Satisfying Complex Data Needs using Pull-Based Online Monitoring of Volatile Data Sources

Haggai Roitman, Avigdor Gal
Technion - Israel Institute of Technology
Haifa
32000 Israel
{avigal@ie, haggair@tx}.technion.ac.il

Louiqa Raschid
University of Maryland
College Park
MD 20742
louiqa@umiacs.umd.edu

Abstract

Emerging applications on the Web, including mashups in general and Web feeds in particular, require better management of volatile data in pull-based environments. In a pull based setting, data may be periodically removed from the server. Data may also become obsolete, no longer serving client needs. In both cases, we consider such data to be volatile. To model such constraints on data usability, and support complex user needs we define profiles to specify which data sources are to be monitored and when. Using an abstraction of execution intervals we model complex profiles that access simultaneously several servers to gain from the used data. Given some budgetary constraints (e.g., bandwidth), the paper analyzes algorithmic solutions to the problem of maximizing completeness. We discuss the complexity of the offline problem and propose an approximated solution. We next handle the online problem, introducing three heuristics for the online case. We use an extensive set of experiments that include real data and show that making use of the complexity level of profiles is dominant in many settings.

1 Introduction

In this work we discuss the optimization of complex data delivery of volatile data sources in a pull-based setting. There is a significant body of research work on volatile data delivery in a push-based setting, e.g., data streams ([3]). However, emerging applications on the Web, including mashups in general and Web feeds in particular, require the extension of the management of volatile data to pull-based environments. Mashups are one of the cornerstones of Web 2.0, thus pull-based volatile data management has a major impact on the development of Web 2.0 technologies and applications.

Volatile data is associated with an expiration time. For push-based data management, such expiration times were justified by the sheer load of the data, which prevents the storage of data for multiple-pass analysis. Therefore, the main focus of existing works is on smart filtering and load shedding. In a pull-based setting, expiration times may stem from the inability of a server to store all history (as is the case with sensors with flash memory or news feeds) and also from the limited usefulness of data to clients. In particular, clients may have different tolerance levels towards delayed monitoring of data. As a result, data that reach a client may no longer be useful (e.g., auction and stock information).

To model user data needs and personalize pull-based data delivery, we define a user profile to specify which data sources are to be monitored and when. Profiles may be complex in that they need to access simultaneously several servers to benefit from the monitored data (hence our reference to complex data delivery). We extend an abstraction of execution intervals (presented in [15]) to model profiles. An execution interval defines a period of time in which a server can be monitored to provide a client with useful information. A profile contains a set of execution intervals, possibly of different sources.

As a concrete example, consider a financial analyst that looks for arbitrage opportunities: “Arbitrage is the practice of taking advantage of a price differential between two or more markets: a combination of matching deals are struck that capitalize upon the imbalance, the profit being the difference between the market prices.”¹ A simple arbitrage monitoring example is illustrated in Figure 1, showing the change in the price of one stock in two different markets. To identify arbitrage opportunities, financial data should be collected from multiple markets. This data is volatile, changing frequently with changes to market prices. Also, the analyst’s data needs require that data from both markets

¹This definition is taken from Wikipedia, http://en.wikipedia.org/wiki/Arbitrage
Figure 1. Example arbitrage monitoring

will be available, with overlapping time reference. Therefore, a user profile for arbitrage contains pairs of execution intervals (marked as rectangles connected with a numbered oval in Figure 1), a single execution interval for each market. These execution intervals overlap to ensure the validity of arbitrage opportunities (otherwise, prices may refer to different times, invalidating the arbitrage opportunity). The analyst gains some benefit only if both servers were monitored in their respective execution intervals. Therefore, a complex profile in this case cannot be satisfied by the monitoring of a single market price. Rather, some (simple or complex) combination of monitoring tasks are needed. Similar profile examples can be found in accessing multiple auctions or multiple Web feeds using applications such as Google Reader\(^2\) (simple profiles) or Yahoo! Pipes\(^3\) (complex profiles).

Given multiple profiles and multiple data sources, we aim at capturing as many of the sets of execution intervals (e.g., price pairs in the arbitrage example) in a profile as possible, given some budgetary constraints (e.g., bandwidth). We analyze existing algorithmic solutions and offer new ones to the problem of maximizing completeness, measured in terms of captured execution-interval sets. We show that the offline problem, in which all execution intervals are known in advance, can be optimally solved, yet it may take a full enumeration of all possible schedules, which is of a high polynomial execution time. We show that an approximation exists and discuss its properties.

