An Adaptive Scheme for Overload Handling in Active Data Warehouses

Hennadiy Leontyev*, Theodore Johnson† and James H. Anderson*
*University of North Carolina at Chapel Hill
{*leontyev, anderson*}*@cs.unc.edu*
†AT&T Labs – Research
johnsont@research.att.com

***Abstract*—This paper presents a novel adaptive approach for scheduling updates in a data warehouse that processes "near-real-time" data streams. Data is pushed to the warehouse from a variety of external sources with a wide range of inter-arrival times (e.g., once a minute to once a day). Due to network conditions, the volume of incoming data can widely vary and data streams can experience intermittent outages. Maintaining data freshness in the presence of outages and load variations can be challenging. In prior work, ad hoc heuristic algorithms have been proposed for doing this. In this paper, a systematic approach based upon global multiprocessor real-time scheduling theory is considered. The proposed approach can handle overload and recovery situations and maintain guarantees on data freshness for warehouse tables that are not impacted by outages. Simulation experiments are presented that show the effectiveness of the proposed approach.**

I. Introduction

Data stream warehouses (or, active data warehouses [16]) are used in critical business applications where near-real-time access to streamed data and integrated access to historical data are required [2], [9]. The DataDepot warehouse designed at AT&T is one example [9]. DataDepot was designed to allow network performance and potential attacks to be monitored by collecting system logs, IP packet traces, traffic summaries, etc.

As explained below, the query-generated workload in systems like DataDepot can be both highly variable and computationally-intensive, necessitating the use of multiprocessor platforms. Thus, when deploying such a system, one is faced with a complex multiprocessor real-time scheduling problem where adaptivity arises. In current warehouse designs, this scheduling problem is solved by using ad hoc methods. In this paper, we examine whether recent research on real-time multiprocessor scheduling can be leveraged to solve this problem in a more systematic way that lends itself to formal analysis. Such analysis is important because it enables principled system design and optimization, which is very important to database designers.

This paper was specifically motivated by issues faced in deployments of DataDepot, the architecture of which is illustrated in Fig. 1. DataDepot consists of a network-accessible disk storage, on which a hierarchy of database tables is stored, and one or more servers that run user queries and warehouse maintenance programs called *update tasks*. New information comes into the warehouse from a number of data feeds that

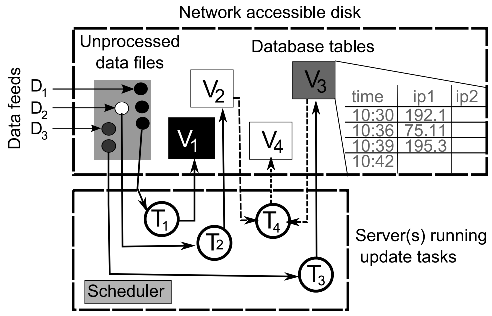

Fig. 1. Data warehouse architecture.

receive *data files* at regular intervals. These files are written onto disk and are accessible by update tasks. Each such task is associated with a particular table and *appends* records to that table by reading data files that come from data feeds or other (source) tables. Each server has a scheduler that manages the update tasks assigned to it. Because the data transformation and loading process is time- and memory-consuming, only a few update tasks can run simultaneously [12], [16]. The supported tables are of two types: *base* tables, which are sourced directly from data streams, and *derived* tables (materialized views), which are sourced from (i.e., defined as the result of an SQL query over) one or more base or other derived tables. Fig. 1 depicts several update tasks and their corresponding tables; V_1, V_2, and V_3 are base tables sourced from data feeds D_1, D_2, and D_3, respectively, and V_4 is a derived table.

The problem. Maintaining high *data freshness* (which requires that the timestamp of the mostly-recently loaded data be "close" to the current time) is particularly challenging if data feeds experience intermittent outages. Outages may occur due to equipment and network failures or when data schemes are changed, which may necessitate temporarily disabling certain feeds. During an outage, new data files can become backed up at the data-feed source and hence be unavailable for prolonged intervals of time. When this occurs, the affected base tables and their derived tables do not reflect the most recent information available in unprocessed data files. In this case, such a table is said to be *late*. In order to restore the freshness of late tables, their respective update tasks have to be given a larger share of the computing resources. However, since only few update tasks can run simultaneously, such share increases may excessively delay updates for non-late tables, and hence, their freshness could also be compromised (see examples in Sec. II for details). In this paper, we develop a scheduling algorithm called the *Adaptive Update Scheduler* (AUS) that is capable of recovering tables after feed outages while maintaining *guarantees* on data freshness and minimizing interference on tables that are not affected by outages.

Our approach. Under AUS, each update task can execute in either *normal* or *recovery* mode as characterized by normal and recovery *periods*, respectively; such periods represent the minimum time between consecutive task invocations. Invocations of an update task, called *jobs* of that task, are assigned *deadlines* according to the task's current period and are scheduled using the global earliest-deadline-first (GEDF) algorithm. A task's normal period is defined so that it just keeps up with arriving data when its table is healthy (i.e., not impacted by outages). Its recovery period is defined to be shorter so that it can process backlogged work due to an outage at a higher rate. Using prior work, it can be shown that AUS limits the total processing capacity consumed by tasks in recovery mode [4], [5]. Additionally, as we show, it allows table freshness to be bounded for healthy tables.

Prior work. The append-only nature of real-time data warehouses is a significant difference in comparison to more conventional real-time databases (RTDBs), which perform transactions on timed data [11], [17], [18]. Still, some similarities exist. In a data warehouse, the goal is to collect a history of events so that the leading edge of the event history is as fresh as possible. The notion of freshness is similar to the concept of "temporal consistency" as applied to data objects. An object is *absolutely consistent* if the difference between the current time and the object's timestamp does not exceed some pre-defined validity threshold [18]. In prior work, these validity thresholds have been generally used for determining priorities and timing constraints of transactions. In contrast, due to the append-only nature of data streams in our setting, stored data is always valid, so validity thresholds are effectively infinite. This allows some leeway in scheduling. Additionally, to our knowledge, global scheduling algorithms have not been considered before in work on RTDBs.

The issue of overload has been partially considered in prior work on data-warehouse scheduling [3], [6], [8], [9]. However, to our knowledge, all existing schedulers use heuristics in a way that does not allow performance to be predicted.

A number of utility-based scheduling approaches have been proposed for dealing with overloads in non-database [7], [14] and database settings [10]. Though such approaches can gracefully handle performance degradation, it might be problematic (or impossible) to determine meaningful utility functions in a warehouse setting. For example, while utilities are often defined to decay rapidly (or immediately) beyond a job's deadline, warehouse updates are considered to be of utility well beyond their deadlines.

In work on dynamically-changing workloads, feedback-control-based algorithms have proved useful. Such an algorithm typically monitors the number of deadline misses of jobs and adjusts their priorities in order to achieve satisfactory performance [1], [15]. Our approach can be seen as an example of a such an algorithm where a very simple controller is used.

Contributions. The AUS algorithm mentioned earlier is the main contribution of this paper. It has been designed by exploiting the fact that certain global multiprocessor schedulers can ensure bounded maximum job response times even if tasks dynamically change their timing requirements, as long as the processing platform is not over-utilized (utilization constraints beyond this are not required) [4], [5]. We show that, given such response-time bounds, bounds on data staleness can be derived (staleness is the inverse of freshness). After deriving such bounds, we discuss the results of experiments in which AUS is compared to the state-of-the-art heuristic-based "proportional" (PRP) scheduler [8] under overload conditions. In this evaluation, a synthetic task set was run on a proprietary warehouse simulator designed at AT&T. In these experiments, AUS exhibited more predictable performance than PRP. Note that, while staleness bounds can be computed for AUS, this is virtually impossible for PRP, given its heuristic nature.

In the rest of the paper, we formally define our system model (Sec. II), describe the AUS algorithm (Sec. III), analyze it by presenting per-job response time bounds (Sec. IV) and data staleness bounds (Sec. V), present our experimental results (Sec. VII), and conclude (Sec. VIII).

II. System Model

As mentioned earlier, we study a relational data warehouse, which is similar to that considered in prior work [8]. We assume that relationships among source and derived tables form a directed acyclic graph and do not change over time. We denote the set of tables as $\{V_1, \ldots, V_n\}$. For a derived table V_k, the set of its source tables is denoted $pred(V_k)$. For a base table V_k, $pred(V_k) = \emptyset$. Each table V_k is characterized by a *nominal update period* p_k that specifies the desired update frequency for this table.

