Scalable Scheduling of Updates in Streaming Data Warehouses

Lukasz Golab, Theodore Johnson and Vladislav Shkapenyuk
AT&T Labs – Research, Florham Park, NJ, 07932, USA
{lgolab,johnsont,vshkap}@research.att.com

Abstract—We discuss update scheduling in streaming data warehouses, which combine the features of traditional data warehouses and data stream systems. In our setting, external sources push append-only data streams into the warehouse with a wide range of inter-arrival times. While traditional data warehouses are typically refreshed during downtimes, streaming warehouses are updated as new data arrive. We model the streaming warehouse update problem as a scheduling problem, where jobs correspond to processes that load new data into tables, and whose objective is to minimize data staleness over time (at time t, if a table has been updated with information up to some earlier time r, its staleness is t minus r). We then propose a scheduling framework that handles the complications encountered by a stream warehouse: view hierarchies and priorities, data consistency, inability to preempt updates, heterogeneity of update jobs caused by different inter-arrival times and data volumes among different sources, and transient overload. A novel feature of our framework is that scheduling decisions do not depend on properties of update jobs (such as deadlines), but rather on the effect of update jobs on data staleness. Finally, we present a suite of update scheduling algorithms and extensive simulation experiments to map out factors which affect their performance.

Index Terms—Data warehouse maintenance, On-line scheduling

1 Introduction

Traditional data warehouses are updated during downtimes [25] and store layers of complex materialized views over terabytes of historical data. On the other hand, Data Stream Management Systems (DSMS) support simple analyses on recently arrived data in real time. Streaming warehouses such as DataDepot [15] combine the features of these two systems by maintaining a unified view of current and historical data. This enables real-time decision support for business-critical applications that receive streams of append-only data from external sources. Applications include:

- On-line stock trading, where recent transactions generated by multiple stock exchanges are compared against historical trends in nearly real time to identify profit opportunities;
- Credit card or telephone fraud detection, where streams of point-of-sale transactions or call details are collected in nearly real time and compared with past customer behavior;
- Network data warehouses maintained by Internet Service Providers (ISPs), which collect various system logs and traffic summaries to monitor network performance and detect network attacks.

The goal of a streaming warehouse is to propagate new data across all the relevant tables and views as quickly as possible. Once new data are loaded, the applications and triggers defined on the warehouse can take immediate action. This allows businesses to make decisions in nearly real time, which may lead to increased profits, improved customer satisfaction, and prevention of serious problems that could develop if no action was taken.

Recent work on streaming warehouses has focused on speeding up the Extract-Transform-Load (ETL) process [28][32]. There has also been work on supporting various warehouse maintenance policies, such as immediate (update views whenever the base data change), deferred (update views only when queried), and periodic [10]. However, there has been little work on choosing, of all the tables that are now out-of-date due to the arrival of new data, which one should be updated next. This is exactly the problem we study in this paper.

Immediate view maintenance may appear to be a reasonable solution for a streaming warehouse (deferred maintenance increases query response times, especially if high volumes of data arrive between queries, while periodic maintenance delays updates that arrive in the middle of the update period). That is, whenever new data arrive, we immediately update the corresponding "base" table T. After T has been updated, we trigger the updates of all the materialized views sourced from T, followed by all the views defined over those views, and so on. The problem with this approach is that new data may arrive on multiple streams, but there is no mechanism for limiting the number of tables that can be updated simultaneously. Running too many parallel updates can degrade performance due to memory and CPU-cache thrashing (multiple memory-intensive ETL processes are likely to exhaust virtual memory), disk-arm thrashing, context switching, etc. This motivates the need for a scheduler that limits the number of concurrent updates and determines which job (i.e., table) to schedule (i.e., update) next.

1.1 Scheduling Challenges

Real-time scheduling is a well-studied topic with a lengthy literature [7]. However, our problem introduces

Digital Object Indentifier 10.1109/TKDE.2011.45 1041-4347/11/$26.00 © 2011 IEEE

unique challenges that must be simultaneously dealt with by a streaming warehouse.

Scheduling metric. Many metrics have been considered in the real-time scheduling literature. In a typical hard real-time system, jobs must be completed before their deadlines – a simple metric to understand and to prove results about. In a firm real-time system, jobs can miss their deadlines, and if they do, they are discarded. The performance metric in a firm real-time system is the fraction of jobs that meet their deadlines. However, a streaming warehouse must load all of the data that arrive; therefore no updates can be discarded. In a soft real-time system, late jobs are allowed to stay in the system, and the performance metric is lateness (or tardiness), which is the difference between the completion times of late jobs and their deadlines. However, we are not concerned about properties of the update jobs. Instead, we will define a scheduling metric in terms of *data staleness*, roughly defined as the difference between the current time and the timestamp of the most recent record in a table.

Data consistency. Similarly to previous work on data warehousing, we want to ensure that each view reflects a "consistent" state of its base data [10][35], even if different base tables are scheduled for updates at different times. We will describe how the append-only nature of data streams simplifies this task.

Hierarchies and priorities. A data warehouse stores multiple layers of materialized views, e.g., a fact table of fine-grained performance statistics, the performance statistics rolled up to a coarser granularity, the rolled-up table joined with a summary of error reports, and so on. Some views are more important than others and are assigned higher priorities. For example, in the context of network data, responding to error alerts is critical for maintaining a reliable network, while loading performance statistics is not. We also need to prioritize tables that serve as sources to a large number of materialized views. If such a table is updated, not only does it reduce its own staleness, but it also leads to updates (i.e., reduction of staleness) of other tables.

Heterogeneity and non-preemptibility. Different streams may have widely different inter-arrival times and data volumes. For example, a streaming feed may produce data every minute, while a dump from an OLTP database may arrive once per day. This kind of heterogeneity makes real-time scheduling difficult. Suppose that we have three recurring update jobs, the first two being short jobs that arrive every ten time units and take two time units each to complete, and the third being a long job that arrives every 100 time units and takes 20 time units to complete. The expected system utilization of these jobs is only 60 percent – in an interval of length 100, we spend 20 time units on the long job, plus 2*10=20 time units on ten instances of each of the short jobs. However, serially executing these jobs starves the short ones (i.e., multiple jobs will accumulate) whenever the long one is being executed. This is because the execution time of the long job is longer than the inter-arrival time (i.e., the period) of the short ones. Here, this means that tables or views whose update jobs are short may have high staleness. However, short jobs correspond to tables that are updated often, which are generally important.

One way to deal with a heterogeneous workload is to allow preemptions. However, data warehouse updates are difficult to preempt for several reasons. For one, they use significant non-CPU resources such as memory, disk I/Os, file locks and so on. Also, updates may involve complex ETL processes, parts of which may be implemented outside the database.

Another solution is to schedule a bounded number of update jobs in parallel. There are two variants of parallel scheduling. In *partitioned scheduling*, we cluster similar jobs together (e.g., with respect to their expected running times) and assign dedicated resources (e.g., CPUs and/or disks) to each cluster [27]. In *global scheduling*, multiple jobs can run at the same time, but they use the same set of resources. Clustering jobs according to their lengths can protect short jobs from being blocked by long ones, but it is generally less efficient than global scheduling since one partition may have a queue of pending jobs while another partition is idle [4][12]. Furthermore, adding parallelism to scheduling problems generally makes the problems more difficult; tractable scheduling problems become intractable, real-time guarantees loosen, and so on [11]. The real-time community has developed the notion of *Pfair* scheduling for real-time scheduling on multi-processors [5]. However, Pfair scheduling requires preemptible jobs.

Transient overload. Streaming warehouses are inherently subject to overload in the same way that DSMSs are. For example, in a network data warehouse, a network problem will generally lead to a significantly increased volume of data (system logs, alerts, etc.) flowing into the warehouse. At the same time, the volume of queries will increase as network managers attempt to understand and deal with the event. A common way to deal with transient overload in a real-time system is to temporarily discard jobs. Discarding some data is acceptable in a DSMS that evaluates simple queries in a single pass and does not store any data persistently [31]. However, all the data must be loaded into a warehouse, so we cannot drop updates, just defer their execution.

During overload, a reasonable scheduler defers the execution of update jobs corresponding to low-priority tables in favour of high-priority jobs. When the overload subsides and low-priority tables can finally be scheduled, they may have accumulated a large amount of work (i.e., multiple "chunks" of new data may have arrived). As a result, these low-priority jobs become long-running and may now starve incoming high-priority updates.

