumption is not necessary [2, 8].

processed parts of history can improve performance in a continuous monitoring system. Indeed, the most suitable views and indexes change with time, yet it would be expensive and often useless to re-process or re-index the entire history every time. To ensure that at least some results are quickly returned, unlike previous work, Moirae exploits partial indexes and materialized views even if they do not match a historical query entirely.

As another technique, the multi-level storage manager [43] integrates a main memory, a disk, and an archive database into a single framework. Moirae's partitioned history is similar in spirit, but it only has two levels and data migrates automatically between levels as it ages. As such, Moirae's storage manager (the Archiver module) is more similar to the LHAM log-structured access method [33], which keeps the most recent data in memory and, as the data ages, moves it in chunks to disk then to tape. The Archiver uses the same strategy. In contrast to both proposals, however, Moirae focuses not only on storing different data on different devices, but also on pre-processing and materializing events differently for present, recent, and old data. The Archiver also serves a role analogous to the OSCAR [12] access method for archiving and retrieving streams. OSCAR, however, focuses on reducing streams through sampling during storage or retrieval. In contrast, the Archiver focuses on keeping and serving recent data from memory. Moirae's Archiver could leverage OSCAR's sampling techniques to handle overload.

Work on online query processing [23, 24, 25, 36, 40, 44] proposes the idea of continuous and incremental query evaluation: early approximate results are produced quickly followed by incrementally more accurate results. Online query processing also introduces the idea of reordering data dynamically while it is being processed based on explicit user feedback [24, 36]. Moirae combines these ideas with the materialization and indexing of subsets of historical events. Furthermore, unlike previous online processing schemes, Moirae emphasizes fairness between historical queries, scheduling them in a manner that ensures all events receive some historical data in a timely fashion. In Moirae, user interaction is based on the notion of events that occur on the live streams rather than on specific values within the historical data.

For contextual hybrid queries, Moirae's goal is to return an approximate set of k most similar past events. There has recently been a significant amount of work on adding support for top-k queries to databases [9, 13, 14, 18, 29]. In particular, the RanqSQL project [29] supports ranking at the database core, enabling rank-aware iterator-fashion query plans that do not necessitate materializing nor sorting entire relations. Chang and Hwang [13] propose an approach for retrieving top-k results with minimal probing. Although both techniques could be useful in our setting, we are fundamentally interested in ranking sets of tuples from multiple relations, or contexts as we name them, instead of individual tuples. Comparing event contexts is also related to the similarity search problem [19, 48]. However, existing techniques focus on comparing individual, multidimensional objects [48] or sequences [19] rather than sets of tuples. Techniques enabling keyword searches over relational databases [21, 30] or combining information retrieval and databases in general [3, 15] are more closely related to our problem, but our goal is not to retrieve a ranked set of tuples matching a set of keywords or a query, it is to compare two groups of tuples.

There exists extensive work in data mining. For example, Horvitz *et al.* built a model of car traffic, JamBayes, over data collected by sensors deployed on highways in Washington state [26]. The work is similar to ours in the sense of considering other contextual information such as weather, events, time, and holidays to predict traffic conditions. Data-mining-based solutions, however, build domain specific models of an environment that they use to answer queries. In contrast, we propose to enable users to see specific past events and ask arbitrary queries over the historical data.

Finally, temporal databases [31, 34, 41] support sophisticated queries over persistently stored temporal data. An important difference in our setting is that, for most queries, the raw stream data must initially be processed in the same manner as the live data by the SPE, to ensure the same events are detected in the same circumstances. We could, however, leverage temporal databases for storing the materialized events and contexts.

6. CONCLUSION

In this paper, we investigated some of the benefits and challenges of integrating history into a continuous monitoring system and we proposed the design of a new engine, called Moirae, that supports such integration. Moirae supports two types of queries that combine live and historical data: standard hybrid queries and contextual hybrid queries. A standard hybrid query lets a user issue an arbitrary query against the data archive given a newly detected event. A contextual hybrid query looks for the same type of events in the past. It compares the context of each past event with the context of the current event and returns an approximate set of k most similar past events.

The key insight behind Moirae's design is that most queries will request recent historical information and users will be more interested in receiving a few relevant results soon after each new event, rather than a complete result set (or the best results) with higher latency. Thus the three major properties that Moirae strives to achieve are *Responsiveness*, *Relevance* and *Similarity*: *i.e.*, to produce the most relevant, recent results with low-latency (and additional results incrementally when necessary) and use context-similarity to identify relevant past data. To achieve these properties, Moirae, uses a combination of techniques including partitioning the archive and prioritizing recent partitions, partitioned and incremental query execution, materialization of recent past events, careful scheduling of historical queries, correlations of incremental historical results with live data streams, and user feedback. To compute event similarity and extract an approximate set of k most similar past events, Moirae uses techniques from information retrieval.

We are currently building a prototype of Moirae, and plan to evaluate it on computer-system monitoring data collected in our department and CoMon/CoTop logs from PlanetLab. We also plan to experiment with network monitoring traces.

Exploiting historical data in continuous monitoring systems is an important problem in many domains. Different types of integrations between live and historical data are possible. We investigated some of them in this paper. As such, we view this work as an important step toward providing fully integrated support for history in a near real-time stream processing engine.

7. ACKNOWLEDGMENTS

We thank the CoMon team for their help with using the PlanetLab monitoring data and Jon Sanislo for helping us get access to the departmental computer-system monitoring data. We thank Samuel Madden and Surajit Chaudhuri for helpful discussions. This material is based upon work supported by Cisco Systems Inc. through the Cisco University Research Program.

8. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proc. of the CIDR Conf.*, Jan. 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), Sept. 2003.