Data stream model. Our data stream model is captured by the following definitions.

Definition 1. For base table V_k, a unique source *data feed* D_k exists. D_k is a sequence $\{D_{k,j} = (\mathsf{arr}(D_{k,j}), \mathsf{ts}(D_{k,j}))\}$, where $j \geq 1$. Each element of the sequence describes a *data file* that needs to be ingested into the base table. A data file is a logical unit of work. $\mathsf{arr}(D_{k,j})$ is the *arrival time* of the j^{th} data file and $\mathsf{ts}(D_{k,j})$ is its *timestamp*, which is the maximum timestamp of a record in that file. A data file's arrival time is the time instant when it becomes available on the disk. The timestamp of a record in the file denotes the physical time when that record was created. We assume that, for each base table V_k, (1) below holds, and for each $j \geq 1$, $\mathsf{ts}(D_{k,j}) \leq \mathsf{arr}(D_{k,j})$, $\mathsf{arr}(D_{k,j}) \leq \mathsf{arr}(D_{k,j+1})$, and $\mathsf{ts}(D_{k,j}) \leq \mathsf{ts}(D_{k,j+1})$.

$$\mathsf{ts}(D_{k,1}) > 0 \tag{1}$$

In order to establish guarantees on the quality of data, we assume that data file $D_{k,j+1}$ only contains records with timestamps within $(\mathsf{ts}(D_{k,j}), \mathsf{ts}(D_{k,j+1})]$ (note that $D_{k,1}$ contains records within $(0, \mathsf{ts}(D_{k,1})]$). Additionally, we assume that data files are processed in order of their arrival.

Definition 2. We call a data feed D_k *healthy* at time t, if there exists $D_{k,j}$ such that $\mathsf{arr}(D_{k,j}) \in [t - p_k - J_k, t]$ and $\mathsf{arr}(D_{k,j}) \in [\phi_k + p_k \cdot (j-1) - J_k, \phi_k + p_k \cdot (j-1)]$, and $\mathsf{ts}(D_{k,j}) \in [\mathsf{arr}(D_{k,j}) - \kappa_k, \mathsf{arr}(D_{k,j})]$, where $\phi_k \geq 0$ is D_k's *phase*, $J_k \in [0, p_k)$ is its *arrival jitter*, and $\kappa_k \geq 0$ is its *timestamp jitter*. A data feed that is not healthy is called *unhealthy*.

Intuitively, in a healthy data feed, data files arrive with some regularity and there is not much discrepancy between the arrival time and the maximum record timestamp in the file. In contrast, if a feed breaks and then is later restored after some noticeable time, then many data files will have the same arrival time and the discrepancy between record timestamps and the arrival time can be large. We say that feed D_k experiences an *outage* during an interval (t_1, t_2) if $t_2 - t_1 > p_k + J_k$ and no new data files arrive during (t_1, t_2).

Definition 3. We call a data file *pending* at time t if it has arrived but has not been uploaded into a base table by time t. The set of pending data files for base table V_k at time t is called the *backlog* of V_k and is denoted $\mathsf{BL}(V_k, t)$. We assume that data files in $\mathsf{BL}(V_k, t)$ are sorted by increasing timestamp.

Table freshness and data consistency. We next define table freshness (which reflects the quality of stored data) and discuss the notion of data consistency adopted in this paper.

Definition 4. [8] The *freshness* of table V_k at time t, denoted $\mathsf{F}(V_k, t)$, is the maximum timestamp of a record stored in V_k by time t. We define $\mathsf{F}(V_k, 0) = 0$.

Definition 5. [8] The *staleness* of table V_k at time t is defined as $\mathsf{S}(V_k, t) = t - \mathsf{F}(V_k, t)$.

Staleness indicates the extent to which the most recent timestamp stored in a table lags behind the current time. The state of derived tables must be consistent with the state of their source tables at some time instant in the past. In this paper, we follow the definition of "trailing edge consistency" [8].

Definition 6. We define the *trailing edge* of table V_k at time t as

$$\mathsf{TE}(V_k, t) = \begin{cases} \max\{\mathsf{ts}(D_{k,y}) \mid \mathsf{arr}(D_{k,y}) \leq t\} & \text{if } V_k \text{ is base,} \\ \min\limits_{V_i \in pred(V_k)} \{\mathsf{F}(V_i, t)\} & \text{otherwise.} \end{cases}$$

Intuitively, for base table V_k, the trailing edge at time t defines a time instant such that all data with timestamps at or before time t has arrived from its data feed. If V_k is derived, then the trailing edge at time t indicates that each of its source tables contains records with timestamps up to time $\mathsf{TE}(V_k, t)$.

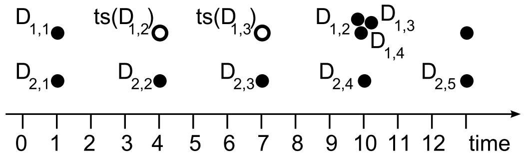

Fig. 2. Data file arrivals in Example 1.

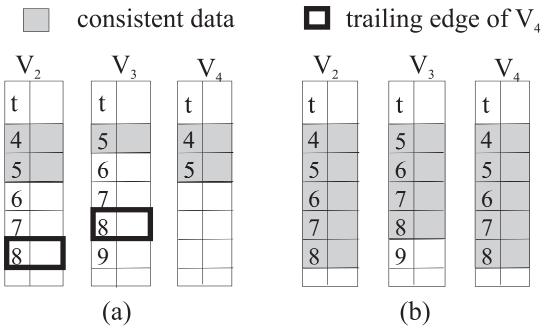

Fig. 3. State of tables at times **(a)** 12 and **(b)** 14 in Example 2.

As illustrated by the two examples below, the discrepancy between $\mathsf{TE}(V_k, t)$ and $\mathsf{F}(V_k, t)$ indicates that table V_k needs to be updated. The following predicate indicates that table V_k reflects the most recent consistent information.

$$\mathsf{FRESH}(V_k, t) \triangleq (\mathsf{TE}(V_k, t) \leq \mathsf{F}(V_k, t)) \tag{2}$$

Each table V_k is updated by an external program referred to as *task* T_k. For base tables, the data is loaded from pending data files, and for derived tables, data from other tables is used.

Example 1. Suppose that the base tables V_1 and V_2 in Fig. 1 are sourced from data feeds D_1 and D_2 and, for data feed D_2, data files $D_{2,j}$ have timestamps and arrival times $\mathsf{arr}(D_{2,j}) = \mathsf{ts}(D_{2,j}) = 1 + 3 \cdot (j-1)$ for $j \geq 1$. Suppose that, for data feed D_1, data files $D_{1,1}, \ldots, D_{1,10}$ have timestamps $\mathsf{ts}(D_{1,j}) = 1 + 3 \cdot (j-1)$ for $j = 1, \ldots, 10$ and arrival times $\mathsf{arr}(D_{1,1}) = 1$ and $\mathsf{arr}(D_{1,j}) = 10$ for $j \geq 2$, respectively. This is illustrated in Fig. 2, where data file arrivals are denoted with black circles. For those data files with timestamps that do not coincide with file's arrival time, their timestamps are shown using empty circles. This example scenario could occur if D_1 and D_2 are required to supply data periodically every 3 time units but D_1 goes down during the interval $[4, 10)$ so that several data files are available at once at time 10. At time 1, $D_{1,1}$ arrives, and hence, by Def. 6, $\mathsf{TE}(V_1, 1) = \mathsf{ts}(D_{1,1}) = 1$. Similarly, $\mathsf{TE}(V_2, 1) = \mathsf{ts}(D_{2,1}) = 1$. By Def. 3, $\mathsf{BL}(V_1, 10) = \{D_{1,2}, D_{1,3}, D_{1,4}\}$, and hence, by Def. 6, $\mathsf{TE}(V_1, 10) = \mathsf{ts}(D_{1,4}) = 1 + 3 \cdot (4-1) = 10$.

