Predicting Maximum Data Staleness in Real-Time Warehouses*

Hennadiy Leontyev[1] Theodore Johnson[2]
James H. Anderson[1]
[1] Department of Computer Science, The University of North Carolina at Chapel Hill
[2] AT&T Labs - Research

Abstract

This paper presents an analysis technique for estimating maximum data staleness in a data warehouse that collects "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). In prior work, ad hoc heuristic algorithms have been proposed for scheduling warehouse updates. In this paper, global multiprocessor real-time scheduling algorithms are considered as an alternative. It is shown that schedulability results concerning such algorithms can be used to analytically derive upper bounds on maximum data staleness based upon characteristics of warehouse tables and the parameters of update tasks. Simulation experiments are presented that show the effectiveness of the proposed approach.

1. Introduction

In many critical business applications where integrated access to historical data as well as recent data pushed to the system in near-real-time are required, data stream warehouses (or, active data warehouses [13]) are used. For example, network data warehouses maintained by large Internet Service Providers collect various system logs, IP packet traces, and traffic summaries to monitor network performance and detect malicious users.

In prior work on implementing data warehouses (which is described in greater detail below), heuristic-oriented scheduling algorithms have been used. In this paper, we consider the viability of multiprocessor real-time scheduling algorithms as an alternative. Research on these algorithms has matured significantly in recent years. We show that prior work on such algorithms can be leveraged to develop data warehouse schedulers that not only can provide comparable performance to prior algorithms, but also allow *guarantees* on data freshness to be made.

We assume that the warehouse consists of a hierarchy of base and derived tables (or materialized views) containing timestamped data records. These records are organized as table rows with columns representing record fields as shown in Figure 1(a). To add new data to the warehouse, maintenance programs called *update tasks* are invoked periodically. Each such task reads data either from an external feed or from a set of source warehouse tables and appends new rows to a target table via an *Extract-Transform-Load* (ETL) process [10, 13]. Once the data has been extracted from its sources and loaded into a table, the transformation process can largely be done via a collection of queries implemented via materialized views. These queries perform tasks such as transforming field values into standard formats, normalization, deduplicatation, and other types of data cleaning. Figure 1(b) shows base tables V_1, V_2, and V_3 and a derived table V_4 and their respective update tasks.

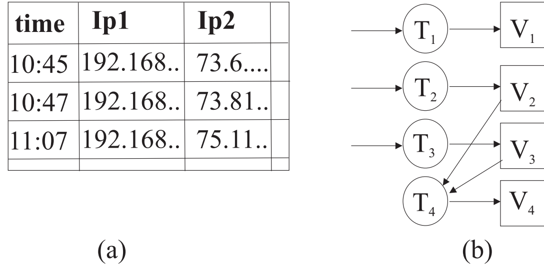
Ensuring the freshness of stored data is critical in many large enterprise systems. For example, service disruptions are considered to be a serious problem within AT&T's network. They are to be avoided if possible, and when they occur, their duration must be minimized. Since a network outage can be caused by many problems, diagnosing the source cause requires data from network alerts, router performance logs, routing update protocol logs, and so on. The data in each of these tables must be as fresh as possible so that network technicians can restore service as quickly as possible.

Given a warehouse configuration as described above, we analytically derive *a priori guarantees* on the quality of data in the warehouse expressed as upper bounds on *table staleness* (formally defined in Section 2), which is the discrepancy between the current time and the maximum timestamp of a record uploaded in the table so far. These upper bounds depend on the table configuration, the parameters of update tasks, and the scheduling algorithm that selects which table to update next.

The following warehouse maintenance policies have been proposed in prior work [4].

- *Deferred view maintenance*, under which the tables are

*Work supported by AT&T, IBM, Intel, and Sun Corps., NSF grants CNS 0834270, CNS 0834132, and CNS 0615197, and ARO grant W911NF-06-1-0425.



time	Ip1	Ip2	
10:45	192.168..	73.6....	
10:47	192.168..	73.81..	
11:07	192.168..	75.11..	

Figure 1. Example warehouse structure.

updated at the time of a query. This policy suffers from high query response times, especially if large amounts of streaming data arrive between query invocations.

- *Periodic view maintenance*, under which all updates to the tables are applied periodically as a batch, e.g., once a day. Unfortunately, this can delay the installation of updates if new data arrives in the middle of the update period.

