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]Website Platform
Amazon.com
{dennisl}@amazon.com

ABSTRACT

In this paper, we present the benefits and challenges of integrating history into a near-real-time monitoring system. In particular, we address the specific problem of enhancing current events with historical information, by efficiently retrieving for each newly detected event similar past events (*e.g.*, car accidents, network intrusions, server failures). We focus 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. We propose a general purpose continuous monitoring system, called Moirae, that efficiently produces with each new event an approximate set of most similar recent events. We present the design of Moirae, show how our proposed architecture achieves the above goals, and discuss the applicability of the architecture to other types of integration between historical data and continuous monitoring.

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), and network monitoring (*e.g.*, intrusion detection). Although the current state of the system is the focus of monitoring applications, when events of interest occur, *historical information* is usually necessary to explain these events, and determine appropriate responses. As an example, consider a computer-system monitoring application, where an administrator receives alerts when servers fail. To determine the cause of a failure, it may help the administrator to see, with each alert, *similar* alerts that occurred in the past as well as the context of these alerts. Similar alerts could be those where the type of failure was the same and the state of the system was similar (*e.g.*, the list of logged users and running processes overlapped, even though the failure occurred on a different server). By comparing the *contexts* of the current and past alerts, the administrator may quickly determine which process and user are causing the problems.

There are many scenarios where historical information is a critical component of continuous monitoring. Today's continuous monitoring engines provide efficient near-real time processing of information streaming-in from the monitored environment [1, 2, 8, 18]. Our goal is to enhance the output produced by these engines with relevant historical information. In particular, we start with the following concrete problem: for each newly detected event, *quickly* retrieve *a set of k most similar events* that occurred in the past along with the *context* of these events. For example, when a server fails, quickly retrieve a set of *k* earlier failures with an overlapping set of logged users and running processes.

The challenge in exploiting historical information comes from the sheer volume of historical data. Monitoring systems can easily produce Gigabytes and even Terabytes of data every day. Some systems use the history to build a model of the monitored environment [14] and, at runtime, compare the current state of the system to the model. We explore a different approach. Our goal is to query history for specific past events, similar to the current event. TelegraphCQ [7] is a continuous monitoring engine with support for integrated queries over both archived data and live data. TelegraphCQ returns all results matching a query over the archive (or a uniform sample). Instead, our goal is to efficiently return a set of *k* most relevant past events, where relevance is defined by the context of the current event.

The problem that we address raises many important issues related to using historical information in monitoring systems. In particular, we identify the following three challenges in this paper:

Context definition and similarity. When a new event occurs in a monitored system, a large fraction of relevant historical information corresponds 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. The first challenge is thus in efficiently comparing the state of a monitored system at different points in time. Of course, we want to compare only those parts of the state which are relevant to the current event (*e.g.*, the list of logged 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. In Moirae, we are exploring techniques for comparing event contexts based on techniques from information retrieval (IR).

Approximate k-NN queries. Because the historical log is large, the complete set of past events related to a current event can be large. To avoid overwhelming the user, the goal of Moirae 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. For example, Moirae could inform a user of previous server failures where the list of logged users and running processes were similar to those appearing in the current event.

Supporting these k-NN queries is challenging. In contrast to previous work [23], the similarity metric is not defined directly on individual objects, but rather on the set of tuples that compose the context of an event. Additionally, because the historical log is large, the straightforward solution of materializing all past events with their contexts and scanning the materialized view when an event occurs is unsuitable: it would impose a large space overhead (espe-

cially if events are frequent, event contexts are large and the workload is changing) and scanning and sorting all past events could take a long time. On the other hand, we argue that accurate results are not necessary. For many applications, rapid access to *k* results among the most similar *and most recent* events is more important than an exact set of *k* most similar results returned with low latency. Therefore, Moirae relies on a partitioned materialization, and hierarchical query execution to efficiently support *approximate* k-NN queries, which return results among the most recent similar ones. If necessary, Moirae can further improve results incrementally.

Performance in face of concurrency. The third and final goal of Moirae is to achieve efficient extraction of similar past events not only for a single new event, but also in the presence of concurrent events. Because continuous queries can produce events at different and possibly varying rates, Moirae must properly allocate resources among variable numbers of ongoing events. Moirae includes an adaptive scheduler to ensure the extraction of at least some historical information for each new event.