Example 2. Suppose that at time 12 the derived table V_4 in Fig. 1 contains data records up to time 5 and its source tables V_2 and V_3 have freshness 8 and 9, respectively, as shown in Fig. 3(a). Table V_4 is consistent with respect to the state of its sources as of time 5. When task T_4 commences execution at time 12 it needs to add new records to V_4 to reflect the changes in its source tables. At time 12, both source tables have all required information up to time $\mathsf{TE}(V_4, 12) = \min_{V_i \in pred(V_4)}\{\mathsf{F}(V_i, 12)\} = \min\{8, 9\} = 8$. Therefore, task T_4 reads all data with timestamps within the range $(\mathsf{F}(V_4, 12), \mathsf{TE}(V_4, 12)] = (5, 8]$ from the source tables and appends the corresponding records to V_4. If task T_4 finishes at time 14, then the state of V_4 is consistent with respect to the state of V_2 and V_3 as of time 8 as shown in Fig. 3(b).

Adaptable task model. In this paper, tasks are characterized by several parameters as specified in the *adaptable sporadic task model*, proposed by Block et. al [5]. We present this model below, and introduce some additional constraints imposed by the specifics of the update scheduling problem.

Changes in $\mathsf{TE}(V_k, t)$ trigger task invocations; each such invocation is called a *job*. The j^{th} job of T_k is denoted $T_{k,j}$ and is characterized by its *release time* $r_{k,j}$, which is the earliest time $T_{k,j}$ can start its execution, and an *absolute deadline* $d_{k,j} \geq r_{k,j}$. We let $s_{k,j} \geq r_{k,j}$ denote the earliest time when job $T_{k,j}$ is scheduled. The completion time of $T_{k,j}$ is denoted $f_{k,j}$. We define $r_{k,0} = d_{k,0} = f_{k,0} = 0$ for each T_k. The *worst-case execution time* of T_k's jobs is denoted e_k.

Due to the nature of the underlying database update process, each job has to be executed non-preemptively and sequentially, i.e., no two jobs of the same task can be scheduled in parallel. In this paper, we assume that for base tables, each job processes exactly one data file. However, all presented results are applicable if jobs are allowed to process multiple data files provided that $e_k \leq p_k$ holds for each task T_k.

Definition 7. Job $T_{k,j}$ is *ready* at time t if $t \geq r_{k,j}$ and its predecessor $T_{k,j-1}$ (if it exists) completes by time t. Only ready jobs can be scheduled.

To adapt to changes in the incoming workload, tasks can independently change their reservations for processing capacity. For task T_k, these reservations are expressed in terms of a task *period*, $\mathsf{p}(T_k, t) \geq e_k$, which defines the minimum time between consecutive job invocations. In Sec. III, we define task periods such that, under normal conditions, jobs of T_i are invoked at the frequency corresponding to V_i's nominal update period. Task periods are used to calculate job deadlines and subsequent release times as follows. When job $T_{k,j}$ is released, its absolute deadline is

$$d_{k,j} = r_{k,j} + \mathsf{p}(T_k, r_{k,j}). \tag{3}$$

Jobs are released so that some minimum separation between consecutive job releases is maintained. In the absence of period changes, for $j \geq 0$, release times are defined according to the following equation.

$$r_{k,j+1} = \begin{cases} d_{k,j} & \text{if } \neg\mathsf{FRESH}(V_k, \max(d_{k,j}, f_{k,j})), \\ \min\{t \mid t \geq \max(d_{k,j}, f_{k,j}) \wedge \neg\mathsf{FRESH}(V_k, t)\} & \\ & \text{otherwise.} \end{cases} \tag{4}$$

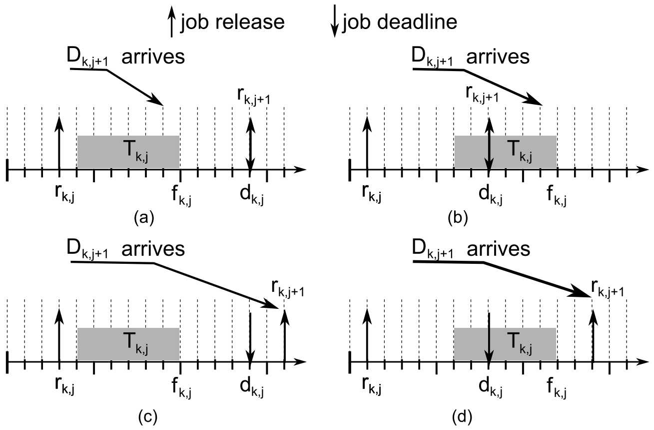

Fig. 4. Illustration of (4) in Example 3.

Example 3. Fig. 4 illustrates (4) when V_k is a base table. In this and subsequent schedules, down-arrows denote job deadlines and up-arrows denote job releases. (Coinciding up- and down-arrows appear appear as a double arrow.) Insets (a) and (b) illustrate the first case of (4). In inset (a), job $T_{k,j}$ completes before its deadline $d_{k,j}$ (i.e., $f_{k,j} \leq d_{k,j}$). However, since data file $D_{k,j+1}$ is not processed by time $d_{k,j}$, table V_k is not fresh at time $d_{k,j}$, and hence, $r_{k,j+1} = d_{k,j}$. Inset (b) is similar except that $T_{k,j}$ finishes after its deadline. Insets (c) and (d) illustrate the second case of (4). In inset (c), $T_{k,j}$ completes by time $d_{k,j}$ and data file $D_{k,j+1}$ arrives after time $d_{k,j}$. Thus, table V_k is fresh at time $d_{k,j}$, and hence, the next release is set to be $r_{k,j+1} = \mathsf{arr}(D_{k,j+1})$, (i.e., the earliest time when the table is not fresh). Inset (d) is similar, except that $T_{k,j}$ completes after its deadline.

As seen, (4) ensures some minimum separation between consecutive job releases if the current task period does not change. In Sec. III, we discuss how release times are calculated if task periods change.

Example 4. Consider the system in Example 1. As before, tables V_1 and V_2 are updated by tasks T_1 and T_2; we assume here that they are scheduled on one processor so that released jobs of T_2 are scheduled first. Insets (a) and (b) of Fig. 5 show schedules in which $\mathsf{p}(T_1, t) = 1$ and $\mathsf{p}(T_1, t) = 1.5$, respectively. In both schedules, $\mathsf{p}(T_2, t) = 3$. In these schedules, the numbers above the arrows indicate $|\mathsf{BL}(V_k, t)|$. Each job is denoted by its index, which is also the index of the data file being processed.

As shown in Example 1, $\mathsf{TE}(V_1, 1) = \mathsf{ts}(D_{1,1}) = 1$. Moreover, because T_1 is not scheduled prior to time 1, $\mathsf{F}(V_1, 1) = 0$, which, by (2), implies $\mathsf{FRESH}(V_1, 1) =$ *false*. This, by the second case of (4), triggers the release of $T_{1,1}$ (note that $d_{1,0} = f_{1,0} = 0$) so that $r_{1,1} = \min\{t \mid t > \max(d_{1,0}, f_{1,0}) \wedge \neg\mathsf{FRESH}(V_1, t)\}) = 1$. Assuming periods as defined in Fig. 5, by (3), $d_{1,1} = r_{1,1} + \mathsf{p}(T_1, r_{1,1}) = 1 + 1 = 2$. Similarly, $r_{2,1} = 1$ and $d_{2,1} = r_{2,1} + \mathsf{p}(T_2, r_{2,1}) = 1 + 3 = 4$.

Update jobs cause table freshness (and hence, staleness) to change. When an update job $T_{k,j}$ of table V_k starts to

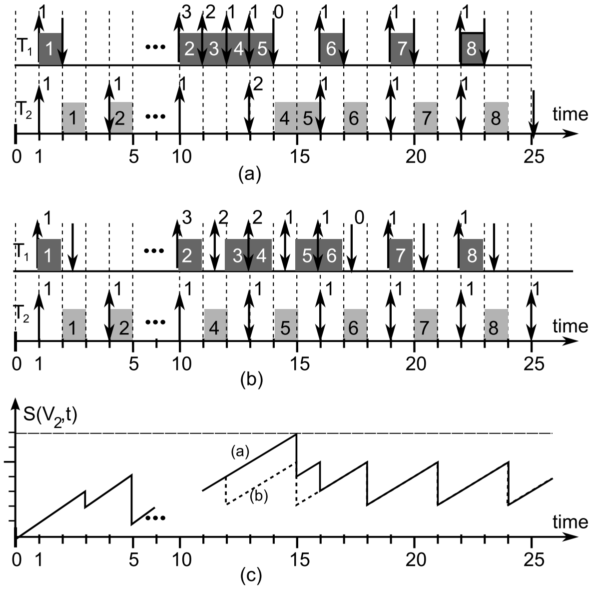

Fig. 5. Schedule with **(a)** $\mathsf{p}(T_1, t) = 1$ and **(b)** $\mathsf{p}(T_1, t) = 1.5$ in Examples 4 and 5. (Note that each schedule is compressed in the interval $[5, 10)$ to fit the column.) **(c)** Table staleness for V_2 in Example 6.

execute at time t, it loads records with timestamps within the range $(\mathsf{F}(V_k, t), \mathsf{NF}]$ into V_k, where $\mathsf{NF} \leq \mathsf{TE}(V_k, t)$ is the new freshness value that will be set. (Records with timestamps up to $\mathsf{F}(V_k, t)$ would have been loaded by $T_{k,j}$'s predecessors.) Therefore, the freshness of V_k at time $f_{k,j}$ becomes $\mathsf{F}(V_k, f_{k,j}) = \mathsf{NF}$.