- *Immediate view maintenance*, under which base table(s) are immediately updated when new data arrives on the corresponding stream.

Even though immediate view maintenance may appear to be a reasonable solution for a real-time stream warehouse, non-trivial problems may arise when handling multiple streams. In particular, the completion of an update on a base table V triggers updates of views that propagate throughout the entire table hierarchy. (In the example warehouse shown in Figure 1, an update of V_2 or V_3 would immediately invoke task T_4.) If the number of update tasks running in parallel is not limited, then performance can degrade severely due to thrashing and context switching. (Also, multiple memory-intensive ETL processes are likely to exhaust virtual memory.) On the other hand, restricting concurrency can unnecessarily delay some updates. This motivates the need for a scheduler that determines which task (i.e., table) to schedule (i.e., update) next.

Warehouse scheduler design considerations. A real-time warehouse scheduler must simultaneously pursue multiple (often conflicting) goals. First, the ultimate goal is to ensure that queries on the warehouse see data that is as up-to-date as possible. Second, the scheduler must maintain data consistency in the sense that a derived table must be equivalent to running its defining query over the state of its source table(s) at some point in the past or present [4]. Third, the scheduler must handle heterogeneous task sets, as different streams may have different data rates and interarrival times; the main challenge here is to prevent short-lived updates from being blocked by long-running updates. Fourth, the scheduler must support multiprocessor machines in order to accommodate high update and query rates. Finally, it is important to establish guarantees on the performance of the warehouse before it is deployed.

Prior work. Some heuristic-based warehouse schedulers pursuing the design considerations listed above have been proposed recently [2, 6]. In [2], the authors consider aggregate staleness and establish a competitive ratio with respect to a clairvoyant optimal scheduler; however, they do not consider table hierarchies. In [6], a simple warehouse model is introduced and a "proportional" (PRP) heuristic is proposed. As its name suggests, PRP attempts to balance the allocations of update tasks so that starvation is minimized. This heuristic is essentially a hybrid of static-priority scheduling with utility-based scheduling. PRP is currently the state-of-the-art algorithm for scheduling updates in hierarchically organized relational data warehouses. Unfortunately, the heuristic nature of PRP makes the derivation of analytical results involving real-time correctness difficult, if not impossible.

Contributions. In this paper, we show that it is possible to schedule warehouse workloads using algorithms that allow real-time guarantees to be made with performance comparable to PRP. We first present simple formulas for calculating upper bounds on maximum data staleness based upon update task parameters and table hierarchies assuming the warehouse model introduced in [6] (the few differences are discussed in detail later in Section 2). Second, we apply the developed analysis to several well-studied real-time schedulers and propose a novel variant of the non-preemptive earliest-deadline-first algorithm, which is described in detail in Section 4.2. This new algorithm allows maximum staleness bounds to be reduced by grouping update tasks with similar execution demands. Third, we present an experimental evaluation that shows that simple predictable real-time schedulers are competitive with sophisticated heuristics, with the added benefit of providing *guarantees*. In this evaluation, a synthetic task set was run on a proprietary warehouse simulator designed at AT&T.

The notion of staleness in this paper is somewhat similar to prior work on ensuring the "temporal consistency" of 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 [14, 15]. In prior work, these validity thresholds have been generally used for determining priorities and timing constraints of update tasks. 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 the timing requirements of update tasks to be relaxed so that, even though it is desirable to perform updates as quickly as possible, some delays can be tolerated.

To deal with these kinds of relaxed requirements, we heavily use prior results on scheduling periodic tasks on

multiprocessor platforms [7, 5]. We specifically leverage the fact that a large class of *global* multiprocessor schedulers can ensure bounded maximum job response times [11]. (Under global scheduling, processors independently take jobs from a shared job queue.) This allows the rate of update job completions to be kept equal to the rate of update arrivals.

The rest of the paper is organized as follows. In Section 2, we introduce our system model, formalize the notion of data freshness and staleness, and explain how to maintain data consistency in the presence of asynchronous updates. In Section 3, we derive a general upper bound on staleness. In Section 4, we compute staleness bounds under some well-studied real-time schedulers. In Section 5, we present an experimental evaluation of our approach. Section 6 concludes.