1.2 Contributions and Organization

We identify and propose a solution to the update scheduling problem in real-time streaming warehouses. In Section 2, we introduce our system model, formalize the notion of staleness, and describe our solution for maintaining data consistency under arbitrary update schedules. Section 3 presents our scheduling framework. In the proposed framework, a *basic algorithm* orders jobs by some reasonable means, e.g., by the marginal benefit of executing them, roughly defined as the reduction in

staleness per unit of processing time. Basic algorithms are then augmented to handle the complications encountered by a streaming warehouse:

- We use the notion of "inheriting priority" to give more weight to tables that serve as sources of a large number of materialized views.
- In order to scale to large and diverse job sets, we propose a mechanism that combines the efficiency of global scheduling with the guarantees of partitioned scheduling. As in partitioned scheduling, we group the update jobs by their processing times. Each group defines a partition and runs its own basic algorithm. Normally, at most one job from each partition can run at any given time. However, under certain circumstances, we allow some pending jobs to be scheduled on a different partition if their "home" partition is busy. Thus, we can reserve processing resources for short jobs while achieving varying degrees of global scheduling.
- Rather than loading all the available data into low-priority tables during a single job, we load one new "chunk" of data (or at most *c* chunks) per job. We call this technique *update chopping* and we use it to prevent deferred low-priority jobs from blocking high-priority jobs. Update chopping adds a degree of preemptibility by scheduling data loads in multiples of their natural groupings.

Section 4 presents experimental results. We find that 1) an effective basic algorithm must incorporate priorities; however, any reasonable such algorithm appears to work nearly equally well, 2) inheriting priority is necessary when dealing with view hierarchies, 3) algorithms which reserve processing resources for short jobs significantly improve performance when the jobs are heterogeneous, as long as global scheduling is handled properly, and 4) update chopping improves performance after periods of transient overload.

Finally, Section 5 discusses related work and Section 6 concludes the paper.

A preliminary version of this paper has appeared as a short paper [16], which briefly introduced the update scheduling model and the basic algorithms that we will show in Section 3.1. In this paper, we present the full scheduling framework, including novel partitioning algorithms in Section 3.2. These new algorithms allow our framework to be used for large and diverse job sets, as required by an industry-strength streaming warehouse.

2 System Model

2.1 Streaming Warehouse Architecture

Table 1 lists the symbols used in this paper. Figure 1 illustrates a streaming data warehouse. Each data stream *i* is generated by an external source, with a batch of new data, consisting of one or more records, being pushed to the warehouse with period P_i. If the period of a stream is unknown or unpredictable, we let the user choose a period with which the warehouse should check for new data. Examples of streams collected by an Internet Service Provider include router performance statistics such as CPU usage; system logs; routing table updates; link layer alerts, etc. An important property of the data streams in our motivating applications is that they are append-only, i.e., existing records are never modified or deleted. For example, a stream of average router CPU utilization measurement may consist of records with fields *(timestamp, router_name, CPU_utilization)*, and a new data file with updated CPU measurement for each router may arrive at the warehouse every five minutes.

Table 1. Symbols used in this paper

Symbol	Meaning
P_s	Period of stream s
J_i	Update job for table i
$E_i(n)$	Execution time of J_i on data produced in a time interval of length n
R_i	Release time of J_i
p_i	Priority of table i
$F_i(\tau)$	Freshness of table i at time τ
$S_i(\tau)$	Staleness of table i at time τ
ΔF_i	Freshness delta of table i
α_i	Time to initialize the ETL process for table i
β_i	Data arrival rate to table i

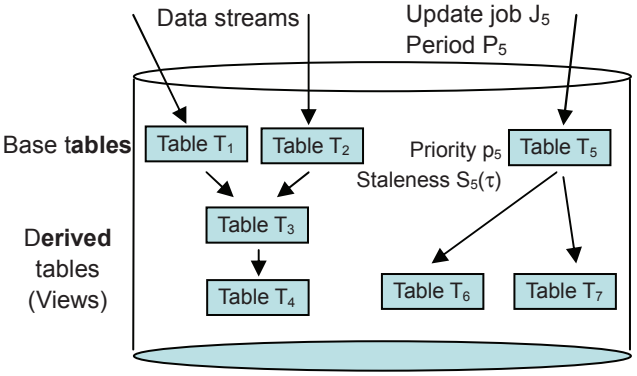

Fig. 1. A streaming data warehouse.

A streaming data warehouse maintains two types of tables: base and derived. Each table may be stored partially or wholly on disk. A base table is loaded directly from a data stream. A derived table is a materialized view defined over one or more (base or derived) tables. Each base or derived table T_j has a user-defined priority p_j and a time-dependent staleness function $S_j(\tau)$ that will be defined shortly. Relationships among source and derived tables form a (directed and acyclic) dependency graph. For each table T_j, we define a set of its *ancestor* tables as those which directly or indirectly serve as its sources, and a set of its *dependent* tables as those which are directly or indirectly sourced from T_j. For example, T_1, T_2 and T_3 are ancestors of T_4, and T_3 and T_4 are dependents of T_1.

When new data arrive on stream *i*, an update job J_i is *released* (i.e., inserted into the scheduler queue), whose

purpose is to execute the ETL tasks, load the new data into the corresponding base table T_i, and update any indices. When this update job is completed, update jobs are released for all tables directly sourced from T_i in order to propagate the new data that have been loaded into T_i. When those jobs are completed, update jobs for the remaining dependent tables are released in the breadth-first order specified by the dependency graph. Each update job is modeled as an atomic, non-preemptible task. The purpose of an update scheduler is to decide which of the released update jobs to execute next; as we mentioned earlier, the need for resource control prevents us from always executing update jobs as soon as they are released.

We assume that the warehouse completes all the update jobs, i.e., data cannot be dropped. Furthermore, if multiple updates to a given table are pending, they must be completed in chronological order. That is, we cannot load a batch of new data into a table if there is an older batch of data for this table that has not yet been loaded.

In practice, warehouse tables are horizontally partitioned by time so that only a small number of recent partitions are affected by updates [13][15]. Historical partitions are usually stored on disk; recent partitions may be stored on disk or in memory [32]. Updates of base tables simply append new records to recent partitions. Affected partitions of derived tables may be recomputed from scratch or updated incrementally. Note that in addition to being updated regularly as new data arrive, derived tables often store large amounts of historical data (on the order of months or years). We can reduce the number of partitions per derived table by using small partitions for recent data and large partitions for historical data [15].

In some cases, we may want to update several tables together. For instance, if a base table is a direct source of a set $\boldsymbol{T}$ of many derived tables, it may be more efficient to perform a single scan of this base table (more specifically, a single scan of the partitions that have changed) to update all the tables in $\boldsymbol{T}$. To do so in our model, we define a single update job for all the tables in $\boldsymbol{T}$.

2.2 Warehouse Consistency

Following the previous work on data warehousing, we want derived tables to reflect the state of their sources as of some point in time [10][35]. Suppose that D is derived from T_1 and T_2, which were last updated at times 10:00 and 10:05, respectively. If T_1 and T_2 incur arbitrary insertions, modifications and deletions, it may not be possible to update D such that it is consistent with T_1 and T_2 as of some point in time, say, 10:00 (we would have to roll back the state of T_2 to time 10:00, which requires multi-versioning). However, tables in a streaming warehouse are not "snapshots" of the current state of the data, but rather they collect all the (append-only) data that have arrived over time (or at least within a large window of time). Since the data are append-only, each record has exactly one "version". For now, suppose that data arrive in timestamp order. We can extract the state of T_2 as of time 10:00 by selecting records with timestamps up to and including 10:00. Using these records, we can update D such that it is consistent with T_1 and T_2 as of time 10:00.

Formally, let $F_i(\tau)$ be the *freshness* of table T_i at time τ, defined as the maximum timestamp of any record in table T_i at that time. Similarly, we define the freshness of a data stream as the maximum timestamp of any of its records that have arrived by time τ. We define:

- the *leading edge* of a set of tables $\boldsymbol{T}$ at time τ as the maximum timestamp of any record in any of the tables, i.e., $\max_{i \text{ in } T}(F_i(\tau))$;
- the *trailing edge* of the set $\boldsymbol{T}$ at time τ as $\min_{i \text{ in } T}(F_i(\tau))$, i.e., the freshness of the least-fresh table in the set.

Let D be a derived table directly sourced from all the tables in a set $\boldsymbol{T}$. Our consistency rule is as follows: when updating D, we only use data from each T_i in $\boldsymbol{T}$ with timestamps up to the trailing edge. After the update is completed, D is consistent with respect to the state of its sources as of the trailing edge.

Now, suppose that data may arrive out-of-order. There is usually a limit on the degree of disorder, e.g., network performance data arrive at most one hour late (and they are discarded if they arrive more than an hour late) [3]. Some sources insert *punctuations* [33] into the data stream to inform the warehouse that, e.g., no more records with timestamps older than t will arrive in the future. We define the *safe trailing edge* of a set of tables $\boldsymbol{T}$ as $\min_{i \text{ in } T}(F'_i(\tau))$, where F'_i is the "safe" freshness of table i, i.e., the maximum time value such that no record with a smaller timestamp can arrive in the future. Returning to the above example, suppose that T_1 has been updated at time 10:00 and we know that no more records with timestamps smaller than 9:58 will arrive. Further, T_2 has been updated at time 10:05, and no more records with timestamps smaller than 10:01 will arrive. Updating D up to the safe trailing edge of 9:58, rather than the trailing edge of 10:00, ensures consistency even with out-of-order arrivals.