In this paper, we discuss why history-enhanced monitoring and the above issues are important and challenging. We present the principled design of Moirae; outline the techniques used in Moirae to address the above challenges; and argue the more general applicability of our approach. 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 [2] stream processing engine (SPE) for continuous monitoring and PostgreSQL [22] for the historical log. Moirae modifies and tightly integrates both engines.

2. MOTIVATION AND GOALS

Providing a historical background to new events in a timely fashion is useful in many types of monitoring applications. Consider the following scenarios from different application domains. In each scenario, a set of data sources produces continuous streams of information about a monitored environment. These data streams are processed by an SPE as they arrive and are archived on disk.

Computer-system monitoring scenario: In a computer-system monitoring application, an administrator receives alerts when a server behaves abnormally: *e.g.*, the server crashes, the server refuses new connections for a period longer than 2 minutes, or the 5-minute average network bandwidth used by the server exceeds a predefined threshold. When a failure occurs, the administrator wants to reuse knowledge about previous similar events to diagnose the new problem. The administrator defines the context of the events as the set of users logged on the server, the location where they are connected from, and the set of processes they are running. Every time an event occurs on one of the servers, the administrator wants to see the list of logged users and running processes. At the same time, the administrator wants to see previous events of the same type that had a similar list of users and processes. By intersecting these lists, the administrator narrows down the set of users causing the problem. The administrator may also better distinguish malicious attacks from honest errors. Seeing past events *at the same time as* the alert helps the administrator respond to new events in a timely fashion. If multiple servers experience problems at the same time, the administrator needs to see the historical backgrounds for all ongoing problems.

Car-traffic monitoring: A car-traffic monitoring application produces an alert if an accident occurs on a user's normal route home. With the alert, the application computes a set of possible alternate routes based on traffic conditions that followed similar events in the past. To compute the route, the application needs to see a set of past incidents that occurred in a similar location and under similar weather and traffic conditions, along with the traffic load that followed each incident. A few examples suffice to determine an alternate route, but the application needs to see them as soon as possible, in order to quickly re-route the user. In case multiple incidents occur around the same time, the application must keep-up with the changing traffic conditions to send the user on an appropriate route.

Sensor-based environment monitoring: An operator is monitoring a plant and receives alerts when a particular combination of sensor values (temperature, pressure, etc.) starts to drift. With each alert, the system produces a set of past alerts that occurred under similar circumstances, along with the list of actions taken by previous operators. The operator uses the historical information to adjust or at least confirm his own response to each alert.

The above scenarios do not cover all possible uses of a historical log, but they illustrate the general benefits of integrating streaming data with historical data. Each scenario also raises the three main challenges that we identified:

1. The need to specify a complex context for events and measure context similarity. In each scenario, the application is interested in past events similar to the current event. The notion of what constitutes similar events changes between scenarios.
2. The need to quickly see an approximate set of similar past events with their contexts. The user needs to see a small set of highly relevant events, rather than a long list of possibly similar events. Each event must also be accompanied by its full context. In each scenario, the historical information must be produced in a timely fashion to allow applications or users to respond quickly to each alert. Timeliness also ensures that the system keeps-up with the continuous stream of new events, producing the historical information for an event and moving on to the next event.
3. The need to handle concurrent events effectively. In all scenarios, multiple alerts can occur quickly one after the other and applications need to see timely historical information for each one of them. Different types of alerts can also occur at the same time, each one requiring its own historical background.

More importantly, extracting the required past information from the historical log requires that *the SPE processes* the archived datastreams because the desired past events are of the same form as the current events. Performing such processing after an event occurs would yield poor response time. Preprocessing the entire log for each new continuous query and materializing all resulting events would cause a tremendous overhead in space and processing time, especially if the workload is changing (continuous queries are added and removed), events are frequent, and event contexts are large. If the log is not preprocessed, each query would have to execute for a long time before accumulating any interesting history.

In the following section, we present, Moirae, a general purpose data management system that enables the above scenarios without the space and time overhead of preprocessing the entire log, and without the need for queries to execute for a long time in order to accumulate their own historical information.

3. HISTORY-ENHANCED MONITORING

We first describe how applications specify continuous monitoring queries, which we call *event-queries*, and how applications specify the context of an event by specifying what we call *context-queries*. We then present our technique for computing the similarity between two event contexts.

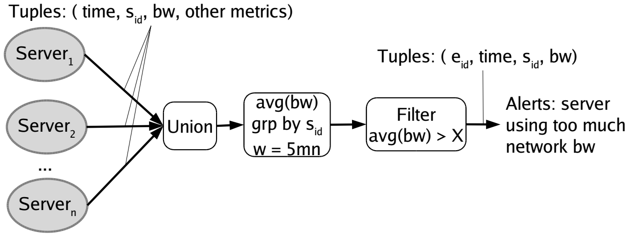

Figure 1: Example of continuous query (event-query).