2. System Model

The warehouse model studied in this paper is very similar to that considered in prior work [2, 6]. We study a relational data warehouse that collects append-only data streams, with new data arriving periodically on each data stream at a specified rate. This data is uploaded to warehouse tables as timestamped data records. A real-time stream warehouse maintains two types of tables: *base* tables that are sourced directly from data streams, and *derived* tables (materialized views) that are sourced from (i.e., defined as the result of an SQL query over) one or more base or other derived tables. We assume that relationships among source and derived tables form a directed acyclic graph. 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, $pred(V_k) = \emptyset$.

Task model. Each table V_k is updated by an external program referred to as *task* T_k. Each task is invoked repeatedly every p_k time units; each such invocation is called a *job*. The jth job of T_i is denoted $T_{k,j}$ and its invocation (or *release*) time is denoted $r_{k,j}$. We assume that $r_{k,j+1} = r_{k,j} + p_i$, where p_k is the *period* of task T_k, and

$$r_{k,1} = \phi_k, \tag{1}$$

where $\phi_k \geq 0$ is T_k's phase. We let $s_{k,j} \geq r_{k,j}$ denote the time when job $T_{k,j}$ is scheduled. 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. The completion time of $T_{k,j}$ is denoted $f_{k,j}$.

Table freshness and data consistency. We next define table freshness and discuss the notion of data consistency adopted in this paper.

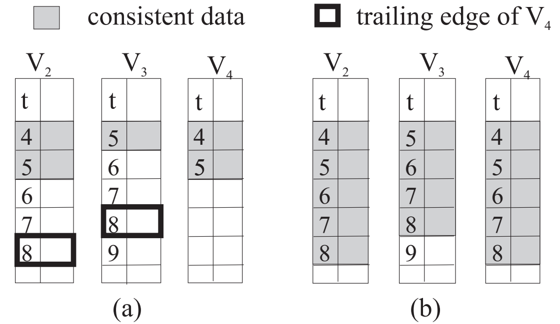


Figure 2. State of tables at times **(a)** 12 and **(b)** 14 in Example 1.

Definition 1. [6] *The freshness* of table V_k at time t, denoted as $\mathsf{F}(V_k, t)$, is the maximum timestamp of a record stored in V_k by time t.

Definition 2. [6] *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.

Update jobs cause table freshness (and hence, staleness) to change. For base tables, this is straightforward. When an update job $T_{k,j}$ of base table V_k starts to execute at time t, it loads all records with timestamps within the range $(\mathsf{F}(V_k, t), t]$ into V_k. (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 $\mathsf{F}(V_k, f_{k,j}) = t$.

Updating freshness for derived tables is more complicated due to dependencies among the tables. In this paper, we follow the definition of "trailing edge consistency."

Definition 3. [6] We define the *trailing edge* of table V_k at time t as

$$\mathsf{TE}(V_k, t) = \begin{cases} t & \text{if } V_k \text{ is base,} \\ \min_{V_i \in pred(V_k)} \{\mathsf{F}(V_i, t)\} & \text{otherwise.} \end{cases}$$

Example 1. Suppose that at time 12 the derived table V_4 in Figure 1(b) 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 Figure 2(a). Table V_4 is consistent with respect to the state of its sources as of time 5. When an update job of V_4 commences execution at time 12 it needs to add new records to V_4 to reflect the most recent information in its source tables. At time 12, both source tables have the 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, the update job 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. When the update finishes at time 14, 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 Figure 2(b).

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

Definition 4. Let $\mathsf{L}_{k,j} = \min(\mathsf{TE}(V_k, s_{k,j}) - \mathsf{F}(V_k, s_{k,j}), p_k)$ be the *update length* of $T_{k,j}$. $\mathsf{L}_{k,j}$ is the amount by which table freshness increases when $T_{k,j}$ finishes.

In contrast to the original definition in [6], we limit the update length by T_k's period in the above definition in order to achieve a predictable worst-case execution time for $T_{k,j}$, which is proportional to $\mathsf{L}_{k,j}$, as explained later in Section 4.

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 we define $f_{k,0} = 0$ and $\mathsf{F}(V_k, t) = \mathsf{F}(V_k, f_{k,0}) = 0$.)