Example 5. Consider again the system illustrated in Fig. 5. At time 2, job $T_{1,1}$ uploads $D_{1,1}$ and sets $\mathsf{F}(V_1, 2) = \mathsf{ts}(D_{1,1}) = 1$. Now consider time 11 in schedule (a), when job $T_{1,2}$ finishes uploading data file $D_{1,2}$ and updates the freshness of V_1. At this time, $\mathsf{F}(V_1, 11) = \mathsf{ts}(D_{1,2}) = 4$, where the last equality holds because $\mathsf{ts}(D_{1,j}) = 1+3\cdot(j-1)$ for all j in Example 1. Since the latest data file that has arrived at time 11 is $D_{1,4}$, by Def. 6, $\mathsf{TE}(V_1, 11) = \mathsf{ts}(D_{1,4}) = 1+3\cdot(4-1) = 10$. Thus, by (2), $\mathsf{FRESH}(V_1, 11) = \textit{false}$ and hence, by (4), $r_{1,3} = d_{1,2} = r_{1,2} + \mathsf{p}(T_1, r_{1,2}) = 10 + 1 = 11$. In contrast, in schedule (b), $r_{1,3} = d_{1,2} = r_{1,2} + \mathsf{p}(T_1, r_{1,2}) = 10 + 1.5 = 11.5$.

As seen in the above example, table freshness changes incrementally. The magnitude of each incremental change is characterized by the following definition.

Definition 8. Let

$$\mathsf{L}_{k,j} = \begin{cases} \min\{\mathsf{BL}(V_k, s_{k,j})\} - \mathsf{F}(V_k, s_{k,j}) & \text{if } V_k \text{ is base,} \\ \min(\mathsf{TE}(V_k, s_{k,j}) - \mathsf{F}(V_k, s_{k,j}), p_k) & \text{otherwise,} \end{cases}$$

where $\min\{\mathsf{BL}(V_k, s_{k,j})\} = \min\{\mathsf{ts}(D_{k,y}) \mid D_{k,y} \in \mathsf{BL}(V_k, s_{k,j})\}$. $\mathsf{L}_{k,j}$ is called the *update length* of $T_{k,j}$ and is the amount by which table freshness increases when $T_{k,j}$ finishes. By the first case of the above equality, the update length for the base table is the difference between the earliest unprocessed data file and current table freshness. It can be shown that $\mathsf{BL}(V_k, s_{k,j}) \neq \emptyset$ and hence update lengths are always defined. For base tables, job $T_{k,j}$ processes the earliest data file pending at time $s_{k,j}$, and data files that arrive during $T_{k,j}$'s execution are processed by $T_{k,j}$'s successors.

Intuitively, for base tables, the next update job is released only if the trailing edge differs from the table's freshness, and hence, there are pending data files. In contrast to the original definition in [8], we limit the update length for derived tables by T_k's period in Def. 8 (see the second case) in order to achieve a predictable worst-case execution time for $T_{k,j}$. For base tables, update lengths are not limited because data files are indivisible units of work.

Table V_k's freshness is updated according to the rules below. Note that these rules apply for both base and derived tables.

Rule 1: At time zero, $\mathsf{F}(V_k, 0) = 0$.
Rule 2: At time t, $\mathsf{F}(V_k, t) = \mathsf{F}(V_k, f_{k,h})$, where $T_{k,h}$ is the latest completed job of T_k by time t. (If no such job exists, then $\mathsf{F}(V_k, t) = \mathsf{F}(V_k, f_{k,0}) = 0$.)
Rule 3: At the completion of job $T_{k,j}$ the freshness of V_k is set to $\mathsf{F}(V_i, f_{k,j}) = \mathsf{F}(V_k, s_{k,j}) + \mathsf{L}_{k,j}$.

We next illustrate the rules presented above for the case of a base table.

Example 6. Fig. 5(c) shows $\mathsf{S}(V_2, t)$, for the schedules shown in insets (a) and (b) of the figure. At time zero, $\mathsf{S}(V_2, 0) = 0$. To see that staleness changes as in inset (c), consider first the interval $[0, 3)$. Since no job of T_2 completes within this interval, by Rule 2, $F(V_2, t) = 0$, and hence, by Def. 5, $\mathsf{S}(V_2, t)$ increases linearly. As shown in Fig. 5(a,b), the first update job of T_2, $T_{2,1}$, released at time $r_{2,1} = 1$, is scheduled at time $s_{2,1} = 2$ and completes at time $f_{2,1} = 3$. By Def. 6, $\mathsf{TE}(V_2, s_{2,1}) = 1$. By Rule 2, $\mathsf{F}(V_2, s_{2,1}) = 0$. Thus, by Def. 8, the update length of $T_{2,1}$ is $\mathsf{L}_{2,1} = 1$. Therefore, $T_{2,1}$ loads all data with timestamps from 0 to 1 into V_2, and hence, by Rule 3, $\mathsf{F}(V_2, f_{2,1}) = \mathsf{F}(V_2, s_{2,1}) + \mathsf{L}_{2,1} = 0 + 1 = 1$. We thus have $\mathsf{S}(V_2, f_{2,1}) = 3 - 1 = 2$. Consider now the interval $[10, 15)$ in schedule (a). By Rule 2, for each $t \in [10, 15)$, $\mathsf{F}(V_2, t) = \mathsf{F}(V_2, f_{2,3}) \overset{\{\text{by Rule 3}\}}{=} \mathsf{F}(V_2, s_{2,3}) + L_{2,3} \overset{\{\text{by Def. 8}\}}{=} \mathsf{ts}(D_{2,3}) = 7$. By Def. 5, $\mathsf{S}(V_2, t) = \max(0, t - \mathsf{F}(V_2, t)) = t–7$, so staleness increases linearly during $[10, 15)$. In contrast, in schedule (b), job $T_{2,4}$, released at time $r_{2,4} = 10$ in response to the arrival of $D_{2,4}$, completes at time 12 and sets $\mathsf{F}(V_2, 12) = 10$, so that staleness is $\mathsf{S}(V_2, 12) = 2$.

Response time and overload management. The *response time* of job $T_{k,j}$ is the delay between its release time and completion, $f_{k,j} - r_{k,j}$. Task T_k's *maximum response time* is $\max_j(f_{k,j} - r_{k,j})$. In Sec. V, we show that if data feeds supply new data at a steady rate, then it is possible to establish analytical bounds on maximum table staleness provided that jobs are scheduled using a real-time scheduling algorithm that ensures that inequality (5) below holds for each job $T_{k,j}$.

$$f_{k,j} \leq r_{k,j} + \Theta_k \tag{5}$$

In (5), Θ_k is an upper bound on T_k's maximum response time.

In practice, new data can arrive unevenly due to feed outages (as illustrated in Example 1) or due to changing network conditions. Additionally, job execution times may vary due to increases in data volumes being pushed or due to hardware issues. Such scenarios can cause overload situations in which the difference between the trailing edge and table freshness temporarily (or permanently) increases compared to an overload-free execution. A scheduler must handle such overloads as gracefully as possible. Its objectives could include minimizing recovery time for critical tables, preserving data quality for tables that do not experience feed outages, etc.