The online version of the problem assumes the algorithm has no knowledge about future arrival of execution intervals. We introduce a classification of heuristic classes and demonstrate each class with a simple heuristic. For one class, we present a well known heuristic (EDF), known to be optimal for the simple case of individual execution intervals. We use this case as a baseline for our evaluation. Using an extensive set of experiments to test the proposed heuristics we show that the heuristic MRSF, which makes use of the complexity level of the profile, is dominant in many settings. We also show that a non-random selection of resources for profiles improves the performance of all heuristics, yielding even better results than the approximated offline solution.

The specific contribution of our work are as follows:

- We provide a framework for evaluating complex profiles, serving an array of contemporary applications. This framework enables the extension of some applications to benefit from the presence of multiple clients.
- For the offline setting, we show that the problem is not NP-Complete, yet may require a full schedule enumeration, which is of an order of a high polynomial time complexity. We further present how an offline approximation can be achieved.
- For the online setting, we provide a three level classification of possible heuristics and discuss their properties.
- We provide a thorough empirical analysis, using a simulation environment we have developed for that purpose, which is based on real data trace feeds. We show that despite their simplicity, the proposed heuristics manage to maintain high completeness levels in many different settings.

The rest of the paper is organized as follows. In Section 2 we describe the related work. In Section 3 we present our model for complex profile monitoring and formally define the problem, followed by the monitoring solution in Section 4. We present experiments in Section 5 and in Section 6 we conclude and provide directions for future work.

2 Related Work

We now review works that involve satisfaction of either simple or complex data needs. While much focus has been given to efficient data processing methods that support complex data needs (expressed for example by queries or user profiles), less attention has been given to efficient data gathering methods in pull-based environments that involve volatile data. Volatile data is accessed by many contemporary applications, which include Web crawlers [16], Web monitors [14], and lately Web 2.0 Mashup applications (e.g., [17]), requiring access to multiple volatile Web sources such as Web feeds. We further classify systems that require such data according to the way it is gathered, either by pull or push.

\(^2\)http://www.google.com/reader
\(^3\)http://pipes.yahoo.com

2
With push based systems, data is pushed to the system and the research focus is mainly on aspects of efficient data processing. Such systems include publish-subscribe (pub-sub) systems (e.g., [4, 8]), stream processing systems (e.g., [3]), and complex event processing (CEP) systems (e.g., [1, 5]). Pub-sub systems such as the ONYX system [8] allow the registration of complex requirements at servers and focus mainly on the trade-off between the scalability of the pub-sub system in terms of data processing efficiency and the expressiveness of the queries that can be processed by the system, where data is assumed to be non volatile. Stream processing systems are also push-based in nature and focus mainly on smart filtering and load shedding techniques. Complex event processing (CEP) systems such as the Cayuga system [5] assume a stream of raw events are pushed into the system and focus mainly on efficient complex events and situations identification.

In this paper we assume a pull based solution. In a pull environment the data processing system is required to collect the data, e.g., via periodically monitoring of resources. Such systems include, among others, query processing in sensor networks (e.g., [6, 7]), Continuous Queries (CQ) and Web Monitoring (e.g., [12, 14]), Grid query processing (e.g., [18]), and Mashups of different Web sources (e.g., [17]). Current pull based solutions cannot handle complex data needs over multiple data sources. For example, current works in CQ and Web monitoring such as WIC [14] handle only simple single resource monitoring tasks that are further assumed to be independent one of each other. Other works in sensor networks further focus mainly on energy efficient data dissemination methods and thus data completeness requirements come in second, while now new opportunities with flash memory aided sensors require new techniques for data dissemination [7].

3 Model and Problem Definition

Servers and clients share data in our model through proxies. A server manages resources and can be queried by the proxy on behalf of the proxy clients. In this work we assume a hybrid approach, where the proxy probes servers for data via a pull protocol (e.g., by HTTP GET queries) and delivers data to clients using a push protocol. Our focus is on the scheduling of proxy pull tasks.

The interest of a client in updates to server’s resources are specified using profiles and stored at the proxy. Each client profile is associated with a set of resources, required by the client, and execution intervals [15]. An execution interval defines periods of time during which the client must be synchronized with the state of the resource in order to satisfy the client profile. The client can combine execution intervals to construct complex monitoring profiles over a set of resources. We refer the reader to [15] for details of a language to specify execution intervals and methods to generate execution intervals, possibly based on stochastic modeling (see [9]). The beginning of an execution interval is determined by either an update event to a resource or a temporal event (e.g., every ten minutes). A condition for terminating an execution interval can be set relatively to the stream of update events (e.g., update overwrite), or again as a temporal relative event (e.g., five minutes after its beginning). For example, a profile defined over Web feeds may require to collect items published on the feed before the server overwrites them. According to a recent intensive study on Web feeds in [10], 55% of Web feeds are updated hourly. [10] further shows that due to heavy workloads that may be imposed by clients on servers (especially on popular Web feed providers such as CNN), about 80% of the feeds maintained by servers have an average size smaller than 10KB. Thus, we can expect servers of such feeds to keep each item available for a limited life period.