Finally, note that if updates are scheduled independently, then related views may not be *mutually* consistent in the sense that they may reflect the state of their source tables at different times [35]. For example, derived tables D_1 and D_2 computed from the same set of source tables need not be updated at the same time. This may be a problem for users who run historical analyses on multiple tables spanning exactly the same time interval. One solution is to maintain two *logical* views of a table: a real-time component that includes the latest data and is updated as often as possible, and a stable historical component that, e.g., does not include the current day's data. The focus of this paper is on the real-time component.

2.3 Data Staleness

We define $S_i(\tau)$, the staleness of table T_i at time τ, to be the difference between τ and the freshness of T_i:

$$S_i(\tau) = \tau - F_i(\tau).$$

In Figure 2, we illustrate the staleness as a function of time for a base table T_i. Suppose that the first batch of new data arrives at time 4. Assume that this batch contains records with timestamps up to time 3 (e.g., perhaps the batch was sent at time 3 and took one time unit to arrive). Staleness accrues linearly until the completion of the first update job at time 5. At that time, T_i has all the

data up to time 3, and therefore its staleness drops to 2. Next, suppose that the second batch of data arrives at time 7, but the system is too busy to load it. Then, the third batch of data arrives at time 9. During this time, staleness accrues linearly. Now suppose that both batches are loaded together at time 11. At that time, all the data up to time 9 have been loaded, so the staleness drops to 2. Notably, if the second update did not arrive, and instead the third update arrived with all the records generated between time 3 and 9, the staleness function would look exactly the same.

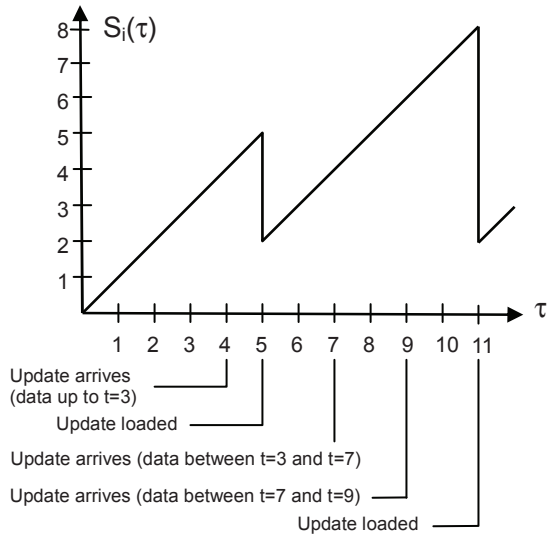

Fig. 2. Staleness of a base table T_i.

According to the above definition, staleness begins to accrue immediately after an update, even if there are no pending updates. We can normalize the staleness resulting from a particular scheduling algorithm with respect to the staleness achieved by an ideal algorithm that receives each update without delay and starts to execute it immediately upon arrival (i.e., there is no contention for resources and no limit on the number of concurrent jobs).

Recall that p_i is the priority assigned to table T_i. Our goal in this paper is to minimize S, the total priority-weighted staleness over time:

$$S = \sum_i p_i \int S_i(\tau) d\tau$$

Note that our objective of minimizing priority-weighted data staleness does not depend on any properties of the update jobs themselves, such as deadlines.

2.4 Scheduling Model

Let J_i be the update job corresponding to table T_i. For base tables, the period of J_i is equal to the period of its source stream. We estimate the period of a J_i corresponding to a derived table as the maximum period of any of T_i's ancestors (recall the definition of ancestor tables from Section 2.1). For base and derived tables, we define the freshness delta of T_i, call it ΔF_i, as the increase in freshness after J_i is completed. For instance, when the second batch of new data arrives in Figure 2 (at time 7), the table contains data up to time 3. Since the update contains data up to time 7, the freshness delta is 4.

Recall from Section 2.1 that tables in a streaming warehouse are usually partitioned by time, meaning that, for many classes of views, updates only need to access one or a few of the most recent partitions. Because of the temporal nature of streaming data, even complex derived tables such as joins require access into a small number of partitions of their source tables during updates. For example, data stream joins typically have "band join" semantics (joined records must have timestamps within a window length of each other). Rather than scanning a whole table to update the join result, it suffices to scan one or a few of the most recent partitions. Thus, we assume that the execution time of an update job is a function of the amount of new data to be loaded.

(This assumption is not crucial. For example, if a derived table needs to be recomputed in its entirety during every update, then we can model its update execution time in terms of the size of the whole table.)

Let n be the time interval of the data to be loaded. We define the execution time of update job J_i as:

$$E_i(n) = \alpha_i + \beta_i * n,$$

where α_i corresponds to the time to initialize the ETL process, acquire locks, etc., and β_i represents the data arrival rate. Clearly, the α_i and β_i may vary across tables. We can estimate the values for α_i and β_i from recently observed execution times; the value of n for a particular update job may be approximated by its freshness delta.

A new update job is released whenever a batch of new data arrives, meaning that multiple update jobs may be pending for the same table if the warehouse was busy updating other tables. For now, we assume that all such instances of pending update jobs (to the same table) are merged into a single update job that loads all the available data into that table (up to the trailing edge). This strategy is more efficient than executing each such update job separately because we pay the fixed cost α_i only once.

(However, as we mentioned, update chopping may be necessary after periods of overload, where we do not load all the available data to prevent update jobs from running for a very long time and blocking other jobs. We will discuss update chopping in more detail in Section 3.4).

3 Scheduling Algorithms

This section presents our scheduling framework. The idea is to partition the update jobs by their expected processing times, and to partition the available computing resources into *tracks*. A track logically represents a fraction of the computing resources required by our complex jobs, including CPU, memory and disk I/Os. When an update job is released, it is placed in the queue corresponding to its assigned partition (track), where scheduling decisions are made by a local scheduler running a basic algorithm (however, the algorithm that we will present in Section 3.2.3 generalizes this assumption). We assume that each job is executed on exactly one track, so

that tracks become a mechanism for limiting concurrency and for separating long jobs from short jobs (with the number of tracks being the limit on the number of concurrent jobs). For simplicity, we assume that the same type of basic scheduling algorithm is used for each track.

At this point, one may ask why we do not precisely measure resource utilization and adjust the level of parallelism on-the-fly. The answer is that it is difficult to determine performance bottlenecks in a complex server, and performance may deteriorate even if resources are far from fully utilized. The difficulty of cleanly correlating resource use with performance leads us to schedule in terms of abstract tracks instead of carefully calibrated CPU and disk usage.

Below, we first discuss basic algorithms, followed by job partitioning strategies, and techniques for dealing with view hierarchies and transient overload.

3.1 Basic Algorithms

The basic scheduling algorithms prioritize jobs to be executed on individual tracks, and will serve as building blocks of our multi-track solutions that we will present in Section 3.2. For example, the Earliest-Deadline-First (EDF) algorithm orders released jobs by proximity to their deadlines. EDF is known to be an optimal hard real-time scheduling algorithm for a single track (w.r.t. maximizing the number of jobs that meet their deadlines), if the jobs are preemptible [7]. Since our jobs are prioritized, using EDF directly does not result in the best performance. Instead we use one of the following basic algorithms.

Prioritized EDF (EDF-P) orders jobs by their priorities, breaking ties by deadlines. Our model does not directly have deadlines, but they may be estimated as follows. For each job J_i, we define its release time r_i as the last time T_i's freshness delta changed from zero to non-zero (i.e., the last arrival of new data in case of base tables, or, for derived tables, the last movement of the trailing edge point of its source tables). Then, we estimate the deadline of J_i to be $r_i + P_i$ (recall that the period of a derived table is the maximum of the periods of its descendants).

Max Benefit Recall that the goal of the scheduler is to minimize the weighted staleness. In this context, the *benefit* of executing a job J_i may be defined as $p_i \Delta F_i$, i.e., its priority-weighted freshness delta (decrease in staleness). Similarly, the *marginal benefit* of executing J_i is its benefit per unit of execution time: $p_i \Delta F_i / E_i(\Delta F_i)$. A natural online greedy heuristic is to order the jobs by the marginal benefit of executing them. We will refer to this heuristic as *Max Benefit*. Since marginal benefit does not depend on the period, we can use Max Benefit for periodic and aperiodic update jobs.

For example, suppose that jobs J_1 and J_2, corresponding to Table 1 and Table 2, respectively, are released at time τ, with $p_1 = p_2 = 1$, $E_1 = 3$, $E_2 = 2$, $\Delta F_1 = 10$, and $\Delta F_2 = 5$. Max Benefit schedules J_1 at time τ because its delta freshness, and therefore its marginal benefit, is higher. Figure 3 plots the weighted staleness of these two tables between time τ and $\tau + 5$, assuming that J_1 runs first. Table 1 begins with a weighted staleness (and delta freshness) of ten at time τ. Three time units later, J_1 is done and staleness drops by ten (from 13 down to 3). The weighted staleness of Table 2 is five at time τ and 8 at time $\tau + 3$ when J_2 begins. At time $\tau + 5$, J_2 is completed and the staleness of Table 2 drops from ten to five. Between times τ and $\tau + 5$, the area under Table 1's staleness curve works out to 42.5 and the area under Table 2's staleness curve is 37.5, for a total weighted staleness of 80. We leave it as an exercise for the reader to verify that if J_2 were to run first, the total staleness in this time interval would be 85.