Example 7. In the schedule shown in Fig. 5(a), the scheduler attempts to recover V_1 as quickly as possible at the price of delaying updates to table V_2 and drastically increasing its staleness (see Fig. 5(c)). In contrast, in the schedule shown in Fig. 5(b), the maximum staleness of V_1 does not grow compared to intervals where new data files arrive regularly. However, it takes longer to process the late data files $D_{1,2}, D_{1,3}$. Also, in the case of arbitrarily long outages, it is not possible to establish an upper bound on the response time of T_2's jobs under the first scheduler. For the second scheduler, however, response-time bounds can be found easily (see Sec. V).

In the next section, we present our algorithm for handling overload situations. This algorithm is based upon the simple idea that, if there are too many pending updates for a table, then the respective update task needs to be given a larger processor share. Similar behavior has been illustrated in Example 5.

III. Adaptive Update Scheduler

The operation of AUS can be summarized as shown in Fig. 6. Each task operates in either normal or recovery mode. Update jobs are released as described in Sec. II with deadlines and release times calculated using (3) and (4), respectively. At most m ready jobs, where m is the number of processors, are selected for execution using non-preemptive global EDF so that jobs with smaller deadlines have higher priority. When new data files arrive or jobs complete, the scheduler monitors the state of each table and decides whether the operational mode for one or more tasks needs to be changed. If one or more tasks need to change their operational modes, then an optimizer component (OPT in Fig. 6) calculates new task periods in two steps. First, if some tasks transition from recovery to normal mode, their periods are increased hence creating spare processing capacity (because their jobs are released less frequently). After that processing capacity becomes available, the periods of tasks that enter recovery mode are decreased and the release times of subsequent jobs are re-calculated using a set of reweighting rules (R/W in Fig. 6). In the rest of the section, we describe AUS more formally.

Definition 9. Let $u_k = e_k/p_k$ be the *utilization* of task T_k and $U_{sum} = \sum_{T_i \in \tau} u_k$ be the total utilization of all tasks. We require that $u_k \leq 1$ and $U_{sum} \leq m$ for otherwise maximum job response times can be unbounded.

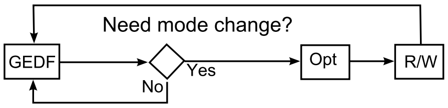

Fig. 6. High-level organization of AUS.

Definition 10. In *normal* mode, task T_k's period is $\mathsf{p}(T_k, t) = p_k$ and, in *recovery* mode, its period is $\mathsf{p}(T_k, t) = p_k^{[r]} \in [e_k, p_k]$. We assume that $\mathsf{p}(T_k, 0) = p_k$.

In normal mode, task T_k's period equals table V_k's nominal update period. If table V_k is not fresh continuously over an interval, then, in the absence of mode changes, by (4), $r_{k,j+1} = r_{k,j} + \mathsf{p}(T_k, r_{k,j}) \leq r_{k,j} + p_k$, i.e., T_k's jobs are released at least every p_k time units within that interval.

Example 8. Consider the system in Example 4 but assume that the arrival times of data files in D_2 are the same as in D_1. V_1 and V_2 are updated by two identical tasks T_1 and T_2 with the execution time for all jobs being exactly one time unit. By Def. 10, the normal task period is $p_1 = p_2 = 3$. Fig. 7 illustrates the execution of these two tasks in normal and recovery modes in separate rows. (No job executes within the interval $[5, 10)$, so the schedule is compressed to fit the column.) In the schedule shown in Fig. 7, jobs $T_{1,1}$ and $T_{2,1}$ have release times $r_{1,1} = r_{2,1} = 1$ and at time 1, tasks T_1 and T_2 use their normal periods, $\mathsf{p}(T_1, 1) = \mathsf{p}(T_2, 1) = p_1 = p_2 = 3$. Thus, by (3), $d_{1,1} = d_{2,1} = r_{1,1} + \mathsf{p}(T_1, r_{1,1}) = 1 + 3 = 4$. If data feeds did not break, then jobs of T_1 and T_2 would continue to be released every three time units. (The rest of the schedule is considered later.)

In contrast to the original definition of the adaptable sporadic task model [5], under AUS, mode changes cannot occur arbitrarily. (Additionally, the original model also allows task execution costs to be changed.) Task T_k's transitions between operational modes are governed by the following two rules.

RN: If $\mathsf{TE}(V_k, f_{k,j}) - \mathsf{F}(V_k, f_{k,j}) = 0$ where $T_{k,j}$ is some job, and task T_k is in recovery mode immediately prior to $f_{k,j}$, then T_k transitions to normal mode at time $\max(f_{k,j}, d_{k,j})$ and $r_{k,j+1}$ is set according to the second case of (4).

NR: At time t, task T_k is eligible to transition from normal to recovery mode if T_k is in normal mode immediately prior to t and $\mathsf{TE}(V_k, t) - \mathsf{F}(V_k, t) > \Delta\mathsf{F}(V_k)$. The actual transition depends on other tasks' modes. Let $\rho_{\mathbf{R}}$ and $\rho_{\mathbf{N}}$ denote the set of tasks in recovery and normal modes, respectively, immediately prior to t. Let $\rho_{\mathbf{NR}}$ and $\rho_{\mathbf{RN}}$ denote the set of tasks that transition between the respective modes at time t. These transitions can take place iff

$$\sum_{T_i \in \rho_{\mathbf{N}}} u_i + \sum_{T_i \in \rho_{\mathbf{R}}} \frac{u_i \cdot p_i}{p_i^{[r]}} + \sum_{T_i \in \rho_{\mathbf{RN}}} \left(u_i - \frac{u_i \cdot p_i}{p_i^{[r]}} \right) + \sum_{T_i \in \rho_{\mathbf{NR}}} \left(\frac{u_i \cdot p_i}{p_i^{[r]}} - u_i \right) \leq m. \quad (6)$$

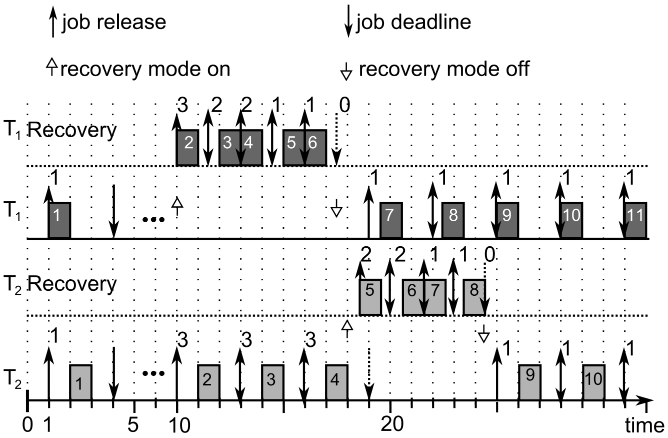

Fig. 7. Adaptive scheduling of the task system in Examples 8–10.

By Rule **RN** above, we would like to decrease T_k's share to a minimum "steady-state" value u_k to allow other tables to catch up. In Rule **NR**, $\Delta\mathsf{F}(V_k)$ is the *recovery threshold*. This rule attempts to increase the processor share of each task in $T_k \in \rho_{\mathbf{NR}}$ to $\frac{u_i \cdot p_i}{p_i^{[r]}}$ if there is an excessive amount of pending work for such tasks. However, these increases can only be performed if the resulting total processor share does not exceed m.

In Rule **RN**, we have shown how the release times are calculated when a task transitions from recovery to normal mode. The rules for calculating release times for the reverse transition are presented below and are more involved. We have tailored the reweighting rules described in [4], [5] to our needs and reduced their description to a set of seven conditions. Due to space constraints, we present only two of them below. The other five conditions can be found in [13]. Let t_c be the time instant when task T_k changes its mode from normal to recovery. If T_k does not execute at time t_c, then the release time of its next job is determined as follows. **(i):** If $T_{k,j}$ is the last-released job of T_k and $d_{k,j} \le t_c$, then $r_{k,j+1} = t_c$.

Alternatively, if job $T_{k,j}$ executes at time t_c, then the release time of its next job is determined as follows. **(ii)**: If $d_{k,j} > t_c$ and $f_{k,j} < d_{k,j}$, then $r_{k,j+1} = f_{k,j} \cdot (1 - \frac{p_k^{[r]}}{p_k}) + r_{k,j} \cdot \frac{p_k^{[r]}}{p_k} + p_k^{[r]}$.