We assume that clients have varying needs. Therefore, each profile may either share or not share some of its execution intervals with other profiles. Given a set of client profiles, the proxy monitors resources, captures updates to resources during their specified execution intervals, and delivers client notifications about resource states, captured during these intervals.

We provide next a formal definition of three building blocks of our model, namely client profiles, execution intervals, and schedules.

3.1 Profiles and t-intervals

To formally represent the notion of a complex profile, we extend the notion of execution intervals to that of t-intervals. A t-interval consists of execution intervals, possibly of different resources. Each execution interval in a t-interval should be monitored at least once for the t-interval to be considered satisfied (or “captured”). A complex profile is simply a set of t-intervals, modeling the client needs.

Formally, let \( R = \{r_1, r_2, \ldots, r_n\} \) be a set of \( n \) resources and let \( T = (T_1,T_2,\ldots,T_K) \) be an epoch with \( K \) chronons.\(^4\) We assume the proxy manages a set of client profiles \( P = \{p_1,p_2,\ldots,p_m\} \). A client profile \( p = \{\eta|\eta = \{I_1,I_2,\ldots,I_k\}\} \) is a collection of t-intervals [2]. A t-interval \( \eta \) contains several execution intervals, where each execution interval (or EI in short) \( I \) is associated with a resource \( r \in R \) and \( I \) contains a start and finish chronons \( I = [T_s,T_f];T_s,T_f \in T; T_s \leq T_f \). Execution intervals are the most primitive building blocks, serving as a formal tool to capture the volatile property of a resource. Profiles, t-intervals, and execution intervals construct a hierarchy, in which a profile is a parent

\(^4\)A chronon is an indivisible unit of time.
of its \( t \)-intervals, and a \( t \)-interval is a parent of its execution intervals. Two \( t \)-intervals within the same profile are siblings, and two execution intervals within the same \( t \)-interval are also siblings. We use the number of EIs in a \( t \)-interval to model profile complexity. Therefore, we denote by \( \text{rank}(p) \) the maximal number of execution intervals in any \( t \)-interval \( \eta \in p \) (\( \text{rank}(p) = \max_{\eta \in p} |\eta| \)), where \( |\eta| \) is the number of execution intervals in \( \eta \). The definition is easily extended to a set of profiles \( P \) as follows: \( \text{rank}(P) = \max_{p \in P} \{\text{rank}(p)\} \).

We now refer back to the graphical example in Figure 1 that provides a graphical illustration of \( t \)-intervals of a profile \( p \) with \( \text{rank}(p) = 2 \). \( p \) requires the monitoring of two different resources (each representing a different stock market). This profile requires to match an execution interval of one resource with an overlapping execution interval of the other resource.\(^5\) The profile requires to deliver the state of each resource on every update before the next update (overwrite policy). The dots represent updates to a resource, and execution intervals are given as rectangles.

Execution intervals of different profiles may overlap in time. Further, execution intervals of the same profile may also overlap. For example, in Figure 1 EIs of the two servers overlap. Overlapping intervals are interesting for two reasons. When intervals of different resources overlap (inter-resource overlap) they are all candidates for being simultaneously probed by the proxy. This can lead to congestion when the available probing budget is low. When the execution intervals associated with an identical resource overlap (intra-resource overlap), there is the potential to exploit this overlap in building a more efficient schedule. The special case of no intra-resource overlap (presented in Figure 1 for a single profile) is of theoretical interest. Therefore, in the rest of the paper we will separate our analysis to the case of no intra-resource overlap, where efficient bounds may be found for approximate solutions, and the general case that allows intra-resource overlap. In both cases, inter-resource overlap is obviously allowed.

### 3.2 Schedules

A data delivery schedule \( S = \{s_{i,j}\}_{i=1,\ldots,n; j=1,\ldots,K} \) (\( n \) resources and \( K \) chronons) assigns \( s_{i,j} = 1 \) if resource \( r_i \in \mathcal{R} \) should be monitored (probed) by the proxy at chronon \( T_j \in \mathcal{T} \), else \( s_{i,j} = 0 \). We denote by \( \mathcal{S} \) the set of all possible schedules.