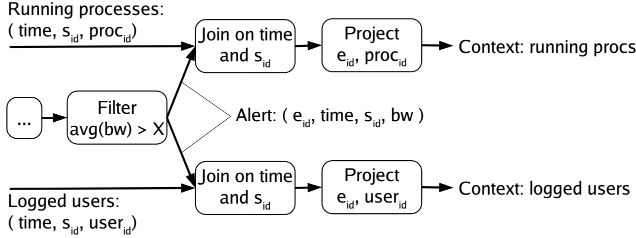

Figure 2: Example of query extracting the context of an event (context-query).

3.1 Specifying Events

For the continuous stream processing part of the system, Moirae uses Borealis [2]. In Borealis applications express continuous queries with a boxes-and-arrows data flow, where boxes represent operators and arrows stand for streams. Figure 1 shows an example of such an 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, `time` is a timestamp, s_{id} denotes the server identifier, and `bw` is the bandwidth utilization of the server. For simplicity, we assume that each output event is assigned a unique identifier, e_{id}.

3.2 Specifying the Context

To specify the context of an event, applications submit one or more queries that join the output of the event-query with other streams or static relations. We call these additional queries *context-queries*, as they produce the context of each event.

Using terminology from temporal databases [21], we expect the context of an event to typically include a subset of 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). We thus expect the join predicate to typically include references to the time of the event, and further restrict the set of tuples with additional predicates and projections. Context-queries can also include arbitrary SQL queries over static relations. For example, a context query can lookup 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.

The union of all tuples that satisfy these queries forms the context of the event. The context of an event can be the empty set. Figure 2 shows an example of context defined as the set of logged 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.

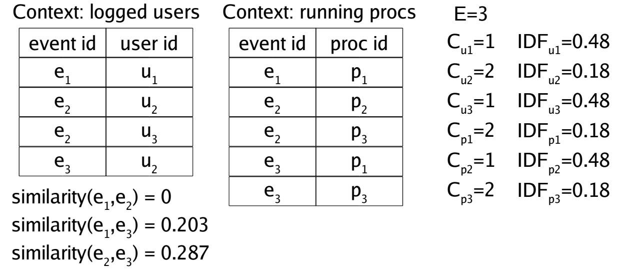

Figure 3: Example of similarity computation for three event contexts.

3.3 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. For continuous-domain attributes (such as a temperature reading), we discretize the values by rounding them to the nearest integer value. Two contexts are similar to each other if they contain a larger number of the same "terms" (attribute-values in our case).

If two contexts contain the same *rare* attribute-value, we weight that value more heavily. Intuitively, in our computer-system monitoring scenario, if a rare process is executing when a given type of failure occurs, then event instances with this same rare process should be considered as more relevant. We compute the weight of each attribute value as its TF-IDF product [4]. 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 ratio of event contexts containing that attribute. We use the following two standard formulas:

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

where E is the total number of events in the log, C_k is the number of event-contexts containing attribute-value k, and TF_{kc} is the frequency with which attribute-value k occurs in a context c. The similarity score between two contexts is then given by a metric called *cosine similarity*: *i.e.*, the cosine angle between the vectors of weights of the two contexts, where each attribute-value corresponds to one dimension. Figure 3 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 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. The IDF_k score for each element of the domain is computed incrementally. As new events are detected, the total number of events, E, and the total number of contexts, C_k containing attribute-value k are updated.

Using an IR engine [4] could improve performance for the above type of processing. We currently store our historical log, events, and contexts in a regular database because we envision performing a broader class of processing on the historical data in the future.

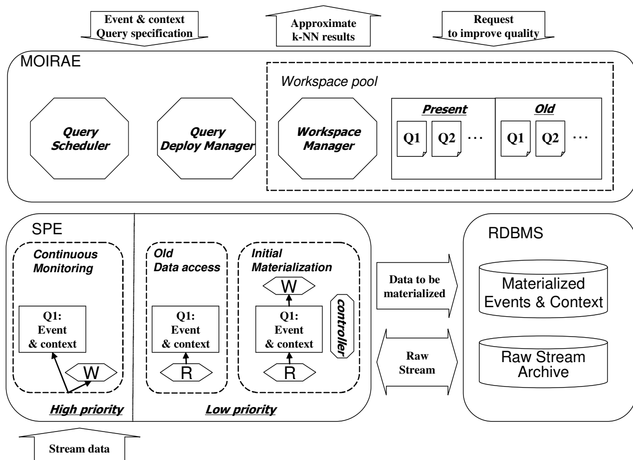

Figure 4: Moirae System architecture.