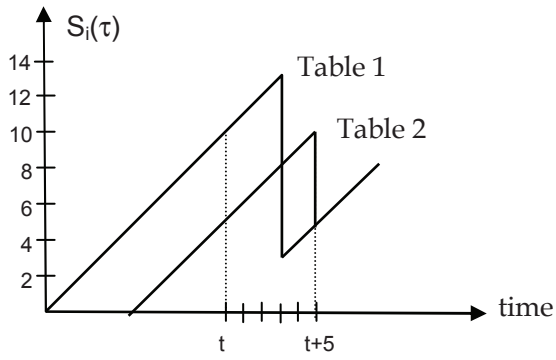

Fig. 3. Staleness of two tables scheduled according to Max Benefit

Since our jobs are assumed to be (approximately) periodic, one may argue that Max Benefit ignores useful information about the release times of future jobs. Recall the above example and suppose that J_3 is expected to be released at time $\tau + 1$, with $p_3 = 10$, $E_3 = 3$, and $\Delta F_3 = P_3 = 50$. It is better to schedule J_2 at time τ and then schedule J_3 when J_2 finishes at time $\tau + 2$, rather than scheduling J_1 at time τ and making J_3 wait two time units. To see this, note that J_3 accrues weighted staleness at a rate of $10*50=500$ per unit time, while J_1 accrues weighted staleness at a rate of 10 per unit time. Hence, making J_1 wait for several time units is better than delaying J_3 by one time unit. Motivated by this observation, we have tested the following extension to the Max Benefit algorithm.

Max Benefit with Lookahead chooses the next job to execute as follows.

1) J_i = *released job with the highest marginal benefit*
2) *For each* J_k *whose expected release time* r_k *is within* E_i *of the current time and whose marginal benefit is higher than that of* J_i
   a) *For each set S of released jobs* J_m *such that* $r_k <= \Sigma_m(E_m) < E_j$
      i) $B[S] = (\Sigma_m(p_m \Delta F_m) + p_k \Delta F_k) / (\Sigma_m(E_m) + E_k)$
3) *If* $\max_S B[S] >$ *marginal benefit of* J_i
   a) *Schedule the job with highest marginal benefit from set* $S^* = \text{argmax}_S B[S]$
4) *Else*
   a) *Schedule* J_i

In the above example, J_1 has the highest marginal benefit, but a job with a higher marginal benefit (J_3) will be released before J_1 is completed. The Lookahead algorithm needs to find alternate sequences of jobs whose total running time is between *1* and *2*, and compute their *B[S]* values; intuitively, *B[S]* represents the marginal ben-

efit of running all the tasks in S followed by J_3. There is one such sequence: {J_2, J_3}, with $B[S] = 505/5 = 101$, which is higher than the marginal benefit of J_1 of 10/3. Thus, it schedules J_2 instead of J_1.

The Lookahead algorithm employs a number of heuristics to prune the number of alternate job sets. First, it only considers sequences that never leave the system idle (line 2a: $r_k <= \Sigma_m(E_m)$), since it is potentially dangerous to avoid job invocation based on unreliable information about future job releases. Second, it only considers future jobs scheduled to arrive before the current job with highest marginal benefit is expected to complete. In our experiments, the scheduling overhead of the Lookahead algorithm as compared to standard Max Benefit was negligible. However, the performance gain of the Lookahead algorithm was very minor as it almost always chose the same job to execute as Max Benefit.

3.2 Job Partitioning

If a job set is heterogeneous with respect to the periods and execution times (long execution times vs. short periods), scheduler performance is likely to benefit if some fraction of the processing resources are guaranteed to short jobs (corresponding to tables that are updated often, which generally have higher priority). The traditional method for ensuring resource allocation is to partition the job set and to schedule each partition separately [7] (and to repartition the set whenever new tables or sources are added or existing ones removed, or whenever the parameters of existing jobs change significantly). However, recent results indicate that global scheduling (i.e., using a single track to schedule one or more jobs at a time) provides better performance, especially in a soft real-time setting, where job lateness needs to be minimized [5][12]. In this section, we investigate two methods for ensuring resources for short jobs while still providing a degree of global scheduling: *EDF-Partitioned* and *Proportional*.

3.2.1 EDF-Partitioned Strategy

The EDF-partitioned algorithm assigns jobs to tracks in a way that ensures that each track has a feasible non-preemptive EDF schedule. A feasible schedule means that if the local scheduler were to use the EDF algorithm to decide which job to schedule next, all jobs would meet their deadlines. In our setting, we assume that the deadline of an update job is its release time plus its period, i.e., for each table, we want to load every batch of new data before the next batch arrives.

To discuss this partitioning algorithm, we need to introduce some additional terminology:

- $u_i = E_i(P_i)/P_i$: The utilization of J_i, assuming that each job is completed before the next one arrives; and therefore the amount of new data to load is proportional to the length of the period.
- $U_r = \sum_i u_i$ The utilization of track r (summed over all jobs assigned to r).
- $Emax_r = max(E_i(P_i) \mid J_i \text{ in track } r)$, i.e., the processing time of the longest job in track r.
- $Pmin_r = min(P_i \mid J_i \text{ in track } r)$, i.e., the smallest period of all the jobs assigned to track r.

A recent result gives sufficient conditions for EDF scheduling of a set of non-preemptive jobs on multiple processors [3]. We use the simplified single-track conditions to find the EDF schedulability condition for track r:

$U_r \le 1 - Emax_r/Pmin_r$.

Finding an optimal allocation of jobs to tracks is NP-hard [27], so we use a modification of the standard greedy heuristic: sort the jobs in order of increasing periods, and then allocate them to tracks, creating a new track whenever the schedulability condition would be violated. If update job J_i is allocated to track r, then r is said to be its home track. If there are more processing tracks available than required to be allocated to the update jobs, the leftover tracks are referred to as free tracks.

Note that the EDF-partitioned strategy is compatible with any local algorithm for scheduling individual tracks. Of course, the feasibility guarantee (no missed deadlines) applies only if we were to use EDF as the local algorithm.

3.2.2 Leveraging Idle Resources via Track Promotion

We have EDF-partitioned the job set to "protect" short jobs from being blocked by long ones. The next issue we need to solve is how to avoid wasting resources when some tracks are idle and others have many pending jobs. The trick is to realize that long jobs are not significantly affected by blocking due to short jobs. Therefore we can "promote" a short job to an idle track that contains long jobs. The $Emax_r/Pmin_r$ term in the schedulability condition represents utilization loss due to non-preemptive blocking. Therefore, we can determine a job promotability condition as follows. Set:

- $Emax_r(J_i) = max(Emax_r, E_i(P_i))$
- $Pmin_r(J_i) = min(Pmin_r, P_i)$

Then the update job J_i can be promoted to track r if

$U_r \le 1 - Emax_r(J_i)/Pmin_r(J_i)$.

A track is available if no job is executing in it (or has been allocated for execution); else the track is unavailable. The final EDF-Partitioned scheduling algorithm is then the following.

1) *Sort the released jobs by the local algorithm*
2) *For each job J_i in sorted order*
   a) *If J_i's home track is available, schedule J_i on its home track*
   b) *Else, if there is an available free track, schedule J_i on the free track*
   c) *Else, scan through the tracks r such that J_i can be promoted to track r*
      i) *If track r is free and there is no released job remaining in the sorted list for home track r,*
         (1) *Schedule J_i on track r*
   d) *Else, delay the execution of J_i*

This algorithm is the non-aggressive version, as J_i is promoted to a track r only if r would not be otherwise be used. If we trust the local algorithm to properly order the jobs, we can use an aggressive version of the algorithm, in which step 2.c.i) is changed to "if track r is free". Finally, we remark that the computational overhead of the EDF-

partitioned algorithm is negligible: in the worst case, it performs the promotability check for each track, and we do not expect the number of tracks to be large in practice.

3.2.3 Proportional Partitioning Strategy

The EDF-partitioned algorithm has some weaknesses, which will be experimentally illustrated in Section 4. For one, a collection of jobs with identical periods (and perhaps identical execution times) might be partitioned among several tracks. The track promotion condition among these jobs and tracks is the same as the condition which limits the initial track packing – and therefore no track promotion will be done. We can patch the EDF-partitioned algorithm by using multi-track schedulability conditions, but instead we move directly to a more flexible algorithm.

The first step in the new algorithm, which we call Proportional, is to identify clusters of similar jobs. While there are many ways to do this, we use the following algorithm, which takes k as a parameter:

1) *Order the jobs by increasing execution time ($E_i(P_i)$)*
2) *Create an initial cluster C_0*
3) *For each job J_i, in order*
   a) *If $E_i(P_i)$ is less than k times the minimum period in the current cluster*
      i) *Add J_i to the current cluster*
   b) *Else, create a new cluster, make it the current cluster, and add J_i to it*

In general, choosing a small value of k may create many small clusters of jobs whose execution times and periods are similar; using a larger k yields fewer clusters that may be more heterogeneous. In other words, using a small k makes our algorithm behave more like a partitioned scheduler, while a larger value of k causes global-scheduling-like behaviour. Furthermore, in practice, a streaming warehouse workload often exhibits a clear separation between update periods. For example, a set of tables may be updated once every five minutes, another set once an hour, and another set once a day. In this particular example, moving from one set of tables to the next roughly increases the update periods by an order of magnitude, so a value of *k=10* may be used in the above algorithm to generate suitable clusters.