To simplify our formal writing, we next define an indicator whose value depends on whether an execution interval is monitored. We next extend it to the capturing of a \( t \)-interval. Given a profile \( p \), a \( t \)-interval \( \eta \in p \), and an execution interval \( I \in \eta \) that refers to resource \( r_i \in \mathcal{R} \), an indicator \( \mathbb{I}(I, S) \) indicates whether the schedule successfully captures resource \( r_i \) state during the required execution interval \( I \); Formally:

\[
\mathbb{I}(I, S) = \begin{cases} 
1, & \exists T_j \in I : s_{i,j} = 1 \\
0, & \text{otherwise} 
\end{cases}
\]

The definition is extended to \( t \)-intervals as follows. Given a profile \( p \) and a \( t \)-interval \( \eta \in p \), we say that \( \eta \) is captured by schedule \( S \in \mathcal{S} \) if \( \mathbb{I}(\eta, S) = \prod_{I \in \eta} \mathbb{I}(I, S) = 1 \).

### 3.3 The monitoring problem

We assume that the proxy has a limited amount of resources that can be consumed for the monitoring task of client profiles. In this paper we consider a constraint similar to the one used in prior works of Web Monitoring [14] and Web Crawlers [16], where at each chronon \( T_j \in \mathcal{T} \) the proxy can monitor up to \( C_j \) resources. This constraint is represented by a budget vector \( \vec{C} = (C_1, C_2, \ldots, C_K) \).

Given a set of client profiles \( P = \{p_1, p_2, \ldots, p_m\} \), the proxy objective is to maximize the gained completeness, that is, to maximize the number of \( t \)-intervals from \( P \) that are captured given the budget \( \vec{C} \). A \( t \)-interval is successfully captured once all of its execution intervals are captured. Every \( t \)-interval \( \eta \in p \) that is successfully captured by the proxy schedule (indicated by \( \mathbb{I}(\eta, S) = 1 \)) increases the gained completeness.

Given a schedule \( S \in \mathcal{S} \), the gained completeness (denoted \( \text{GC} \) in short) from monitoring \( P \) during \( T \) according to \( S \) is calculated as follows (where \( |p| \) denotes the number of \( t \)-intervals in profile \( p \)):

\[
\text{GC}(P, T, S) = \frac{\sum_{p \in P} \sum_{\eta \in p} \mathbb{I}(\eta, S)}{\sum_{p \in P} |p|} 
\]

Formally, the monitoring problem is defined by the following constrained optimization problem.

**Problem 1 (Complex Monitoring)** Given a set of profiles \( P \) and an epoch \( T \):

- maximize \( \text{GC}(P, T, S) \)
- s.t. \( \sum_{i=1}^{n} s_{i,j} \leq C_j \), \( \forall j = 1, 2, \ldots, K \)

### 4 Monitoring Solutions

We now present solutions to the monitoring problem, presented in Section 3.3. We first discuss an offline solution (Section 4.1). Then, in Section 4.2, we propose an efficient online solution.
4.1 Offline Solution

In an offline setting, the proxy is provided with all t-intervals in \( P \) for \( K \) chronons in advance and has to determine the schedule \( S \) of probing resources in \( R \). We first discuss the complexity of a full enumeration of feasible schedules, yielding an optimal yet costly solution. Then, we describe an approximation algorithm for the offline case and discuss its properties.

4.1.1 Schedule Enumeration

A feasible solution to Problem 1 is a schedule that satisfy the problem constraints. Lemma 1 provides the cost of solving Problem 1 using full enumeration.\(^6\)

Lemma 1 Given \( n \) resources, \( K \) chronons, and a constraint \( C \) on the number of probes per chronon, all feasible schedules can be enumerated in \( O(n^KC_{\max}) \) time, where \( C_{\max} = \max_{j=1,2,\ldots,n}(C_j) \).

It is worth noting that \( C_{\max} \) and \( K \) are known yet arbitrary constants and therefore the problem is polynomial in the number of resources. This serves as little consolation whenever \( C_{\max} \) or \( K \) are large (e.g., \( K = 100 \)). We assume that \( C_{\max} \neq O(n) \), otherwise scheduling would be easy since there will be sufficient budget to probe most resources at each chronon. Nevertheless, \( C_{\max} \) may still be of substantial size. To date, we are unaware of any low-polynomial algorithm for solving Problem 1.