4. Moirae DESIGN

In this section, we discuss the design of Moirae and present its system architecture shown in Figure 4.

At a high level, Moirae is a middleware layer between the application and the underlying SPE and RDBMS. Moirae uses the SPE for continuous monitoring and the RDBMS for storing the raw historical data streams and the preprocessed events. However, Moirae modifies and integrates the two engines as we describe below.

An application or user communicates two pieces of information to Moirae: (1) the continuous query that defines the event of interest (*i.e.*, the event-query) and (2) the specification of what constitutes the context of the event (*i.e.*, the context-queries). Both pieces of information take the form of queries in the language of the underlying SPE. Moirae forwards these queries to the SPE, which executes them in its `Continuous Monitoring` component. At the same time, a `Write` operator asynchronously stores the incoming raw data streams directly in the `Raw Stream Archive`. Every time an event occurs, the SPE produces the event along with its context. Moirae forwards the information to the application, and queries the historical log for matching past events.

To extract the historical information, the naive approach would be for Moirae to run the event-query and context-queries on the whole historical log, materializing all past events with their context offline. At runtime, Moirae could then scan the materialized view, order past events on the similarity of their context, and output only the k most similar events. There are two problems with this naive approach. First, if events occur frequently, the materialized view will take a large amount of space. Also, because the workload changes with time, new continuous queries are introduced in the system, and old queries are removed, materializing all events for all continuous queries for the entire log would be expensive and likely not necessary. Second, scanning and sorting all past events will take a long time, especially if the number of events is large and the similarity function takes a long time to compute.

Instead, we propose an approach that somewhat sacrifices the accuracy of the results for significant gains in response time and overhead. Instead of examining the full log and outputting the most similar k events ever seen, we propose to output the most similar events *among the N most recent ones* (where N is a parameter defined by the administrator), based on the assumption that for most monitoring application recent events are more relevant than older events.

After producing an initial set of best k out of N results, if the user requests more results or if the system has spare cycles, Moirae can also incrementally examine older parts of the log and improve the quality of the responses returned so far.

This approximate k-NN approach is based on two key techniques: a *hierarchical partitioning* of the historical log and a *hierarchical query execution*. We describe these techniques in the following sections.

4.1 Hierarchical Partitioning

The key idea behind the hierarchical partitioning of the historical log is to partition the log into chunks and prioritize recent chunks over older chunks instead of considering the historical log as a monolithic sequence of events. In particular, we propose to distinguish between three types of chunks:

1. **Present chunk**: The most recent chunk of the historical log, typically incomplete.
2. **Recent chunks**: A small set of relatively recent chunks frequently searched when looking for relevant historical data.
3. **Old chunks**: Rarely accessed older parts of the log. These chunks are only processed in the presence of rare events that do not have many similar events in earlier parts of the log.

The size of chunks is defined by the administrator. For example, each chunk could correspond to a few hundred megabytes of data.

This chunked partitioning leads us to separate the architecture of Moirae into three layers. At the top layer, the present chunk always resides in memory and is searched first. At the second layer, the frequently accessed recent chunks reside on disk, but they are fully preprocessed and the materialized events and contexts are stored on disk. These materialized views are also indexed to speed-up event searches. Finally, at the bottom layer, older chunks reside on disk. They are not preprocessed and not indexed.

To illustrate the goals of this hierarchical partitioning, Figure 5, shows the results of a simple benchmark retrieving k tuples from a 1GB log partitioned into 5MB chunks. When k is small, indexing only half of all 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 decreases. Our goal is similar, although our queries are significantly more complex and the runtime for the raw stream data chunks is much slower. By materializing views and indexing only the most recent chunks, the cost of the views can remain constant as the log grows. The engine can efficiently retrieve a small set of recent events. Additional events can be retrieved if necessary, but at a greater cost.

4.2 Hierarchical Query Execution

We now present the detailed architecture for supporting the hierarchical partitioning described above and the steps that occur during query execution.

For the present chunk, Moirae receives the output of the `Continuous Monitoring` module of the SPE and accumulates the data in the `Workspace Pool`. Once the present chunk fills-up, Moirae stores it on disk in the `Materialized Events and Context` tables, and starts accumulating a new present chunk.