Next, we compute the fraction of resources to allocate to each cluster. Let c be the number of clusters.

1) *For each cluster C_j*
   a) *Compute $UC_j = \sum E_i(P_i)/P_i$, ranging over tasks in C_j.*
2) *Compute $U_{tot} = \sum C_j$*
3) *Compute an adjustment factor $M \geq 1.0$ such that $M*U_{tot}$ is an integer*
4) *Set track_lo[c] = 0, track_hi[c] = $M*UC_c$*
5) *For j=c-1 to 1 do*
   a) *Set track_lo[j] = track_hi[j+1]*
   b) *Set track_hi[j] = track_hi[j+1] + $M*UC_j$*

As with the EDF-partitioned algorithm, if any tracks are left over, they are free tracks. The point of the adjustment factor M is to ensure that the clusters are proportionally allocated to tracks, with no partial track left over. The range of tracks assigned to a cluster is analogous to the home tracks in the EDF-partitioned algorithm. However, a track might be partially allocated to two or more clusters, in which case the scheduling algorithm will need to share the track among the clusters.

The choice of M can range from $\lceil Utot \rceil / Utot$ (in which case we may leave many tracks free) to N/U_{tot}, where N is the number of available tracks (in which case all the available tracks will be allocated to the clusters). Thus, in contrast to EDF partitioning, which must use a certain number of tracks in order to meet the schedulability conditions, Proportional partitioning can allocate job clusters to any number of tracks. This is useful if we know the optimal number of tracks (i.e., the optimal level of parallelism) for the given warehouse. Of course, this optimal number is architecture-dependent and likely requires a great deal of empirical testing.

The Proportional strategy uses the following scheduling algorithm:

1) *Sort the released jobs using the local algorithm*
2) *For each job J_i in sorted order,*
   a) *Let j be the cluster of J_i*
   b) *If a track between track_lo[j]+1 .. track_hi[j] is available, schedule J_i on that track*
   c) *Else, if track track_lo[j] is available, schedule J_i on that track*
   d) *Else, if a free track is available, schedule J_i on that track*
   e) *Else, if there is an available track r numbered between 0 and track_lo[j]-1 such that there is no released job remaining in the sorted list that would be scheduled on r, schedule J_i on r*
   f) *Else, delay the execution of J_i*

As with the EDF-partitioned algorithm, step 2e) is the non-aggressive version. For the aggressive version, we change the condition in this step to "if there is an available track r numbered between 0 and *track_lo[j]-1*". Note that the overhead of the Proportional algorithm is small – in the worst case, it verifies whether each track is free.

3.2.4 Example

We now give a simple example to highlight the differences between EDF partitioning and Proportional partitioning. Suppose that we have four jobs to allocate to two tracks. The four jobs and their periods/execution times/utilizations are as follows: J_1 (4/1/0.25), J_2 (4/1/0.25), J_3 (8/2/0.25) and J_4 (10/1.25/0.125).

EDF partitioning begins by sorting the jobs in order of increasing periods and allocating them to tracks. The first track becomes the home track for J_1 and J_2 with a track utilization of 0.5 (we cannot add J_3 to the first track because the track utilization would become 0.75, which is greater than $1 - Emax_r/Pmin_r = 1 - 2/4 = 0.5$). J_3 and J_4 are allocated to the second track with a track utilization of 0.375. It is straightforward to work out that only J_4 satisfies the promotability check, meaning that the other three jobs can only be executed on their home tracks.

In contrast, the Proportional algorithm first sorts the jobs by increasing execution time. Suppose that we set k=2 in the clustering subroutine. This creates cluster C_0

with J_1 and J_2, and cluster C_1 with J_3 and J_4. Adding up the utilizations of all four jobs, we get $U_{tot} = 0.875$. Setting $M = N/U_{tot}$, C_0 receives tracks 1 and 2 as its home tracks and C_1 receives track 2 as its home track.

3.3 View Hierarchies

Materialized view hierarchies can make the proper prioritization of jobs difficult. For example if a high-priority view is sourced from a low priority view, then it cannot be updated until the source view is – which might take a long time since the source view has low priority. Therefore, source views need to inherit the priority of their dependent views. Let Ip_i be the inherited priority of table T_i. We explore three ways of inheriting priority:

- **Sum**: Ip_i is the sum of the priorities of its dependent views (including itself).
- **Max**: Ip_i is the maximum of the priorities of its dependent views (including itself).
- **Max-plus**: Ip_i is K times the maximum of the priorities of its dependent views (including itself), for some K>1.0. A large value of K increases the priority of base tables relative to derived tables, especially those derived tables which have a long chain of ancestors.

3.4 Dealing with Transient Overload

During transient overload, low-priority jobs are deferred in favor of high priority jobs. When the period of transient overload is over, the low-priority jobs will be scheduled for execution. Since they have been delayed for a long time, they will have accumulated a large freshness delta and therefore a large execution time – and therefore might block the execution of high-priority jobs. A solution to this problem is to "chop up" the execution of the jobs that have accumulated a long freshness delta to a maximum of c times their period, for some $c \geq 1.0$. This technique introduces a degree of pre-emptibility into long jobs, reducing the chances of priority inversion (low-priority jobs blocking high-priority jobs) [7].

A reasonable rule-of-thumb for choosing c is to use a small number greater than one. Large values of c result in little chopping, while setting $c = 1$ forces the scheduler to act as though there is no overload and every individual chunk of data needs to be loaded separately.

4 Experiments

4.1 Setting

The complex nature of the problem setting prevents us from examining performance implications all-at-once; instead we develop a series of experiments which identify the effects of different scenarios on different algorithms.

We wrote a simulator framework to test the performance of the algorithms. The simulator framework generates periodic data arrivals, monitors track usage, and generates a call to the scheduler on every data arrival or job completion. Each data point in our graphs is the result of processing one million of these events. An advantage of using a simulation rather than a prototype of a streaming data warehouse is the ability to perform a very large number of tests in reasonable time and under precisely controlled conditions.

Given an update job instance J_i with a non-zero freshness delta to be executed on a track, the simulator examines the job parameters to determine the execution time of the job instance, using the formula from Section 2.4, namely $E_i(n) = \alpha_i + \beta_i * n$. Since sources can provide variable amounts of data over time, we pick the execution time uniformly at random from the interval $[b*E_i(n), (1+b)*E_i(n)]$. Our simulator also introduces transient slowdowns. The system alternates between normal periods and slowdown periods. When in a slowdown period, each execution time is multiplied by a slowdown constant S (so the execution time used is $S*E_i(n)$).

We ran two types of experiments. In the first type, the simulation always executes in a slowdown period, and we adjust S to vary the total utilization U_{tot}. In the second type, the simulation alternates between normal and slow periods, and we vary α_i and β_i to vary U_{tot}. As we vary job execution times, the table staleness will vary. In order to obtain comparable numbers, we report the relative lateness, which is the average weighted staleness reported for an experiment divided by the average weighted staleness reported by an experiment with the same parameters, but with no contention for resources (i.e., every job is executed when released).

The slowdown factor S can have a significant effect on the lateness incurred by the scheduling algorithms. In Figure 4, we show the effect of S on the relative lateness. We adjusted to length of the slowdown period relative to the normal period to ensure an average execution time which is double that of the normal execution time. In this experiment, all jobs are identical, and we only display the results for local scheduling on a single track (other scenarios lead to identical trends). We pick a value of $S = 3$ for the remainder of the experiments; other values of S yield similar results. Other factors, such as the values of α_i and β_i, and the number of tracks used in the experiment, have a far less significant effect on relative lateness. In particular, we found that the number of tracks does not significantly affect the qualitative or comparative results, as long as the utilization remains constant. Thus, rather than reporting similar results for many different values of these parameters, we fix their values as shown in Table 2.

Table 2. Experimental parameters

Parameter	Value
α_i	1
β_i	0.1
P_i	100
B	0.5
Number of tracks	4 (unless otherwise noted)
Number of jobs	30

4.2 Basic Algorithms

Once priorities are accounted for, the choice of the basic algorithm has surprisingly little effect. In our experiments, Max-Benefit, Max-Benefit with Lookahead, and

EDF-P almost always make the same choices and the differences in relative lateness are too small to measure. One reason for this indistinguishability is the processing time model, in which there is a start-up cost to execute a jobs. Later jobs (i.e., those whose deadlines are nearer) have larger freshness deltas, and therefore they have better marginal benefits – and therefore Max-Benefit and EDF-P make the same decisions.