4.1.2 Offline Approximation

We now shortly discuss how to achieve the best offline approximation for Problem 1. For this purpose we need to present some additional notation. We denote by \( P^{[1]} \) a set of profiles for which any execution interval of any t-interval has a width of exactly one chronon. Proposition 2 establishes the relationship between a solution to problems with input \( P^{[1]} \) to problems with a general set of profiles of the same complexity.

Proposition 2 Let \( P \) be an arbitrary set of profiles with \( \text{rank}(P) = k \). Let \( A \) be an algorithm that provides \( \alpha(k) \)-approximation given an arbitrary \( P^{[1]} \) with \( \text{rank}(P^{[1]}) = k \), then \( A \) can provide \( \alpha(k+1) \)-approximation given \( P \).

Bar-Yehuda et al. have proposed in [2] an offline approximation to the problem of scheduling t-intervals, also termed split intervals in [2]. In their problem, t-intervals are composed of segments of arbitrary length\(^7\) and a segment represents an interval in which a resource is needed. Note that this setting is slightly different than ours since we require that a resource should be probed only in a single chronon of each execution interval. We argue that a simple (possibly exponential) transformation from their setting to ours, enables us to use their algorithm in approximating an offline solution to Problem 1. To the best of our knowledge, [2] provides the best approximation to the problem of scheduling split intervals in general.

Equipped with Proposition 2, we utilize the Local Ratio scheme\(^8\) [2], which has the best approximation ratio given an arbitrary \( P^{[1]} \) with \( \text{rank}(P^{[1]}) = k \), and provides \( 2k \)-approximation for \( C_{\max} = 1 \) and \( (2k + 1) \)-approximation for \( C_{\max} > 1 \). Thus, for an arbitrary \( P \) with the same rank, according to Proposition 2, applying the Local Ratio scheme of [2] we can achieve \( (2k+2) \)-approximation when \( C_{\max} = 1 \), and \( (2k+3) \)-approximation for \( C_{\max} > 1 \). Thus, for example, assuming that we have as an input profiles of \( P^{[1]} \) with \( \text{rank}(P^{[1]}) = 2 \) (each t-interval has at most 2 segments), in the case of \( C = 1 \), we can produce a feasible solution to Problem 1 that guarantees at least \( 1/(2 \cdot 2) = 25\% \) of the optimal gained completeness.

The Local Ratio scheme of [2] adds additional constraints to Problem 1 and transforms it into a problem of finding a maximal independent set of t-intervals in a split interval graph. Using this scheme we can provide the best offline approximation ratio to Problem 1, yet this solution does not scale well for real world problem instances (as we shall see in Section 5.4). Thus, we next seek an alternative scalable online policies, easy to implement, which can handle Problem 1 efficiently.

4.2 Online solution

4.2.1 General description

In the online setting, the proxy has to make decisions without knowing the stream of future t-intervals required in \( P \) in advance. Specifically, at every chronon \( T_j \), the proxy may receive a set of new t-intervals \( \eta(j) = \{\eta_1, \eta_2, \ldots, \eta_q\} \). The proxy then has to decide which resources in \( R \) to probe, while considering the set of all candidate t-intervals, including those submitted prior to \( T_j \), which have not been completely captured yet, and the new set of t-intervals \( \eta(j) \). We denote the set of all candidate t-intervals at chronon \( T_j \) as \( \text{cands}(\eta) \) and the union bag of all their execution intervals (termed candidate EIs) as \( \text{cands}(I) = \bigcup_{\eta_q \in \text{cands}(\eta)} \eta_q \). The bag notation \( (\cup) \) is used due to intra-resource overlaps.

To determine which candidate EIs in \( \text{cands}(I) \) to choose the proxy uses policies. At chronon \( T_j \), a policy \( \Phi \) considers the candidate EIs in \( \text{cands}(I) \), the chronon \( T_j \), and the budget \( C_j \), and returns up to \( C_j \) EIs to probe. Such poli-

\(^6\)We omit proofs from the paper due to space consideration. The interested reader is referred to [13].

\(^7\)It is worth noting that segments in [2] correspond to execution intervals, and the segments of each t-interval refer to a single resource.

\(^8\)We refrain from presenting the Local Ratio scheme in details and the interested reader is referred to [2, 13].
cies can be efficiently implemented and used by the proxy to efficiently select EIs for probing.

Each of the policies we propose can be executed in either a non-preemptive or preemptive manner. Non-preemptive policies do not allow new candidate t-intervals to be scheduled for monitoring at time $T$ if previously selected t-intervals need to be probed at $T$. Therefore, a non-preemptive policy $\Phi$ first selects EIs $I \in cands(I)$ that belong to those previously selected t-intervals; then, if there is any budget left, $\Phi$ selects more EIs to probe from the newly introduced t-intervals.