[3] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In *Proc. of the CIDR Conf.*, Jan. 2003.

[4] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In *Proc. of the 30th VLDB Conf.*, 2004.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the 21st ACM Symposium on Principles of Database Systems (PODS)*, 2002.

[6] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbets, and S. Zdonik. Retrospective on Aurora. *VLDB Journal*, 13(4), Dec. 2004.

[8] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *Proc. of the 2005 SIGMOD Conf.*, June 2005.

[9] M. J. Carey and D. Kossmann. On saying "enough already!" in SQL. In *Proc. of the 1997 SIGMOD Conf.*, May 1997.

[10] S. Chandrasekaran. *Query Processing over Live and Archived Data Streams*. PhD thesis, University of California, Berkeley, 2005.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the CIDR Conf.*, Jan. 2003.

[12] S. Chandrasekaran and M. J. Franklin. Remembrance of streams past: Overload-sensitive management of archived streams. In *Proc. of the 30th VLDB Conf.*, 2004.

[13] K. Chang and S. Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *Proc. of the 2002 SIGMOD Conf.*, June 2002.

[14] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *Proc. of the 25th VLDB Conf.*, Sept. 1999.

[15] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR technologies: What is the sound of one hand clapping? In *Proc. of the CIDR Conf.*, Jan. 2005.

[16] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the CIDR Conf.*, Jan. 2003.

[17] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. of the 2003 SIGMOD Conf.*, June 2003.

[18] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4), 2003.

[19] C. Faloustos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of the 1994 SIGMOD Conf.*, May 1994.

[20] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design considerations for high fan-in systems: The HiFi approach. In *Proc. of the CIDR Conf.*, Jan. 2005.

[21] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *Proc. of the 24th VLDB Conf.*, Aug. 1998.

[22] J. Goldstein and P.-Å. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *Proc. of the 2001 SIGMOD Conf.*, 2001.

[23] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *Proc. of the 1999 SIGMOD Conf.*, 1999.

[24] J. M. Hellerstein, R. Avnur, and V. Raman. Informix under CONTROL: Online query processing. *Data Mining and Knowledge Discovery*, 4(4), 2000.

[25] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. of the 1997 SIGMOD Conf.*, 1997.

[26] E. Horvitz, J. Apacible, R. Sarin, and L. Liao. Prediction, expectation, and surprise: Methods, designs, and study of a deployed traffic forecasting service. In *Proc. of the 21st UAI Conf*, July 2005.

[27] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Proc. of the 21st ICDE Conf.*, Apr. 2005.

[28] N. Jain, L. Amini, H. Andrade, R. King, Y. nho Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *Proc. of the 2006 SIGMOD Conf.*, June 2006.

[29] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. RankSQL: query algebra and optimization for relational top-k queries. In *Proc. of the 2005 SIGMOD Conf.*, June 2005.

[30] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *Proc. of the 2006 SIGMOD Conf.*, June 2006.

[31] D. B. Lomet, R. S. Barga, M. F. Mokbel, and G. Shegalov. Transaction time support inside a database engine. In *Proc. of the 22nd ICDE Conf.*, 2006.

[32] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the CIDR Conf.*, Jan. 2003.

[33] P. Muth, P. O'Neil, A. Pick, and G. Weikum. The LHAM log-structured history data access method. *VLDB Journal*, 8(3-4), 2000.

[34] G. Ozsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. Knowl. Data Eng.*, 7(4), 1995.

[35] PlanetLab: An open platform for developing, deploying, and accessin g planetary-scale services. `http://www.planet-lab.org/`.

[36] V. Raman and J. M. Hellerstein. Partial results for online query processing. In *Proc. of the 2002 SIGMOD Conf.*, June 2002.

[37] E. Ryvkina, A. Maskey, M. Cherniack, and S. Zdonik. Revision processing in a stream processing engine: A high-level design. In *Proc. of the 22nd ICDE Conf.*, Apr. 2006.

[38] C. Sartori and M. R. Scalas. Partial indexing for nonuniform data distributions in relational DBMS's. *IEEE Transactions on Knowledge and Data Engineering*, 6(3), 1994.

[39] P. Seshadri and A. N. Swami. Generalized partial indexes. 1995.

[40] J. Shanmugasundaram, K. Tufte, D. DeWitt, D. Maier, and J. F. Naughton. Architecting a network query engine for producing partial results. *Lecture Notes in Computer Science*, Vol. 1997, 2001.

[41] R. Snodgrass and I. Ahn. A taxonomy of time in databases. In *Proc. of the 1985 SIGMOD Conf.*, May 1985.

[42] M. Stonebraker. The case for partial indexes. *SIGMOD Record*, 18(4), 1989.

[43] M. Stonebraker. Managing persistent objects in a multi-level store. In *Proc. of the 1991 ACM SIGMOD International Conference on Management of Data*, 1991.

[44] K.-L. Tan, C. H. Goh, and B. C. Ooi. Online feedback for nested aggregate queries with multi-threading. In *Proc. of the 25th VLDB Conf.*, Sept. 1999.

[45] The PostgreSQL Global Development Group. Postgresql database management system. `http://www.postgresql.org`, 2006.

[46] P. A. Tucker and D. Maier. Dealing with disorder. In *Proc of the Workshop on Management and Processing of Data Streams (MPDS)*, June 2003.

[47] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3), May 2003.

[48] K. Vu, K. A. Hua, H. Cheng, and S.-D. Lang. A non-linear dimensionality-reduction technique for fast similarity search in large databases. In *Proc. of the 2006 SIGMOD Conf.*, June 2006.

[49] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional- share resource management. Technical report, Cambridge, MA, USA, 1995.