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 2. Figure 3(a) shows $\mathsf{S}(V_1, t)$, where V_1 is a base table that is updated by an update task with $p_1 = 5$. At time zero, $\mathsf{S}(V, 0) = 0$. As shown in Figure 3(b), suppose that the first update job $T_{1,1}$, released at time $r_{1,1} = 3$, is scheduled at time $s_{1,1} = 3$ and completes at time $f_{1,1} = 5$. Also, suppose that the second update job $T_{1,2}$, released at time $r_{1,2} = 8$, is scheduled at time $s_{1,2} = 8$ and completes at time $f_{1,2} = 10$. To see that staleness changes as in inset (a), consider first the interval $[0, 5)$. Since no job completes within this interval, by Rule 2, $F(V_1, t) = 0$, and hence, by Definition 2, $\mathsf{S}(V_1, t)$ increases linearly. At time 5, job $T_{1,1}$ completes. By Definition 4, its update length is $\mathsf{L}_{1,1} = \min(\mathsf{TE}(V_1, s_{1,1}) - \mathsf{F}(V_1, s_{1,1}), p_1)$, which by Definition 3 and Rule 2 implies $\mathsf{L}_{1,1} = \min(3 - 0, 5) = 3$. Therefore, $T_{1,1}$ loads all data with timestamps from 0 to 3 into V_1 and hence, by Rule 3, $\mathsf{F}(V_1, f_{1,1}) = \mathsf{F}(V_1, s_{1,1}) + \mathsf{L}_{1,1} = 0 + 3$. We thus have $\mathsf{S}(V_1, f_{1,1}) = 5 - 3 = 2$. Consider now the interval $[5, 10)$. By Rule 2, $\mathsf{F}(V_1, t) = \mathsf{F}(V_1, f_{1,1}) = 3$ for each $t \in [5, 10)$. By Definition 2, $\mathsf{S}(V_1, t) = \max(0, t - \mathsf{F}(V_1, t)) = t - 3$, so staleness increases linearly during $[5, 10)$.

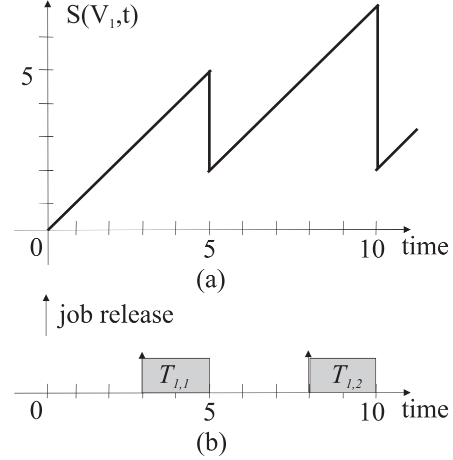


Figure 3. (a) Staleness of base table V_1 and (b) the schedule of update jobs in Example 2.

Response time. 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 $\max_j(f_{k,j} - r_{k,j})$. It has been shown that certain global multiprocessor scheduling algorithms can ensure bounded response times for any task set where jobs are released periodically (provided that the system is not overutilized) [5, 11]. In the next section, we derive table staleness guarantees assuming that inequality (2) below holds for each job $T_{k,j}$.

$$f_{k,j} \leq r_{k,j} + \Theta_k \tag{2}$$

In (2), Θ_k is an upper bound on the maximum response time of T_k. In Section 4, we briefly describe how to compute response-time bounds for several schedulers.

The main technical contribution of the paper is the following. Given a set of tables and materialized views $\{V_1, \ldots, V_n\}$, the parameters of update tasks $\{T_1, \ldots, T_n\}$, and maximum task response-time bounds $\{\Theta_1, \ldots, \Theta_n\}$, we establish analytical bounds on maximum table staleness for each table V_k. This allows guarantees on data freshness to be made.

3. Computing Maximum Staleness

In the remainder of this section, we show that $A(V_k)$ defined below upper bounds table V_k's staleness. To show this, we first establish lower bounds on table freshness for base and derived tables. Then, using the lower bound for an individual table, we find an upper bound on its staleness.

Definition 5. Let $A(V_k) = \Theta_k + \max(p_k, \phi_k) + \max_{V_i \in pred(V_k)} \{A(V_i)\}$.