The implementation of these policies is rather simple and due to space considerations we refrain from presenting the details and the interested reader is referred to the supplementary online version [13], which provides also detailed complexity analysis of our own implementation.

4.2.2 Online policies

Online policies for solving Problem 1 can be classified according to the amount of information needed about the candidate t-intervals and their execution intervals. We next propose a three level classification, illustrating each class with a sample heuristic.

**Rank level**: A rank level policy bases its decision on profile complexity by considering the rank of the parent t-interval. As a representative of this level we suggest the Minimal Residual Stub First (MRSF) policy, $\Phi_{MRSF}$. This policy prefers execution intervals that belong to parent t-intervals with a minimal number of EIs left to be captured. The intuition behind this policy is that a t-interval with less EIs remaining to probe has a higher probability of success. Formally, given an execution interval $I$, $I \in \eta$ and $\eta \in \rho$, then the $\text{MRSF}$ value is calculated as follows:

$$\text{MRSF}(I) = \text{rank}(p) - \sum_{I' \in \eta} \mathbb{I}(I', S)$$

where $I'$ iterates over all EIs in $\eta$. The following proposition provides a bound for the performance of this policy.

**Proposition 4** Given $\mathcal{P}$ without intra-resource overlap and $\text{rank}(\mathcal{P}) = k$, the $\Phi_{MRSF}$ policy is k-competitive.

**Multi-EIs level**: A multi-EIs level policy utilizes all information about execution intervals of a parent t-interval (including sibling information). As a representative of this level we suggest the Multi Interval EDF (M-EDF) policy, $\Phi_{M-EDF}$, which prefers execution intervals that have the minimal M-EDF value, calculated as follows (for $I \in \eta$):

$$\text{M-EDF}(I, T) = \sum_{I' \in \eta} (\text{S-EDF}(I', T) \cdot [1 - \mathbb{I}(I', S)])$$

The M-EDF value combines the EDF values of execution interval $I$ and its siblings that were not captured up to chronon $T$, where for each execution interval $I'$ (again, iterating over all EIs in the t-interval) if the execution interval is not yet active (chronon $T < I'.T_s$), then the EDF value is calculated with $T = 0$. The intuition behind this policy is that a t-interval with less total remaining chronons have less chance to collide with other t-intervals, thus higher probability for more gain in completeness.

**Proposition 5** For problem instances with $\mathcal{P}[1]$ profiles the M-EDF policy is equivalent to the MRSF policy.

**Example 1** Figure 2 illustrates an example of a candidate t-interval that includes four required execution intervals. At chronon $T$, we show how it is evaluated for each of the policies. At the bottom we give the calculated values of each policy. S-EDF counts the number of remaining chronons.
5 Experiments

5.1 Datasets and Experiment Setup

We used two data sets of update events in our experiments. The first is a real-world trace of eBay auctions selling Intel®, IBM®, and Dell® laptop computers, where the trace events were obtained by extracting bid information from Web feeds published by eBay® during a time period of three months. To generate the $t$-intervals, we used the FPN(1) update model [14], which assumes a perfect knowledge of the real update trace. The second data set is a synthetic data set, generated using a Poisson($\lambda$) update model, where $\lambda$ allows to control the update intensity of each resource during the epoch $T$.

We generated synthetic profiles in a three-stage process utilizing two Zipf distributions. First, given that \( \text{rank}(P) = k \) we determined the rank (number of execution intervals in each $t$-interval) of a profile according to a Zipf($\beta$, $k$) distribution, where $\beta = 0$ implied a random selection at uniform $U[1,k]$, while positive $\beta$ values produced more profiles that contained $t$-intervals with smaller number of execution intervals. Such selection models intra-user preferences; that is, user preference regarding the complexity of a profile.

As a second step, given that the profile rank has been set to $k$, we have used a Zipf($\alpha$, $n$) distribution to select for each profile the set of resources to be required by the profile, where $\alpha = 0$ implies a random selection at uniform $U[1,n]$, while positive $\alpha$ values implied a preference towards “popular” resources. Such a selection models inter-user preferences and imitates the way popular resources are chosen by users such as in the case of Web Feeds with $\alpha = 1.37$ [10].