In Moirae, as in other SPEs, queries can be submitted at any time. When a query is submitted, Moirae needs to process the chunks at the second layer, materializing all relevant events and their contexts. Moirae does this by setting-up a lower-priority continuous query in the SPE, in the `Initial Materialization` module. This query reads chunks from the `Raw Stream Archive`, it executes the event-query and context-queries on it, and writes the

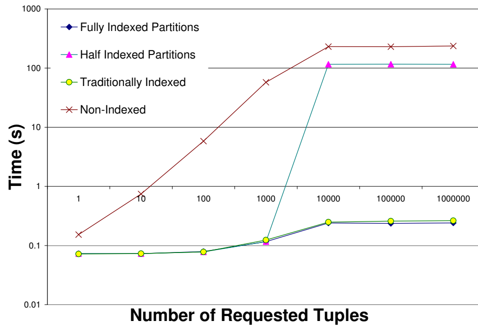

Figure 5: Illustration of performance of partitioned indexing. The "traditionally indexed" and "fully indexed partitions" curves overlap.

output to the `Materialized Events and Context` tables. A simple aggregate `Controller` operator, tracks the size of the materialized results, comparing the total size to the maximum allowed size (given as parameter). After each chunk, if the threshold is not yet reached, the `Controller` triggers the next, older chunk to be read and processed. The number of processed chunks is thus determined by the total amount of space allocated for this materialized view, the frequency of events, and the size of their contexts. We assume, however, that at least N events are materialized.

After materializing the older events and their contexts, Moirae can optionally index the materialized views for an even faster response time. We are currently investigating context-indexing techniques, but we do not present them here due to space constraints.

At runtime, when an event occurs, Moirae receives the event and its context from the SPE. To retrieve the relevant historical information, Moirae first queries the present chunk stored in the workspace pool. If more results are necessary, Moirae submits a series of queries to the RDBMS. Each query retrieves similar events from increasingly older chunks stored in the `Materialized Events and Context` tables. If all preprocessed chunks have been examined, but more results are necessary, Moirae needs to process older raw data streams to extract additional events. For this purpose, Moirae sets-up a copy of the event-query and context-queries as low-priority queries in the SPE (in the `Old Data Access` module). These queries read increasingly older chunks of raw stream information, process these streams, and output the results directly to the `Workspace Pool`. Moirae can then query these results in the same way it queries the present chunk. These results are discarded as soon as they are processed. We expect that few events will require this third level of processing.

With this approach, the returned results are not only approximate because Moirae examines only a subset of chunks, but also because events that fall on chunk boundaries may go undetected.

For all three types of chunks, *the same query plan* is executed. The only difference is in the location of the input data to the plan. For the present chunk and the old chunks, the data comes from the workspace pool. For recent chunks, the data comes from the `Materialized Events and Context` tables.

The query plan necessary for extracting similar events is relatively straightforward. The tables with the event contexts are already sorted by increasing event identifiers, as this order follows from the sequential processing of the streams. These tables can thus easily be joined. For each resulting *group of tuples*, a new score operator computes the similarity between each old event and the new event, using the technique from Section 3.3. The resulting events can then be sorted on their scores, and the top-k events for each chunk are produced.

Moirae must then perform some additional work. When older events are processed, Moirae filters out events less relevant than the ones already produced, by keeping track of the top k events produced for each alert. Moirae also keeps track of the total number of results produced for each alert. It uses this number to decide when more results are necessary.

Although designed for efficiently retrieving similar past events, the proposed hierarchical architecture and runtime execution are more generally applicable to various types of queries over a historical log, where fast queries over most recent fragments are more important than a complete query over the full log. We plan to explore these different types of queries in future work.

4.3 Query Scheduling

At runtime, 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. 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 is given quickly at least some historical information.

We call the set of queries that retrieve the historical information for an alert the *history-query* of the alert. Each query in the set extracts past events from one historical chunk. To ensure history-queries for all alerts quickly produce some results, we propose to use the stride scheduling algorithm [24]. This technique is a variant of lottery scheduling. Each query receives *tickets*. The number of tickets can be dynamically changed. History-queries with more tickets are scheduled more frequently. Every time a history-query set is scheduled, it executes only one of the queries in its set. After the query completes, the scheduler removes from the history-query a number of tickets proportional to the number of results produced (thus increasing the stride of the query). Because history-queries are scheduled based on their number of tickets, this approach ensures that history-queries with few results are scheduled more frequently.