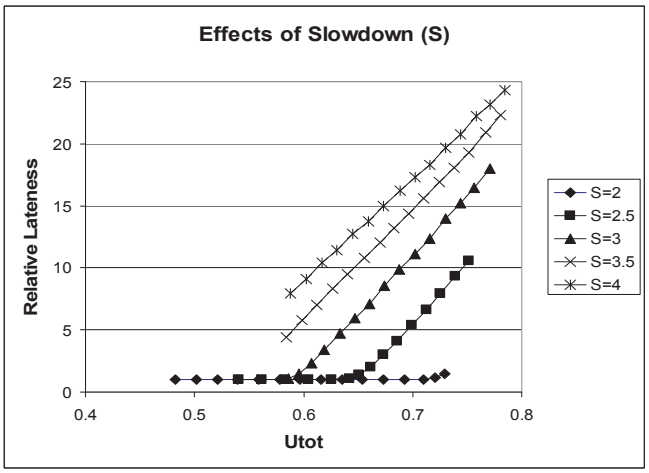

Fig. 4. Effects of slowdown on relative lateness.

Next, we highlight the difference between EDF-P and Max-Benefit. In this experiment, there are two classes of jobs, one with $\beta_i = 0.1$ and another with $\beta_i = 0.01$. That is, the former class consists of tables that can be updated nearly ten times as fast as those in the latter class. This experiment, shown in Figure 5, does show a difference but in an ambiguous way. When the relative lateness first starts to increase, EDF-P is better, but Max-Benefit is better at higher values of U_{tot}. We conclude that Max-Benefit seems to be the safer algorithm, and use it instead of EDF-P (or Max-Benefit with Lookahead) for the rest of our experiments. We note that for the remainder of this section, we did run experiments with EDF-P and obtained results identical to those using Max-Benefit.

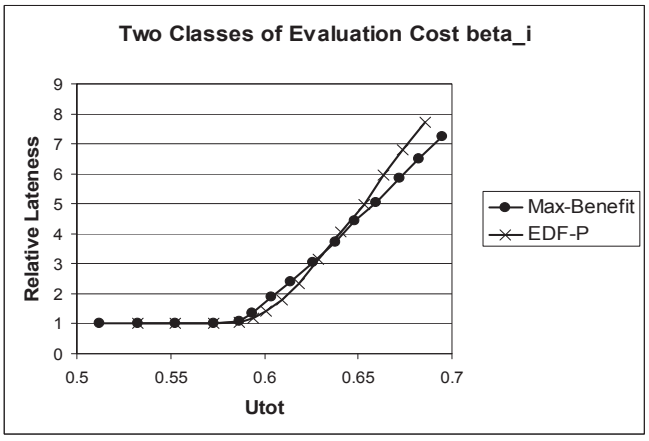

Fig. 5. Max-Benefit vs. EDF-P.

4.3 No Slowdowns

This experiment serves as a "sanity check" to show that there is no need for partitioning if the jobs are uniform, and to show that the Proportional algorithm does not incur any computation overhead. We found that, in the absence of widely varying periods, all algorithms generally have the same very good performance if there are no transient slowdowns. Figure 6 shows relative lateness for uniform jobs with identical parameters. The algorithms tested are: the local Max-Benefit algorithm running on a single track, a Random local scheduling algorithm (which always chooses the next job to execute randomly among the jobs in the released set), the EDF-Partitioned algorithm, and the Proportional algorithm. We note that in this experiment, we used five tracks instead of four, as the EDF-Partitioned algorithm could not fit the jobs into four tracks and therefore could not run (recall the discussion from Section 3.2.3, where we explain that EDF partitioning requires a specific number of tracks in order to meet the schedulability conditions, while the Proportional algorithm can work with virtually any number of available tracks). The Partitioned algorithm performs the worst (even worse than Random!) because it wastes processing resources (no track promotion is possible). Note that the performance of the Proportional algorithm is on par with that of the single-track Max-Benefit algorithm, i.e., the overhead of the Proportional algorithm is negligible.

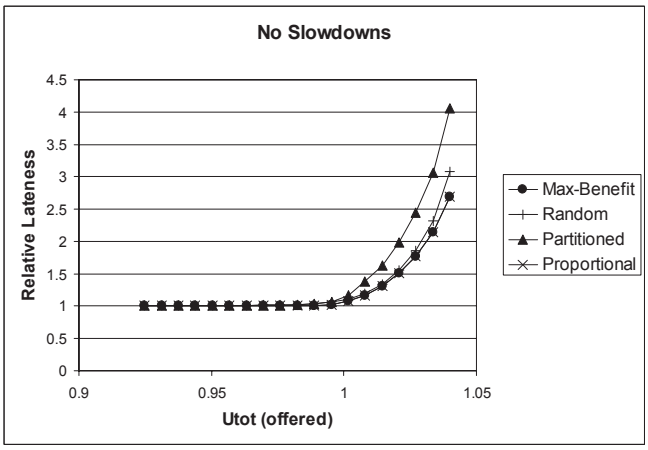

Fig. 6. No slowdowns, uniform jobs, 5 tracks.

The algorithms start to incur lateness only at a nearly 100% resource utilization. In this chart, we show an offered U_{tot} larger than 100% without a loss of stability. The reason for this behaviour is the nature of our cost model, with a start-up cost plus a cost proportional to the freshness delta. As update jobs get delayed, they become relatively cheaper to process; hence, more than the offered 100% workload can be performed.

4.4 Effect of Priorities

Intuitively, a prioritized scheduler performs better than a non-prioritized scheduler when jobs have varying priorities. To measure the benefit, we ran the experiment shown in Figure 7. We used two classes of jobs which are identical except that the first class has a priority of one

and the second has a priority of 10. The prioritized basic algorithm (in this case, Max-Benefit) incurs a far lower lateness than the non-prioritized basic algorithms, EDF and Random.

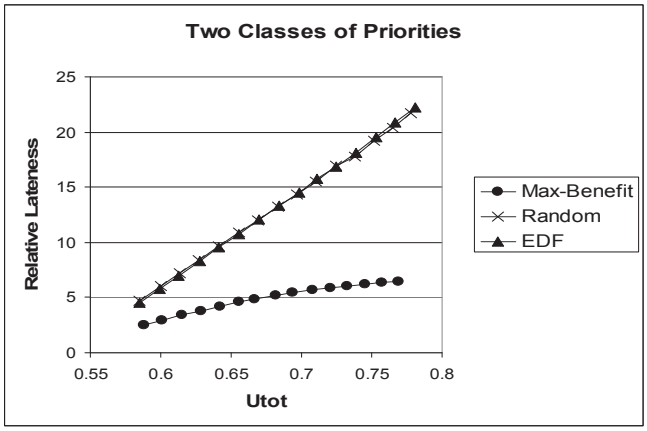

Fig. 7. Effects of prioritized jobs.

4.5 View Hierarchies

What is the effect of job hierarchies? One problem with this class of experiments is the wide range of view dependency graphs that can arise in practice. We ran experiments with a variety of graphs and received results similar to those shown in Figure 8. In this particular experiment, all hierarchies are a single chain with a depth of three. Base tables have priorities of 0.001, the next level (i.e., all the derived tables sourced from base tables) has priority of one, and the next level has a priority of either 1, 10, or 100 (with identical numbers of each). We ran the Max Benefit algorithm and tested what happens if no inheritance occurs, versus if the inheritance mechanism is Max, Sum, or Max-plus. Performance is poor if there is no inheritance because a) base tables are starved, and b) the scheduler cannot determine which base tables feed high versus low priority derived tables. Among the algorithms that use priority inheritance, their relative performance is often very similar (as in Figure 8), but, overall, Max performs the best while Sum performs the worst. This is because Max achieves the right balance between prioritizing a base table high enough for it to run in a timely manner, but not so high as to starve the actual high-priority tables.

4.6 Heterogeneous Jobs

We then developed a set of experiments to evaluate the partitioning algorithms and their variants on a heterogeneous job set. We used two classes of jobs: one with a period of 100, the other with a period of 10,000. We normalized the priority of the jobs to the inverse of their periods, so the period-100 jobs have a priority of 100 and the period-10,000 jobs have a priority of 1 (so that the jobs have equal weighted staleness at the end of their periods). A first question is whether partitioned scheduling is beneficial, and which partitioning algorithm works best. The result of this experiment is shown in Figure 9. The local Max-Benefit algorithm (i.e., no partitioning) clearly incurs a large relative lateness. The Partitioned algorithm is significantly better at lower utilizations, but performs worse than the non-partitioned algorithm at higher utilizations – because its local scheduling wastes processing resources. The Proportional algorithm is best overall. We used a clustering constant of $k = 1$.

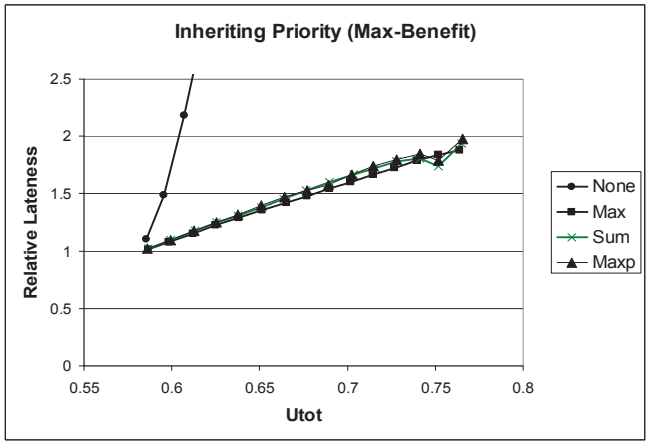

Fig. 8. Hierarchies and inheriting priorities.