Finally, using the update event traces, we generated the $t$-intervals using an “AuctionWatch($k$)” profile template. This template profile requires to monitor the price of an item sold in parallel $k$ auctions and to notify the user once a new bid was posted in all of the auctions. To determine the length of each execution interval, we used two common data delivery restrictions, namely overwrite and window($W$). The Overwrite restriction requests every bid update on each auction to be delivered before the next update occurs and overwrites the last published bid. Such restriction models user preference of data completeness. The window($W$) restriction requests every bid update to be delivered within a window of $W$ chronons from the time the bid was posted. The window($W$) restriction models user tolerance to stale data.

We implemented a simulation-based environment to test the different solutions. Given a set of profiles and an update event trace as input, we generate a stream of $t$-intervals to be captured by the proxy. In the offline case the proxy was provided with the complete stream for $K = 1000$ chronons in advanced, while in the online setting, the proxy is provided at each chronon with a set of $t$-intervals that overlap the chronon. The online monitoring has been executed on the same problem instances as the offline approximation for a given $K = 1000$ chronons. We repeated each such execution (offline/online) 10 times and recorded the average performances with regard to completeness and we further recorded the runtime of each solution.

The experiments were run on a Lenovo® IBM® Thinkpad® T60 machine, with a 2.00GHz Intel® Centrino® Duo processor, and 2.00GB of RAM. The algorithms were implemented in Java, using JDK version 1.4.2. The JVM was initiated with a heap-memory size of 1.000GB.

Table 1 further summarizes the controlled parameters and contains the baseline parameter settings that where used in the experiments.

5.2 Real-World Trace

We first used the real-world trace together with AuctionWatch($3$) profiles, where each profile requires to monitor up to 3 auctions and report every time a new bid
was posted in all of the auctions. We compared the gained completeness of each online policy using various parameter settings with and without preemption. Figure 3 has the results of such comparison with 400 auction resources and a window of \( w = 20 \). We further used a budget of \( C = 2 \) probes per chronon. We observe that the two policies that rank according to the \( t \)-interval as a whole, \( MRSF(P) \) and \( M-EDF(P) \), perform better then \( S-EDF \). These results imply that additional knowledge of the profile complexity (or multi-interval view) can indeed help to discover more opportunities for completeness gain using preemption. Our experiences with the \( S-EDF \) policy showed that in general it performs better when being executed without preemption for \( C = 1 \), while for \( C > 1 \), the preemptive version is preferable. On the other hand, our experiences further showed that the two other policies perform almost always better when being executed with preemption. These results were consistent for most of the parameter settings that we used, with a difference of up to 20\% in gained completeness between the preemptive and non-preemptive versions of each online policy. As a convention from now on, we shall accompany a policy name with “(P)” to denote the preemptive version of the policy, and “(NP)” for non-preemptive.

### 5.3 Online Performance vs. Offline Approximation

We compared the performance of the offline approximation to the online policies for different parameter settings. We performed experiments where we set \( w = 0 \) that requires an immediate probing of each execution interval, and further set \( C = 1 \). Such parameter selection produces \( P^{[1]} \) profiles for which the offline approximation guarantees a 2\( k \)-approximation (assuming \( rank(P^{[1]}) = k \)) which is the best possible offline approximation (see Section 4.1.2). It is worth noting that according to Proposition 5, for this setting the \( MRSF(P) \) and \( M-EDF(P) \) policies perform the same and thus we do not present here the \( M-EDF(P) \) policy. In general, our experiments results show that both \( M-EDF(P) \) and \( MRSF(P) \) policies outperform both the offline approximation and the \( S-EDF \) policy (both preemptive or not). The \( S-EDF \) policy does not dominate the offline approximation.

As one representative example, consider Figure 4, in which \( rank(P) \) serves as the independent parameter. We refrain from presenting the preemptive version of the \( S-EDF \) policy since for \( w = 0 \), preemption has no impact on this policy. It is worth noting that for \( rank(P) = 1 \) the gained completeness presented in Figure 4.1.2 is optimal. We observe that complex profiles force the proxy to devote more probes for each \( t \)-interval and thus completeness suffers. We observe that the \( S-EDF(NP) \) policy is dominated by the offline approximation for rank values > 2, while they are both dominated by \( MRSF(P) \) policy which manages to increase performance over that of the offline approximation by 11\% – 23\%.