We begin by proving some auxiliary claims and lemmas. For convenience, we rewrite Rule 3 as

$$\mathsf{F}(V_k, f_{k,j}) = \min(\mathsf{TE}(V_k, s_{k,j}), \mathsf{F}(V_k, s_{k,j}) + p_k). \quad (3)$$

Lemma 1. *If V_k is a base table, then $\mathsf{F}(V_k, f_{k,j}) \geq r_{k,j} - \max(0, \phi_k - p_k)$.*

Proof. We prove this lemma by induction on the job index j. By (3), we have

$$\begin{aligned}
&\mathsf{F}(V_k, f_{k,j+1}) \\
&\quad= \min(\mathsf{TE}(V_k, s_{k,j+1}), \mathsf{F}(V_k, s_{k,j+1}) + p_k) \\
&\qquad \left\{ \begin{array}{l} \text{by Definition 3, for base tables,} \\ \mathsf{TE}(V_k, s_{k,j+1}) = s_{k,j+1} \geq r_{k,j+1} \end{array} \right\} \\
&\quad\geq \min(r_{k,j+1}, \mathsf{F}(V_k, s_{k,j+1}) + p_k) \\
&\qquad \{\text{by Rule 2, } \mathsf{F}(V_k, s_{k,j+1}) = \mathsf{F}(V_k, f_{k,j})\} \\
&\quad= \min(r_{k,j+1}, \mathsf{F}(V_k, f_{k,j}) + p_k). \qquad (4)
\end{aligned}$$

If $j = 0$, then, because, by Rule 2, $\mathsf{F}(V_k, f_{k,0}) = 0$, (4) implies $\mathsf{F}(V_k, f_{k,1}) = \min(r_{k,1}, p_k) = r_{k,1} + \min(0, p_k - r_{k,1}) = r_{k,1} - \max(0, r_{k,1} - p_k) = r_{k,1} - \max(0, \phi_k - p_k)$, where the last equality follows from (1). Alternatively, if $j \geq 1$, then, by (4) and the induction hypothesis, $\mathsf{F}(V_k, f_{k,j+1}) \geq \min(r_{k,j+1}, r_{k,j} - \max(0, \phi_k - p_k) + p_k) = \min(r_{k,j+1}, r_{k,j+1} - \max(0, \phi_k - p_k)) = r_{k,j+1} - \max(0, \phi_k - p_k)$. □

Definition 6. Let $Z_{i,k,j} = \max(0, r_{k,j} - \max(0, \phi_k - p_k) - A(V_i))$.

The following lemma is a counterpart of Lemma 1 for derived tables. Its proof can be found in an appendix.

Lemma 2. *Let V_k be a derived table. If $\mathsf{F}(V_i, t) \geq \max(0, t - A(V_i))$ for all $V_i \in pred(V_k)$ and $t \geq 0$, then $\mathsf{F}(V_k, f_{k,j}) \geq \min_{V_i \in pred(V_k)}\{Z_{i,k,j}\}$ for all $j \geq 1$.*

Lemma 3 below gives a lower bound on table freshness provided that jobs of update tasks have bounded response times.

Lemma 3. *If the response time of each job of task T_i is at most Θ_i, then $\mathsf{F}(V_k, t) \geq \max(0, t - A(V_k))$.*

Proof. To begin, note that if $t \leq A(V_k)$, then the required result follows because, by Rule2, $\mathsf{F}(V_k, t) \geq 0$. In the remainder of the proof, we consider the more interesting case, $t > A(V_k)$. By Definition 5,

$$t > A(V_k) \geq \max(p_k, \phi_k) + \Theta_k. \quad (5)$$

From (2), we have $f_{k,1} \leq r_{k,1} + \Theta_k \overset{\{\text{by (1)}\}}{=} \phi_k + \Theta_k \leq t$, where the last inequality follows from (5). Therefore, job $T_{k,1}$ completes by time t. Let $T_{k,h}$ be the latest job of T_k to complete by t. ($T_{k,h}$ is well-defined since $T_{k,1}$ completes by t.) We thus have $f_{k,h+1} > t$, which, by (2), implies $r_{k,h+1} + \Theta_k > t$, and hence,

$$\begin{aligned}
&r_{k,h} \\
&\quad= r_{k,h+1} - p_k \\
&\quad> t - \Theta_k - p_k. \qquad (6)
\end{aligned}$$

Because the dependencies among tables form a directed acyclic graph, we complete the proof via a structural induction over the topological ordering of tables with V_k being a base table in the base case.