Example 9. Consider the system in Example 8. Suppose that the recovery task period is $p_1^{[r]} = p_2^{[r]} = 1.5$, and the recovery threshold is $\Delta\mathsf{F}(V_1) = \Delta\mathsf{F}(V_2) = 8$. At time 10, data files $D_{k,2}$, $D_{k,3}$, and $D_{k,4}$ arrive for $k = 1, 2$. By Rule 2, $\mathsf{F}(V_k, 10) = \mathsf{ts}(D_{k,1}) = 1$. By Def. 6, $\mathsf{TE}(V_k, 10) = \mathsf{ts}(D_{k,4}) = 10$.

By Rule **NR**, both tasks T_1 and T_2 are in normal mode prior to time 10 and are eligible to enter recovery mode at time 10. This is because $\mathsf{TE}(V_k, 10) - \mathsf{F}(V_k, 10) = 9 > \Delta\mathsf{F}(V_k) = 8$ for $k = 1, 2$. However, only task T_1 can actually change its mode, (i.e. $\rho_{\mathbf{NR}} = \{T_1\}$) for otherwise, (6) will be violated. This is because, before the switch, $\sum_{T_i \in \rho_{\mathbf{N}}} u_i = 2/3$, $\sum_{T_i \in \rho_{\mathbf{R}}} u_i = 0$, $\sum_{T_i \in \rho_{\mathbf{RN}}} (u_i - \frac{u_i \cdot p_i}{p_i^{[r]}}) = 0$, which, by (6) (recall that $m = 1$), implies $\sum_{T_i \in \rho_{\mathbf{NR}}} \frac{u_i \cdot p_i}{p_i^{[r]}} - u_i) \le 1/3$. By (ii), the release time of $T_{1,2}$ is defined as $r_{1,2} = 10$.

Because at least one data file is pending for V_1 within $[10, 17)$, by (4), jobs of T_1 are released 1.5 time units apart as shown in Fig. 7. Table V_1 recovers because after the initial burst at time 10 only one new data file arrives every three time units but two data files are processed. Jobs of T_2 are released 3 time units apart within the interval $[10, 17.5)$ because T_2's mode did not change at time 10. Note that T_2 receives minimum processing capacity so that its backlog remains steady.

As the example above illustrates, only a limited number of tasks may be able to switch to recovery mode. These tasks can be selected using an arbitration policy. In our experiments, described in Sec. VII, we selected eligible tasks in order of increasing $\frac{u_i \cdot p_i}{p_i^{[r]}} - u_i$.

Tasks in recovery mode for which there is no pending work change their mode using Rule **RN**.

Example 10. Consider the system from Example 8 again. At time $f_{1,6} = 17$, there are no pending data files for task T_1. By Rule **RN**, T_1 changes its mode at time $d_{1,6} = 17.5$. Thus, at time 17.5, $\rho_{\mathbf{R}} = \{T_1\}$, $\rho_{\mathbf{N}} = \{T_2\}$, and $\rho_{\mathbf{RN}} = \{T_1\}$. By Rule **NR**, task T_2 enters recovery mode at time 17.5 because (6) holds if $\rho_{\mathbf{NR}} = \{T_2\}$. By Rule **RN**, the next job of T_1, $T_{1,7}$, is released at time $r_{1,7} = \min\{t \mid t \ge \max(d_{1,6}, f_{1,6}) \wedge \neg\mathsf{FRESH}(V_1, t)\} = \min\{t \mid t \ge 17.5 \wedge \neg\mathsf{FRESH}(V_1, t)\} = \mathsf{arr}(D_{1,7}) = 19$. Job $T_{2,4}$ executes at time $t_c = 17.5$ and $d_{2,4} > t_c$. Thus, by Rule **NR** and (ii), $r_{2,5} = f_{2,4} \cdot (1 - \frac{p_2^{[r]}}{p_2}) + r_{2,4} \cdot \frac{p_2^{[r]}}{p_2} + p_2^{[r]} = 18 \cdot (1 - 1.5/3) + 16 \cdot (1.5/3) + 1.5 = 17 + 1.5 = 18.5$. The intuition behind this setting of $r_{2,5}$ is as follows. Because there is excessive unprocessed backlog for V_2, we would like to increase the frequency of T_2's jobs and start their execution right away. However, the above setting of release time introduces some separation between the execution of jobs $T_{2,4}$ and $T_{2,5}$ so that T_2 does not overallocate the processor and lets other tasks to execute.

Finally, when job $T_{2,8}$ completes, there are no pending files in D_2 so T_2 changes its mode from recovery to normal using Rule **RN**. As tasks T_1 and T_2 have normal periods 3 after time 25, their jobs are released three time units apart and the system becomes fully recovered.

Selecting recovery periods. The design of AUS allows late data to be processed at a faster rate while maintaining the progress of other tables. In the above discussion, we assumed that task recovery periods were known in advance. In contrast to task normal periods, which are derived from data feed parameters and task execution times, the choice of recovery periods is a tradeoff between the recovery speed for an individual table, response-time (staleness) bounds, and the number of tables that can recover simultaneously. Though a comprehensive evaluation of these tradeoffs is beyond the scope of this paper, we give two heuristics below.

First, for each task T_i, we can set

$$p_i^{[r]} = \frac{e_i}{\max(1, m - U_{sum} + u_i)}, \tag{7}$$

which is the minimum possible value such that T_i is able to enter recovery mode and maintain (6). This selection of recovery periods maximizes the recovery rates of individual tables.

Alternatively, if we set $p_i^{[r]} = \max(e_i, \frac{p_i \cdot U_{sum}}{m})$ for each task T_i, then all tasks could enter recovery mode simultaneously. However, the recovery time for an individual table will be larger.

In the next section, we quantitatively characterize the performance of AUS. First, we apply prior results to find response-time bounds for tasks scheduled using AUS. Second, we establish table freshness (staleness) guarantees for "healthy" base and derived tables, i.e., the tables for which incoming data files arrive regularly. Finally, we determine the time needed to recover a table after an outage of one or more data feeds occurs.

IV. Calculating Response-Time Bounds

To find response-time bounds for update jobs, we first introduce the definition of deadline tardiness.

Definition 11. The *tardiness* of $T_{i,j}$ is defined as $\max(0, f_{i,j} - d_{i,j})$, where $f_{i,j}$ is $T_{i,j}$'s completion time. A *task*'s tardiness is the maximum of the tardiness of any of its jobs. We let Y_i denote a finite upper bound on task T_i's tardiness.

It has been shown by Block et. al that the maximum deadline tardiness of an adaptable sporadic task system is upper-bounded [5].

Theorem 1. (Proved in [4].) *Let τ be an adaptable sporadic task system, where for any $t \geq 0$, $\sum_{T_i \in \tau} e_i / \mathsf{p}(T_i, t) \leq m$ holds. Let $W(T_z) = e_z / \min_{t \geq 0}(\mathsf{p}(T_z, t))$. Then, for any task T_i, non-preemptive* GEDF *on m processors ensures a tardiness of at most*

$$Y_i = e_i + \frac{\sum_{T_z \in \mathcal{E}(\tau,m)} e_z - \min(e_z)}{m - \sum_{T_z \in \mathcal{X}(\tau,m-1)} W(T_z)},$$

where $\mathcal{E}(\tau, m)$ is the set of $\min(|\tau|, m)$ tasks with the largest execution times and $\mathcal{X}(\tau, m-1)$ is the set of $\min(|\tau|, m-1)$ tasks with the largest values of $W(T_z)$.

The deadline tardiness bounds given by Theorem 1 can be used to calculate response-time bounds.

Lemma 1. *(5) holds for $\Theta_k = p_k + Y_k$.*

Proof: If the tardiness bound Y_k is known for task T_k, then job $T_{k,j}$ completes within Y_k time units after its deadline. We thus have $f_{k,j} \leq d_{k,j} + Y_k \overset{\{\text{by (3)}\}}{=} r_{k,j} + \mathsf{p}(T_k, r_{k,j}) \leq r_{k,j} + p_k + Y_k$. ■

V. Staleness Bounds

In this section, we present guarantees on table staleness for tables whose source data feeds are healthy under the assumption that maximum job response times are bounded. According to the definition of table healthiness below, a table is healthy if the discrepancy between the current time and its freshness is at most some constant.