### 5.4 Runtime scalability

We next performed scalability analysis and measured the runtime of the offline approximation and the online policies aggregated over a run with \( K = 1000 \) chronons. The results are given in Figure 5. We ran the experiments with increasing number of profiles, generating as a result an increasing number of execution intervals. The number of profiles and the update intensity that were used are given on top of the curves, and on the upper left side of the graphs respectively. First we compared the offline approximation and online policies with small workloads (\( \lambda = 20 \) and 100-500 profiles; see Figure 5(1)). We can see that the offline approximation has a much worse runtime then the online policies. We further continued to investigate the scalability of the online policies and increased the workload using 2.5 times higher updates intensity and increased the number of profiles up to 2500 (see Figure 5(2), offline approximation is eliminated to allow easy differentiation in the performance of the online policies). We can see that there is still a linear trend in the policies runtime behavior, suggesting that for real problem instances the online policies are scalable and more robust compared to the offline approximation.

The scalability analysis suggests that offline performance is poor for large instances. Therefore, we explore further the behavior of the online policies as a scalable solution.

### 5.5 Workload analysis

We next studied the effect of different workload settings on the gained completeness of the online policies. For this
purpose we controlled two parameter settings, namely the average updates intensity per resource (given by $\lambda$), and number of profiles ($m$).

Figure 6 contains the results. In general, we observe that both the $\text{MRSF}(P)$ and $\text{M-EDF}(P)$ policies are much better than $\text{S-EDF}$ policy (either preemptive or not) for all workload parameter settings that were used. We also observe that the $\text{M-EDF}(P)$ policy is slightly lower than the $\text{MRSF}(P)$ policy, and the same for $\text{S-EDF}(P)$ compared to $\text{S-EDF}(NP)$ (recall that we use strict budget $C = 1$).

Figure 6. Workload analysis

Figure 6 (1) shows the effect of average resource update intensity on the policies performance. We observe that as the average update intensity increases, there are more updates to each resource, and thus, each profile requires to capture more $t$-intervals and as a result the gained completeness decreases.

Figure 6 (2) further shows the effect of number of profiles on the policies performance. We again observe that as the number of profiles increases, completeness decreases, due to the increase in the number of $t$-intervals that need to be captured by the proxy.

5.6 The impact of user preferences

We now report on the impact of user preferences on performance. For this analysis, we have used various settings of $\alpha$ (inter-user preferences), and $\beta$ (intra-user preference). See Section 5.1 for the definition of these parameters.

Figure 7(1) shows the impact of inter-user preferences. As $\alpha$ increases, there is less random selection of resources in each profile, with more execution intervals coming from popular resources. The online policies gain more completeness due to more opportunities to capture intra-resource overlapping execution intervals of popular resources. Furthermore, we observe that the $\text{S-EDF}(P)$ policy has gained less completeness than its non-preemptive counterpart, implying that non-preemption better utilizes the intra-resource overlaps.

Figure 7(2) shows the impact of intra-user preferences. As $\beta$ increases, users prefer less complex profiles. The impact, as observed in Figure 4 as well, is an increase in performance. We further observe that both $\text{MRSF}(P)$ and $\text{M-EDF}(P)$ policies still outperform the $\text{S-EDF}$ policy, with a slight variation between the first two. This may be attributed to the fact that there are still $t$-intervals that require to probe more than one execution interval, where $\text{S-EDF}$ was shown in Figure 4 to be dominated.

5.7 Effect of budgetary limitations

We now study the effect of budgetary limitations on the different policies. So far we have used a strict budgetary allocation of $C = 1$. We now show that impact of additional budget has on performance. The results are given in Figure 8. We observe that as the proxy budget increases, allowing it to probe more resources per chronon, a remarkable increase in performance is achieved. In particular,
MRSF(P) policy utilizes the budget much better than the S-EDF policy. We conclude that the aggregated view of MRSF(P) and M-EDF(P) policies helps to better utilize the budget. As a final note, we observe that S-EDF(P) improves linearly with budget increase while S-EDF(NP) shows only sub-linear increase, which makes S-EDF(P) a more suitable heuristic than S-EDF(NP) for real-world settings.

6 Conclusions and Future work

In this work we presented a framework for satisfaction of complex data needs in pull based environments that involve volatile data. We then presented offline solutions, and since those solutions fail to scale we suggested efficient online policies. Using intensive experiments we further analyzed the performance of these policies under different settings and showed that even under restrictive budget constraints they can perform well. We further showed that utilizing additional profile structures can assist to improve performance significantly.

As future extension of this work we shall consider more general profile satisfaction constraints given as client profile utilities. Such utilities can further help to construct better prioritized policies. In this paper we assumed that all execution intervals of any t-interval are required to be captured. We further intend to extend the notion of t-intervals to a more general construction which allow also alternatives (e.g., capture of a subset of execution intervals).

Acknowledgment

We would like to thank Reuven Bar-Yehuda and Michal Schmueli-Scheuer, for useful discussions.

References