Base case: *V_k is a base table***.** We lower-bound $\mathsf{F}(V_k, t)$ as follows.

$$\begin{aligned}
&\mathsf{F}(V_k, t) \\
&\qquad \{\text{by Rule 2}\} \\
&\quad= \mathsf{F}(V_k, f_{k,h}) \\
&\qquad \{\text{by Lemma 1}\} \\
&\quad\geq r_{k,h} - \max(0, \phi_k - p_k) \\
&\qquad \{\text{by (6)}\} \\
&\quad> t - \Theta_k - p_k - \max(0, \phi_k - p_k) \\
&\quad= t - \Theta_k - \max(p_k, \phi_k) \\
&\qquad \{\text{by Definition 5 (note that } pred(V_k) = \emptyset)\} \\
&\quad= t - A(V_k)
\end{aligned}$$

Induction step: *V_k is a derived table***.** In this case, $pred(V_k) \neq \emptyset$. By the induction hypothesis, the conclusion of the lemma holds for each $V_i \in pred(V_k)$. We lower-bound $\mathsf{F}(V_k, t)$ as follows.

$$\begin{aligned}
&\mathsf{F}(V_k, t) \\
&\quad \{\text{by Rule 2}\} \\
&= \mathsf{F}(V_k, f_{k,h}) \\
&\quad \{\text{by the induction hypothesis and Lemma 2 }\} \\
&\geq \min_{V_i \in pred(V_k)} \{Z_{i,k,h}\} \\
&\quad \{\text{by Definition 6}\} \\
&= \min_{V_i \in pred(V_k)} \{\max(0, r_{k,h} - \max(0, \phi_k - p_k) - A(V_i))\} \\
&\quad \{\text{by (6)}\} \\
&> \min_{V_i \in pred(V_k)} \{\max(0, t - \Theta_k - p_k - \max(0, \phi_k - p_k) \\
&\qquad\qquad - A(V_i))\} \\
&= \min_{V_i \in pred(V_k)} \{\max(0, t - \Theta_k - \max(p_k, \phi_k) - A(V_i))\} \\
&\qquad (7)
\end{aligned}$$

Because $t > A(V_k)$, by Definition 5, we have $t > \Theta_k + \max(p_k, \phi_k) + \max_{V_i \in pred(V_k)}\{A(V_i)\}$, and hence,

$\max(0, t - \Theta_k - \max(p_k, \phi_k) - A(V_i)) = t - \Theta_k - \max(p_k, \phi_k) - A(V_i)$. From this and (7), we have

$$\begin{aligned} &\mathsf{F}(V_k, t) \\ &> t - \Theta_k - \max(p_k, \phi_k) - \max_{V_i \in pred(V_k)} \{A(V_i)\} \\ &\quad \{\text{by Definition 5}\} \\ &= t - A(V_k). \end{aligned}$$

□

Finally, we find an upper bound on maximum table staleness.

Theorem 1. $\mathsf{S}(V_k, t) \leq A(V_k)$.

Proof. We upper-bound table staleness as follows.

$$\begin{aligned} &\mathsf{S}(V_k, t) \\ &\quad \{\text{by Definition 2}\} \\ &= t - \mathsf{F}(V_k, t) \\ &\quad \{\text{by Lemma 3}\} \\ &\leq t - \max(0, t - A(V_k)) \\ &= \min(t, t - t + A(V_k)) \\ &\leq A(V_k) \end{aligned}$$

□

It follows from Theorem 1 and Definition 5 that in order to minimize the bound on V_k's staleness, we need to minimize the response-time bound Θ_k of the respective update task T_k as well as the response times of the update tasks of V_k's source tables.

4. Bounding Response Times

In the previous sections, we did not make any assumptions about the scheduling algorithm except that update jobs of the same task are required to execute sequentially and individual task response-time bounds are known.

In an actual data warehouse, job response-time bounds may be extremely complicated to determine precisely due to the fact that each update job requires a number of various resources such as CPUs, I/O bus(es), disk arms, fiber channel bandwidth, etc. Individual queries can exhibit very heterogeneous behavior. For example, some basic **"select *"** queries are disk-bandwidth-intensive, nested-loop joins are disk arm intensive, hash joins are CPU-intensive, etc. [8]. Because update tasks access physical system resources in some unknown and unpredictable fashion, we assume a simpler system model, which is the same as that assumed in the development of the state-of-the-art PRP algorithm [6]. In this model, update jobs are scheduled using m identical resources or *tracks*. Update jobs are assigned to tracks, so that the track becomes unavailable until its assigned job completes. Therefore, the adopted model is simply multiprocessor non-preemptive scheduling. In the rest of the paper, we treat tracks as processors on which tasks execute. However, it should be noted that, in reality, tracks do not represent particular physical resources. For example, if an update job $T_{i,j}$ leaves a CPU unused while performing a disk access, then a different job cannot be assigned to $T_{i,j}$'s track and be scheduled.