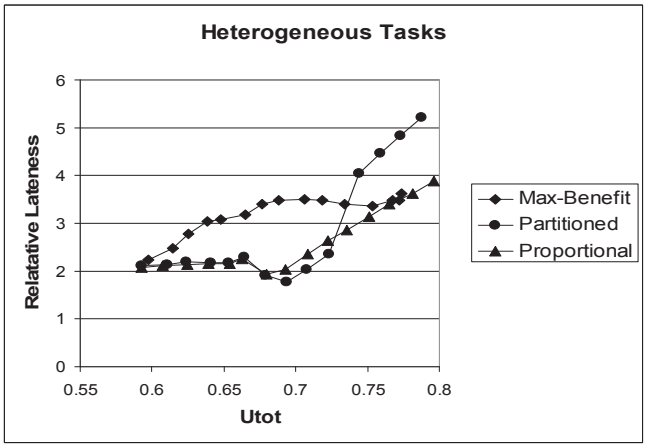

Fig. 9. Effects of heterogeneous jobs.

The performance graph of the Partitioned and the Proportional algorithms has a number of dips and sharp climbs. This effect is not due to instability in the simulator, rather the algorithms are entering different operating regions. We use non-aggressive scheduling for Partitioned and Proportional, and used a value of $\lceil Utot \rceil / Utot$ for M in the proportional algorithm – leaving free tracks at lower utilizations.

The variations in performance led us to investigate variants of the Proportional algorithm. First, we changed the value of M to N/U_{tot} (where N is the number of tracks), so that there are no free tracks. This change results in significantly better performance at lower utilizations, as shown in Figure 10. The "basic" version leaves free tracks, while the "all" version allocates all tracks. Next, we observed that the non-aggressive algorithm blocks higher-rated jobs in favor of lower rated jobs. By making the Proportional algorithm aggressive as well as allocat-

ing all tracks (referred to in Figure 10 as "all, aggr."), we achieve good performance at all utilization levels.

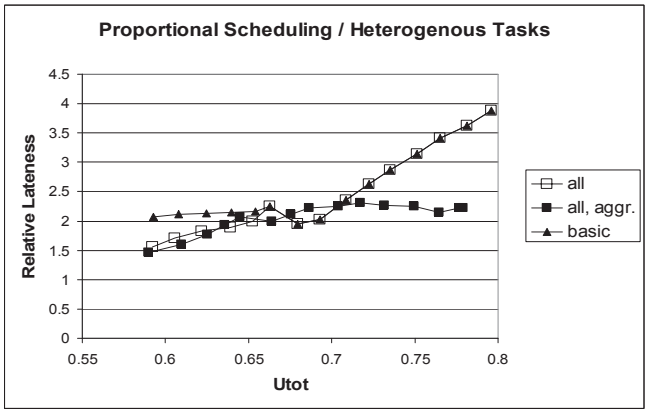

Fig. 10. Proportional scheduling variations.

4.7 Update Chopping

Finally, we evaluate the benefits of update chopping. Following slowdown periods, low-priority jobs may have accumulated a large freshness delta and can block high-priority jobs for a long time. We proposed update chopping to avoid this kind of blocking by adding a degree of pre-emptibility to the jobs. We report a representative set of results on a job set whose dependency graph consists of single-chain hierarchies with a depth of two (i.e., each base table is used to define one derived table). The base table jobs all have a priority of one and the derived tables have a priority of ten. We configured update chopping to limit the amount of data loaded at once to three times the job period. The results, illustrated in Figure 11, show that update chopping is an effective strategy.

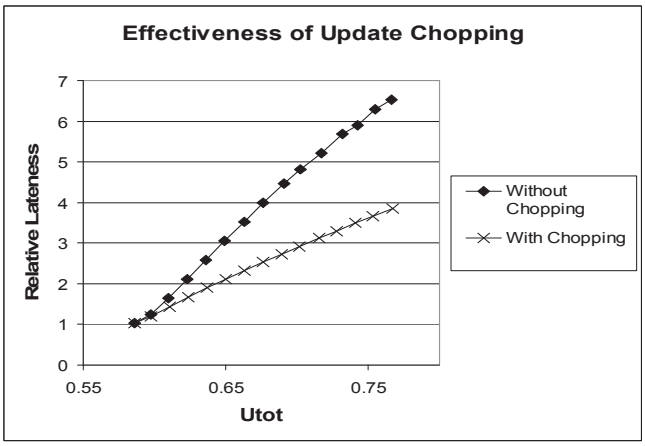

Fig. 11. Update chopping with two priority classes.

4.8 Lessons Learned

We observed that basic algorithms behave similarly, but an effective basic algorithm must incorporate table priorities. Furthermore, our Proportional algorithm (especially the "aggressive" variant that allocates all tracks) outperforms straightforward partitioned and global scheduling. Finally, we have shown the importance of inheriting priority when dealing with view hierarchies and the need for update chopping after periods of overload.

5 Related Work

This section compares the contributions of this paper with previous work in three related areas: scheduling, data warehousing, and data stream management.

5.1 Scheduling

The closest work to ours is [26], which finds the best way to schedule updates of tables and views in order to maximize data freshness. Aside from using a different definition of staleness, our Max Benefit basic algorithm is analogous to the max-impact algorithm from [26], as is our "Sum" priority inheritance technique. Our main innovation is the multi-track Proportional algorithm for scheduling the large and heterogeneous job sets encountered by a streaming warehouse. Additionally, we propose update chopping to deal with transient overload.

Another closely related work is [6], which studies the complexity of minimizing the staleness of a set of base tables in a streaming warehouse (i.e., hierarchies of views are not considered).

In general, interesting scheduling problems are often NP-hard in the off-line setting and hard to approximate off-line [4][14]. This motivates the use of heuristics such as our greedy Max Benefit algorithm.

While we believe that update scheduling in a streaming warehouse is novel, our solution draws from a number of recent scheduling results. In particular, there has been work on real-time scheduling of heterogeneous tasks on a multi-processor to address the tension between partitioned scheduling (which guarantees resources to short jobs) and global scheduling (which makes the best use of processing resources) [7][12][27]. The Pfair algorithm [5] and its variants have been proposed when tasks are preemptible; however we must assume that data loading tasks are non-preemptible. Our Proportional algorithm attempts to make a fair allocation of resources to non-preemptible tasks in a multi-track setting, and is the first such algorithm of which we are aware.

Overload management has been addressed in, e.g., [22][23]. However, these algorithms handle overload by dropping jobs, while a data warehouse can defer updates for a while, but cannot drop them.

Derived table relationships are similar to precedence constraints among jobs, e.g., a derived table cannot be updated until its sources have been updated. Previous work on scheduling with precedence constraints focused on minimizing the total time needed to compute a set of non-recurring jobs [24]. Our update jobs are recurring, and we use the notion of freshness delta to determine when a derived table update should be scheduled.

There has also been work on adding real-time functionality to databases. However, most of this work focuses on scheduling read-only transactions to achieve some quality-of-service guarantees (e.g., meeting deadlines) [21]. The works closest to ours are [1] and [20] which dis-

cuss the interaction of queries and updates in a firm real-time database, i.e., how to install updates to keep the data fresh, but also ensure that read transactions meet their deadlines. However, their system environments are significantly different from ours: transactions can be preempted, all tables in the database are "snapshot" tables, and updates are very short-lived (data are written to memory, not disk), meaning that they can be deferred until a table is queried. Similar work has also appeared in the context of Web databases, which aims to balance the quality of service of read transactions (requests for Web pages) against data freshness [34]. It also assumes a "snapshot" model rather than our append-only model.

5.2 Data Warehousing

There has been some recent work on streaming data warehousing, including system design [15], real-time ETL processing [28] and continuously inserting a streaming data feed at bulk-load speed [32]. These efforts are complementary to our work---they also aim to minimize data staleness, but they do so by reducing the running time of update jobs once the jobs are scheduled.

A great deal of existing data warehousing research has focused on efficient maintenance of various classes of materialized views [18], and is orthogonal to this paper. References [10][35] discuss consistency issues under various view maintenance policies. As discussed earlier, maintaining consistency in a streaming data warehouse is simpler due to the append-only nature of data streams.

There has also been work on scheduling when to pull data into a warehouse to satisfy data freshness guarantees [9][17][30]. This work does not apply to the push-based stream warehouses studied in this paper, which do not have control over the data arrival patterns.