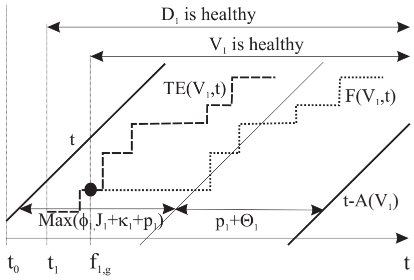

Fig. 8. Illustration of Theorem 2 in Example 11.

Definition 12. A table V_k is said to be H-healthy at time t if $\mathsf{F}(V_k, t) \geq t - H$.

Our freshness (staleness) guarantee result, formally given in Theorems 2 and 3 below, states that if table V_k is fresh at some time t_0 and its sources are healthy, then V_k is $A(V_k)$-healthy after time t_0 if the maximum job response time is bounded. (In other words, at any time $t \geq t_0$, the most recent consistent data in V_k has timestamp within $A(V_k)$ time units from current time t). The constant $A(V_k)$ is defined below.

Definition 13. Let

$$A(V_k) = \begin{cases} \Theta_k + p_k + \max(\phi_k, p_k + J_k + \kappa_k) & \text{if } V_k \text{ is base,} \\ \Theta_k + p_k + \max\limits_{V_i \in pred(V_k)} \{A(V_i)\} & \text{otherwise.} \end{cases}$$

Theorem 2. (Proved in [13].) *If table V_k is base,* $\mathsf{FRESH}(V_k, f_{k,g}) = $ *true for some $g \geq 0$, data feed D_k is healthy at all $t \in [f_{k,g}, t_e]$, and T_k's per-job response time is at most Θ_k, then V_k is $A(V_k)$-healthy and $\mathsf{S}(V_k, t) \leq A(V_k)$ for all $t \in [f_{k,g}, t_e]$.*

Theorem 3. (Proved in [13].) *If table V_k is derived,* $\mathsf{FRESH}(V_k, f_{k,g}) = $ *true for some $g \geq 0$, each source table $V_i \in pred(V_k)$ is $A(V_i)$-healthy at all $t \in [f_{k,g}, t_e]$, and T_k's per-job response time is at most Θ_k, then V_k is $A(V_k)$-healthy and $\mathsf{S}(V_k, t) \leq A(V_k)$ for all $t \in [f_{k,g}, t_e]$.*

We informally explain Theorem 2 using an example below.

Example 11. Consider a base table V_1 with source data feed D_1 that is updated by task T_1. Suppose that D_1 experiences an outage during interval $[t_0, t_1]$ and is healthy during interval $[t_1, +\infty)$ as shown in Fig. 8. It can be shown that $\mathsf{TE}(V_1, t) \leq t$ and $\mathsf{TE}(V_1, t) \geq t - \max(\phi_1, p_1 + J_1 + \kappa_1)$ for each $t \in [t_1, +\infty)$ as illustrated in Fig. 8. If table V_1 recovers at time $f_{1,g}$, then its freshness becomes equal to its trailing edge. After time $f_{1,g}$, all changes in V_1's trailing edge (which happen when new data files arrive) are reflected in V_1's freshness within at most $p_1 + \Theta_1$ time units because T_1's jobs have bounded maximum response time. Therefore, V_1's freshness becomes lower-bounded as shown in Fig. 8. Theorem 3 can be illustrated similarly.

Setting recovery thresholds $\Delta\mathsf{F}(V_k)$ as $A(V_k)$ we can ensure that task T_k does not switch to recovery mode as long as V_k remains healthy.

VI. Recovery Performance Guarantees

In this section, we calculate the time needed to recover an individual base table after an outage. We assume that no data file arrives during the interval $[t_0 - \lambda, t_0]$, where λ is sufficiently long. Additionally, we assume that all data files that arrive prior to $t_0 - \lambda$ are processed by time t_0, and at time t_0, T_k releases a job and switches to recovery mode. By Rule **RN**, task T_k switches to normal mode at or after time t_s such that $\mathsf{TE}(V_k, t_s) = \mathsf{F}(V_k, t_s)$, which for base tables implies $\mathsf{BL}(V_k, t_s) = \emptyset$. We next calculate x such that $\mathsf{BL}(V_k, t_1) = \emptyset$, where $t_1 \leq t_0 + x$ assuming that D_k is healthy after time t_0. (i.e., we want to find the latest time $t_0 + x$ when T_k has processed its entire backlog).

Theorem 4. *If, for base table V_k, task T_k switches to recovery mode at time t_0 after an outage of length λ, then it takes at most x_k time units to clear the backlog, where*

$$x_k = \frac{\Theta_k \cdot p_k + (\lambda + J_k + p_k) \cdot p_k^{[r]}}{p_k - p_k^{[r]}}.$$

The theorem can be proved by comparing the maximum number of data files that can arrive within any interval of length $\lambda + x_k$ to the minimum number of data files that can be processed within the interval $[t_0, t_0 + x_k]$.

VII. Experimental Evaluation

In this section, we report on experiments that were conducted to evaluate the performance of AUS.

Scheduling algorithms. We compared AUS to the state-of-the art PRP algorithm [8]. Under PRP, tasks with shorter update periods have higher priority, which seems like a natural prioritization for multiple streams with different update rates. Any ties are broken in favor of jobs with the largest ratio of the resulting freshness increase to the job's worst-case execution time. Additionally, in order to prevent starvation, tasks are grouped into clusters based upon their period lengths. However a task can be *promoted* if there is an available processor. Finally, under PRP, the update length is not upper-bounded and several pending updates can execute as a single job.

Task-set generation procedure. Our experiments involved running a synthetic task set on a proprietary warehouse simulator designed at AT&T. We designed our synthetic task set using a table configuration from an actual network data warehouse as a guideline. In that configuration, 10, 10, 14, and 196 tables have update periods of 300, 900, 3600, and 28800 seconds, respectively. According to system logs, most of the workload is generated by update tasks for tables with 300 and 900 seconds. In prior work, it was also noted that it might be beneficial to schedule update tasks with significantly different periods on different processor sets [8].

Taking these considerations into account, in our experiments, we considered *two* classes of tables — C_1 and C_2 — with periods 300 and 900 seconds, respectively. Both classes contained seven tables $V_1, \ldots, V_7$ and $V_8, \ldots, V_{14}$, respectively. For each table, the source data feed supplied new data files regularly with zero timestamp and arrival jitter and the data feeds for tables $V_1, \ldots, V_4$ experienced an outage from time 5,000 to time 10,000 (in seconds).

To account for execution time variability, job $T_{i,j}$'s execution time was taken uniformly at random from the interval $[(1-b) \cdot e_{i,j}, (1+b) \cdot e_{i,j})$, where $e_{i,j} = S_i + R_i \cdot \mathsf{L}_{i,j}$, $S_i = 0.01 \cdot p_i$, and $R_i = 0.1$ are execution time parameters and $b = 0.2$ is the variability. (In [8], all tasks are normalized to have $S_i = 1$ and $P_i = 100$. Also, in [8], $b = 0.5$ is used. We now have empirical evidence that suggests that a smaller value is more appropriate.) Because data files' timestamps are strictly periodic, the maximum update length for table V_k is p_k, and hence, the worst-case job execution time is $e_k = (1+b) \cdot (S_i + R_i \cdot p_k)$. Because, by (7), recovery periods may depend on worst-case task execution times, we examined two settings for recovery periods. In the first setting, we set the recovery period $p_k^{[r]}$ for each task using (7) and assuming $e_k = S_k + R_k \cdot p_k$. This resulted in $p_k^{[r]} \approx p_k/5$. We refer to this setting as the "average-case" provisioning rule because the average job execution time is $S_k + R_k \cdot p_k$. In the second setting, we set the recovery period $p_k^{[r]}$ for each task using (7) and assuming $e_k = (1+b) \cdot (S_k + R_k \cdot p_k)$ (the maximum possible job execution time). This resulted in $p_k^{[r]} \approx p_k/3$. We refer to this setting as the "worst-case" provisioning rule. We examined a processor count of $m = 2$ and simulated a schedule up to time 30,000 seconds.