In future work, it would be interesting to explore 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.

5. RELATED WORK

SPEs (*e.g.*, [1, 2, 8, 18]) typically focus on efficient processing of continuously arriving data. Some systems do allow joins between streams and stored relations [18], or SQL operations on stored relations [5], but they assume these relations are sufficiently small that the system can keep-up with input data rates. In Aurora [1] and Borealis [2], connection points can store a little history, but it can only be replayed to satisfy ad-hoc queries [1] or process tuple revisions [2]. The work most similar to the one we propose, is Chandrasekaran's work [7] on supporting *hybrid* queries over live and archived streams. The main difference with our proposal lies in the query model. Chandrasekaran's goal is to support three types of queries: queries that start in the past and continue in the present, queries that access a specific part of a historical log (*e.g.*, compare today's average with yesterday's average), or queries over the entire log. For the latter type of queries, Chandrasekaran assumes the entire log is properly indexed and the query over the log will typically execute quickly and return a small set of results. We focus on the latter types of queries, but we assume the log is too large to be queried in its entirety, and most queries return a large number of results. Hence, our goal is to produce an approximate set of k-NN results, where the most relevant results depend on the context of the current event. In terms of handling overload, Chan-

drasekaran's work is complementary to ours, and Moirae could leverage some of the proposed sampling techniques for storing and retrieving streams.

There has recently been a significant amount of work on adding support for top-K queries to databases [6, 9, 10, 15, 11]. In particular, the RanqSQL project [15] 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 [9] 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 related to the similarity search problem [12, 23]. However, existing techniques focus on comparing individual, multidimensional objects [23] or sequences [12] rather than sets of tuples. Techniques enabling keyword searches over relational databases [13, 16] are more closely related to our problem, but our goal is not to retrieve the closest tuples matching a set of keywords, it is to compare two groups of tuples.

Adaptive query processing [3, 20] enables a user to quickly see out-of-order results or approximate results that are incrementally improved with time. Our query scheduler builds on these ideas, but reorders results at a coarser granularity. Its goal is to ensure that all new events are quickly complemented with some historical information that is improved with time up-to a threshold.

There exists extensive work in the area of 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 [14]. The work is similar to Moirae in the sense of considering other contextual information such as weather, events, time, and holidays to predict traffic conditions. Data-mining-based solutions build domain specific models of an environment that they use to answer queries. In contrast, Moirae's goal is to enable users to see specific past events.

Finally, temporal databases [17, 19, 21] support sophisticated queries over persistently stored temporal data. In Moirae, 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 explored some of the benefits and challenges of enhancing a continuous monitoring system with support for context-aware historical queries. These queries automatically retrieve for each newly detected event past events with a similar context. The goal of these queries is to help users understand current events by rapidly showing them similar past events with their contexts.

We presented the design of Moirae, a new type of system that integrates continuous monitoring with the querying of a historical log to enable this new type of history-enhanced monitoring. The main insight behind the design of Moirae is that users will be more interested in receiving a few relevant results soon after each new event (especially if these events are recent), rather than a complete set of results or the best results with higher latency. We thus proposed a system architecture based on hierarchical log partitioning and hierarchical query execution, where the recent past is stored at a higher cost, but can be queried faster than older data.

We introduced preliminary ideas for addressing the specific challenges of computing the similarity between events (which we modeled as an instance of an information retrieval problem), retrieving an approximate set of *k* most similar past events, incrementally improving results, and scheduling the exploration of the historical log between multiple concurrent events.

We are currently building a prototype of Moirae, and plan to run experiments with our prototype on real-time streams and historical logs of car traffic data collected in the Seattle area since 1998[1], and computer-system monitoring data currently being collected in our department[2].

7. REFERENCES

[1] Abadi et. al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), Sept. 2003.

[2] Abadi et. al. The design of the Borealis stream processing engine. In *Proc. of the CIDR Conf.*, Jan. 2005.

[3] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *Proc. of the 2000 SIGMOD Conf.*, May 2000.

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

[5] Balakrishnan et. al. Retrospective on Aurora. *VLDB Journal*, 13(4), Dec. 2004.

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

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

[8] Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the CIDR Conf.*, Jan. 2003.

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

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

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

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

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

[14] 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.

[15] 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.

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

[17] 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.

[18] Motwani et. al. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the CIDR Conf.*, Jan. 2003.

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

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

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

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

[23] 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.

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

[1] We thank Daniel Dailey for providing us the car traffic data.

[2] We thank Jan Sanislo for helping us get access to this data.