Quantifying the freshness of a data warehouse was addressed in several works. For instance, [1] proposes two definitions: maximum age, which corresponds to the definition used in this paper, and unapplied update, which defines staleness as the difference between the current time and the arrival time of the oldest pending update. Unapplied update is not appropriate in a real-time stream warehouse that must handle sources with various arrival rates and inter-arrival times. For example, suppose that two updates have arrived simultaneously, one containing two minutes of recent data from stream 1, and the other carrying one day of data from stream 2. Clearly, the table sourced by stream 2 should be considered more out-of-date, yet both are equal under unapplied update. Reference [9] proposes to measure the average freshness over time, but their definition of freshness is far simpler than ours: a database object is assumed to have a freshness of one if it is up-to-date and zero otherwise.

5.3 Data Stream Management

One important difference between a DSMS and a data stream warehouse is that the former only has a limited working memory and does not store any part of the stream permanently. Another difference is that a DSMS may drop a fraction of the incoming elements during overload, whereas a streaming data warehouse may defer some update jobs, but must eventually execute them. Scheduling in DSMS has been discussed in [2][8][19][29], but all of these are concerned with scheduling individual operators inside query plans.

6 Conclusions

In this paper, we motivated, formalized, and solved the problem of non-preemptively scheduling updates in a real-time streaming warehouse. We proposed the notion of average staleness as a scheduling metric and presented scheduling algorithms designed to handle the complex environment of a streaming data warehouse. We then proposed a scheduling framework that assigns jobs to processing tracks and uses basic algorithms to schedule jobs within a track. The main feature of our framework is the ability to reserve resources for short jobs that often correspond to important frequently-refreshed tables, while avoiding the inefficiencies associated with partitioned scheduling techniques.

We have implemented some of the proposed algorithms in the DataDepot streaming warehouse, which is currently used for several very large warehousing projects within AT&T. As future work, we plan to extend our framework with new basic algorithms. We also plan to fine-tune the Proportional algorithm – in our experiments, even the aggressive version with "all" allocation still exhibits signs of multiple operating domains, and therefore can likely be improved upon (however, it is the first algorithm of its class that we are aware of). Another interesting problem for future work involves choosing the right scheduling "granularity" when it is more efficient to update multiple tables together, as mentioned in Section 2.1. We intend to explore the tradeoffs between update efficiency and minimizing staleness in this context.

Acknowledgments

We would like to thank James Anderson and Howard Karloff for their helpful discussions, and we thank the anonymous reviewers for their helpful comments.

References

[1] B. Adelberg, H. Garcia-Molina, and B. Kao, Applying update streams in a soft real-time database system, *SIGMOD 1995*, 245-256.

[2] B. Babcock, S. Babu, M. Datar, and R. Motwani, Chain: Operator scheduling for memory minimization in data stream systems, *SIGMOD 2003*, 253-264.

[3] S. Babu, U. Srivastava, and J. Widom, Exploiting k-constraints to reduce memory overhead in continuous queries over data streams, *ACM Trans. On Database Sys.*, 29(3), 545-580 (2004).

[4] S. Baruah, The non-preemptive scheduling of periodic tasks upon multiprocessors, Real Time Systems, 32(1-2):9-20 (2006).

[5] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel, Proportionate progress: A notion of fairness in resource allocation, *Algorithmica*, 15:600-625 (1996).

[6] M. H. Bateni, L. Golab, M. T. Hajiaghayi, and H. Karloff, Scheduling to minimize staleness and stretch in real-time data warehouses, *SPAA 2009*, 29-38.

[7] A. Burns. Scheduling hard real-time systems: a review, *Software Engineering Journal*, 6(3):116-128 (1991).

[8] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker, Operator scheduling in a data stream manager, *VLDB 2003*, 838-849.

[9] J. Cho and H. Garcia-Molina, Synchronizing a database to improve freshness, *SIGMOD 2000*, 117-128.

[10] L. Colby, A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross, Supporting multiple view maintenance policies, *SIGMOD 1997*, 405-416.

[11] M. Dertouzos and A. Mok, Multiprocessor on-line scheduling of hard- real-time tasks, *IEEE Trans. on Software. Eng.*, 15(12):1497-1506 (1989).

[12] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling. *Real-Time Systems*, 38(2):133-189 (2008).

[13] N. Folkert, A. Gupta, A. Witkowski, S. Subramanian, S. Bellamkonda, S. Shankar, T. Bozkaya, and L. Sheng, Optimizing refresh of a set of materialized views, *VLDB 2005*, 1043-1054.

[14] M. Garey and D. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, New York: W. H. Freeman, 1979.

[15] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk, Stream warehousing with DataDepot, *SIGMOD 2009*, 847-854.

[16] L. Golab, T. Johnson, and V.Shkapenyuk, Scheduling updates in a real-time stream warehouse, *ICDE 2009*, 1207-1210.

[17] H. Guo, P. A. Larson, R. Ramakrishnan, and J. Goldstein, Relaxed currency and consistency: How to say "good enough" in SQL, *SIGMOD 2004*, 815-826.

[18] A. Gupta and I. Mumick, Maintenance of materialized views: Problems, techniques, and applications, In IEEE Data Eng. Bulletin, Special Issue on Materialized Views and Data Warehousing, 18(2):3-18 (1995).

[19] M. Hammad, M. Franklin, W. Aref, and A. Elmagarmid, Scheduling for shared window joins over data streams, *VLDB 2003*, 297-308

[20] K. -D. Kang, S. Son, and J. Stankovic, Managing deadline miss ratio and sensor data freshness in real-time databases, *IEEE Trans. On Knowledge and Data Eng.*, 16(10):1200-1216 (2004).

[21] B. Kao and H. Garcia-Molina, An overview of real-time database systems, In Advances in Real-Time Systems (ed. S.H. Son), 463-486, Prentice Hall, 1995.

[22] G. Koren, D. Shasha. Dover, an optimal on-line scheduling algorithm for an overloaded real-time system, *RTSS 1992*, 292-299.

[23] G. Koren, D. Shasha. An approach to handling overloaded systems that allow skips, *RTSS 1995*, 110-119.

[24] Y. K. Kwok and I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, *ACM Computing Surveys*, 31(4):406-471 (1999).

[25] W. Labio, R. Yerneni, and H. Garcia-Molina, Shrinking the warehouse update window, *SIGMOD 1999*, 383-394.

[26] A. Labrinidis and N. Roussopoulos, Update propagation strategies for improving the quality of data on the Web, *VLDB 2001*, 391-400.

[27] Y. Oh and S.H. Son. Tight performance bounds of heuristics for a real-time scheduling problem. Tech. report CS-93-24, U. Virginia (1993).

[28] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.-E. Frantzell, Supporting Streaming Updates in an Active Data Warehouse, *ICDE 2007*, 476-485.

[29] M. Sharaf, P. Chrysanthis, A. Labrinidis, and K. Pruhs, Algorithms and metrics for processing multiple heterogeneous continuous queries, *ACM Trans. On Database Sys.*, 33(1) (2008).

[30] R. Srinivasan, C. Liang, and K. Ramamritham, Maintaining temporal coherency of virtual data warehouses, *RTSS 1998*, 60-70.

[31] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, Load shedding in a data stream manager, *VLDB 2003*, 309-320.

[32] C. Thomsen, T. B. Pedersen, and W. Lehner, RiTE: Providing on-demand data for right-time data warehousing, *ICDE 2008*, 456-465.

[33] P. Tucker, *Punctuated Data Streams*, Ph.D. Thesis, Oregon Health & Science University, 2005.

[34] H. Qu and A. Labrinidis, Preference-aware query and update scheduling in Web-databases, ICDE 2007, 356-365.

[35] Y. Zhuge, J. Wiener, and H. Garcia-Molina, Multiple view consistency for data warehousing, *ICDE 1997*, 289-300.

AUTHOR BIOGRAPHIES

Lukasz Golab is a member of the Database Research department at AT&T Labs – Research. His research interests include data stream management, data warehousing and data quality analysis. He received his Ph.D. from the University of Waterloo in 2006, winning the Alumni Gold Medal as the top PhD student at the university. Lukasz has co-authored a book on Data Stream Management as well as 20 refereed papers.

Theodore Johnson received a Ph.D. in Computer Science from NYU in 1990. From 1990 through 1996 Theodore was an assistant, then associate, professor in the CISE department of the University of Florida. In 1996 he joined the Database Research department of AT&T Labs - Research, where he works today. His research interests focus on data quality, large-scale data warehousing, and data stream management systems. Theodore has written three applications in widespread use within AT&T, GS (a data stream management system), DataDepot (a very large scale data warehousing system) and Bellman (a data quality browser for large and complex databases). He has co-authored two books, "Distributed Systems and Algorithms", and "Exploratory Data Mining and Data Cleaning", and more than 80 refereed papers.

Vladislav Shkapenyuk is a senior member of technical staff in the Database Research department of AT&T Labs – Research. He received his Ph.D. from Rutgers University in 2007. Vladislav's research interests include data stream management, large-scale real-time data warehousing and data feed management. He is a major contributor to several systems widely used within AT&T, including GS (data stream management system for network monitoring), Update Manager (real-time update scheduler for data warehouses) and Bistro (data feed management system). Vladislav co-authored 16 referred papers.