Results. The goal of our experiments was to compare how AUS and PRP recover table freshness after an outage. Inset (a) of Fig. 9 shows the difference between the trailing edge and table freshness as a function of time for tables $V_1, \ldots, V_4$, which have a nominal update period of 300 seconds and experience an outage. The upper graph gives results for PRP and the lower graph for AUS. The recovery periods are set according to the average-case provisioning rule. During the interval $[0, 5000]$ new data files come regularly and the difference between the trailing edge and table freshness does not exceed the table period. During the interval $[5000, 10000]$ new data files do not arrive. At time 10,000 backed-up data files become available and the discrepancy between the trailing edge and freshness grows sharply. As can be seen in Fig. 9(a), PRP aggressively recovers all late tables simultaneously during the interval $[10000, 12500]$. In contrast, AUS recovers one late table at a time. In each graph, the descending parts of the curves correspond to time intervals where the respective update tasks execute in recovery mode. As seen, it takes AUS slightly longer to recover all late tables compared to PRP. This may make AUS seem not competitive with PRP unless we consider the effects of recovery.

Inset (b) of Fig. 9 shows the difference between the trailing edge and table freshness as a function of time for tables $V_8, \ldots, V_{11}$, which have a nominal update period of 900 seconds and do not experience an outage. Under AUS, this difference does not exceed the table period at any time. However, under PRP, for tables V_8 and V_9 the discrepancy between

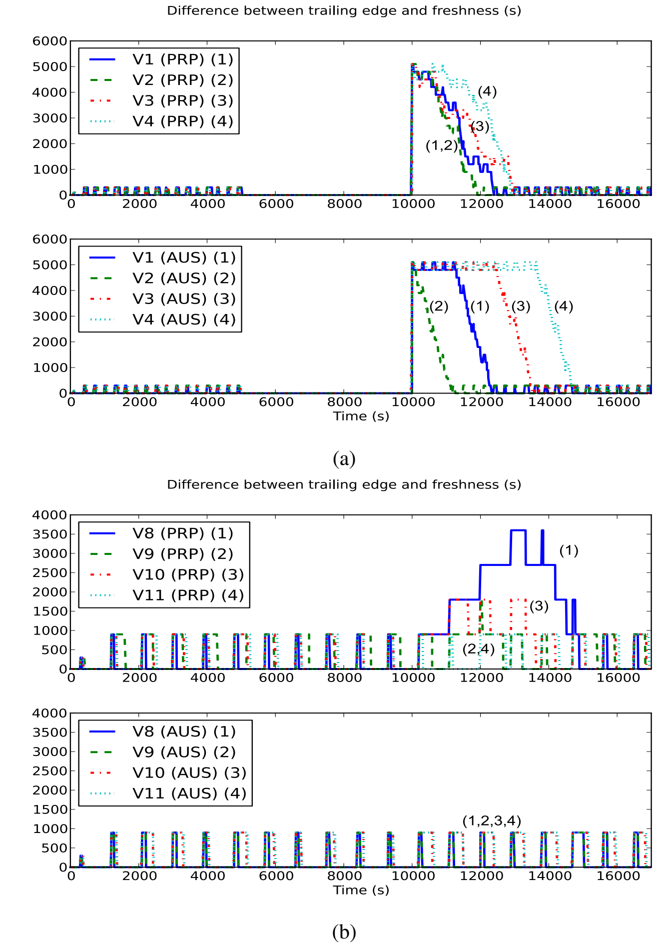

Fig. 9. The difference between trailing edge and freshness for tables with period **(a)** 300 and **(b)** 900. Recovery periods are average-case provisioned.

the trailing edge and freshness grows significantly after time 10,000. This is because PRP dedicates all available resources for recovering late tables with short periods. Moreover, the freshness of V_8 is not completely restored until time 15,000, which is about the time when AUS fully completes recovery. As mentioned in the introduction, such unpredictable disruptive behavior is not good for a data warehouse. The situation with worst-case recovery provisioning is similar except that AUS completes the recovery significantly after PRP (time 25,000 vs 15,000). This is because AUS conservatively delays updates and the available processing capacity is not fully used. As data warehouse[s] are soft-real-time applications, they are more likely to be provisioned using average-case or near-average-case execution costs. Given this, our "average-case" results are probably more meaningful. Note that, with an average-case provisioning, our analytically-derived bounds can be assumed to hold in expectation only. However, as the above experiments show, predictability is still significantly enhanced.

To summarize the above discussion, PRP recovers late tables very quickly but causes unwanted interference on tables that are not late. In contrast, due to its more conservative scheduling policy, AUS recovers late tables while maintaining data freshness of tables that are not late.

VIII. Conclusion

In this paper, we have proposed an adaptive approach to scheduling updates in data warehouses. The presented scheduling algorithm, AUS, is capable of recovering tables after feed outages while maintaining guarantees on data freshness and minimizing interference on tables that are not affected by outages. Our experimental evaluation shows that AUS is competitive with and more predictable than a state-of-the-art heuristic-based proportional scheduler.

In the future, we plan to evaluate tradeoffs pertaining to the selection of recovery periods and recovery arbitration policies depending on the relative criticality of tables. We also plan to implement AUS in DataDepot to evaluate its performance under real workloads.

References

[1] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, "Analysis of a reservation-based feedback scheduler," in *Proc. of the 23rd Real-Time Systems Symposium*, 2002, pp. 71–80.

[2] M. Ahuja, C. C. Chen, R. Gottapu, J. Hallmann, W. Hasan, R. Johnson, M. Kozyrczak, R. Pabbati, N. Pandit, S. Pokuri, and K. Uppala, "Peta-scale data warehousing at yahoo!" in *Proc. of SIGMOD*, 2009, pp. 855–862.

[3] S. Babu and J. Widom, "Continuous queries over data streams," *SIGMOD Record*, vol. 30, no. 3, pp. 109–120, 2001.

[4] A. Block, "Adaptive multiprocessor real-time systems," Ph.D. dissertation, UNC Chapel Hill, 2008.

[5] A. Block, J. Anderson, and U. Devi, "Task reweighting under global scheduling on multiprocessors," *Real-Time Systems*, vol. 39, pp. 123–167, 2008.

[6] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker, "Operator scheduling in a data stream manager," in *Proc. of VLDB*, 2003.

[7] H. Cho, B. Ravindran, H. Wu, and E. D. Jensen, "On multiprocessor utility accrual real-time scheduling with statistical timing assurances," in *Proc. of the IFIP International Conference on Embedded And Ubiquitous Computing*, August 2006, pp. 274–286.

[8] L. Golab, T. Johnson, and V. Shkapenyuk, "Scheduling updates in a real-time stream warehouse," in *Proc. of the 25th International Conference on Data Engineering*, 2009, pp. 1207–1210.

[9] L. Golab, T. Johnson, J. Spencer, and V. Shkapenyuk, "Stream warehousing with datadepot," in *Proc. of SIGMOD*, 2009, pp. 847–854.

[10] J. R. Haritsa, M. J. Carey, and M. Livny, "Value-based scheduling in real-time database systems," *VLDB Journal*, vol. 2, no. 2, pp. 117–152, 1993.

[11] A. K. Jha, M. Xiong, and K. Ramamritham, "Mutual consistency in real-time databases," in *Proc. of the 27th Real-Time Systems Symposium*, 2006, pp. 335–343.

[12] A. Karakasidis, P. Vassiliadis, and E. Pitoura, "ETL queues for active data warehousing," in *Proc. of IQIS*, 2005, pp. 28–39.

[13] H. Leontyev, T. Johnson, and J. Anderson, "An adaptive scheme for overload handling in active data warehouses," October 2009. [Online]. http://cs.unc.edu/~ anderson/papers/adaptive_warehouse.pdf

[14] P. Li, "Utility accrual real-time scheduling: Models and algorithms," Ph.D. dissertation, Virginia Polytechnic Institute and State University, July 2004.

[15] C. Lu, J. Stankovic, S. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms," *Real-Time Systems*, vol. 23, no. 2-3, pp. 85–126, 2002.

[16] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.-E. Frantzell, "Supporting streaming updates in an active data warehouse," in *Proc. of the 23rd International Conference on Data Engineering*, 2007, pp. 476–485.

[17] J. Stankovic, S. S. Hyuk, and J. Hansson, "Misconceptions about real-time databases," *IEEE Computer*, vol. 32, no. 6, pp. 29–36, 1999.

[18] M. Xiong, J. A. Stankovic, K. Ramamritham, D. Towsley, and R. Sivasankaran, "Maintaining temporal consistency: issues and algorithms," in *Proc. of International Workshop on Real-Time Database Systems*, 1996, pp. 2–7.