As in [6], we assume that the execution time of a single update task invocation is a linear function of the update length.

Definition 7. **[6]** We assume that $T_{k,j}$'s *execution time*, $e_{k,j}$, is at most

$$S_k + R_k \cdot \mathsf{L}_{k,j}, \tag{8}$$

where $S_k \geq 0$ and $R_k \leq 1$. Because, by Definition 4, $\mathsf{L}_{k,j} \leq p_k$, we define $e_k = S_k + R_k \cdot p_k$ to be the *worst-case execution time* for T_k and assume that $e_k \leq p_k$.

The task model described above is a subcase of the widely-studied periodic task model wherein each task T_i is characterized by its worst-case execution time e_i and job inter-arrival time $p_i \geq e_i$. Under the periodic task model with implicit deadlines, the absolute deadline of job $T_{i,j}$ is $d_{i,j} = r_{i,j} + p_i$, where $r_{i,j}$ is the release time $T_{i,j}$. The *utilization of task* T_i is defined as $u_i = e_i/p_i$, and the *utilization of the task system* τ as $U_{sum} = \sum_{T_i \in \tau} u_i$. We assume

$$U_{sum} \leq m. \tag{9}$$

Definition 8. 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 of task T_i's tardiness.

In recent work, it has been shown that certain classes of schedulers can ensure bounded deadline tardiness for periodic tasks provided that (9) holds, i.e., there exists a constant Y_i as defined in Definition 8 for each task T_i [5, 11]. Closed-form expressions for Y_i have also been derived.

If the tardiness bound Y_i is known for task T_i, then job $T_{i,j}$ completes within Y_i time units after its deadline, which is $r_{i,j} + p_i$. Thus, the response time of $T_{i,j}$ is at most $p_i + Y_i$. From this, Lemma 4 below follows.

Lemma 4. *(2) holds for* $\Theta_i = p_i + Y_i$.

Our remaining proof obligation is to determine Y_i from Definition 8. In the remainder of this section, we show how to determine Y_i under the conventional non-preemptive global earliest-deadline-first (NP-GEDF) scheduling algorithm, and a novel variation of it that improves the maximum deadline tardiness bound.

4.1. NP-GEDF

Under NP-GEDF, job $T_{a,b}$ has higher priority than job $T_{k,j}$ iff $d_{a,b} < d_{k,j}$ or $(d_{a,b} = d_{k,j}) \wedge a < k$. It has been

shown that NP-GEDF can ensure bounded deadline tardiness for all periodic task sets with implicit deadlines if (9) holds for $m \geq 2$.

Theorem 2. (Proved in [5]) *If $m \geq 2$ and (9) holds, then under* NP-GEDF*, $Y_i = e_i + x(\tau, m)$, where $x(\tau, m) \geq 0$. (The expression for $x(\tau, m)$ can be found in [5].)*

Note that if $|\tau| \leq m$ or a hard real-time schedulability test for NP-GEDF such as in [3, 12] passes, then $Y_i = 0$.

4.2. Clustered NP-GEDF

According to [5], $x(\tau, m)$ in Theorem 2 may be large if task execution times are large. Intuitively, non-preemptive low-priority jobs may occupy processors for long durations of time thereby blocking high-priority jobs. Tardiness bounds can be improved by partitioning τ into *clusters* $\{C_1, \ldots, C_K\}$ with similar execution times, and scheduling tasks in different clusters on non-intersecting sets of processors using NP-GEDF. We call this approach *clustered* NP-GEDF or C-NP-GEDF.

The task clusters must respect the following constraints. First, by (9), each cluster C_k requires at least $m(C_k) = \lceil \sum_{T_i \in C_k} u_i \rceil$ processors. Second, the maximum tardiness bound for each task in C_k can be easily computed using Theorem 2 if $m(C_k) \geq 2$ holds. Alternatively, if $m(C_k) = 1$, then we can apply a hard real-time schedulability test to the tasks in C_k to determine whether each task has zero tardiness [9]. If such a test does not hold, then we set $Y_i = \max(e_i) - \min(e_i) + e_i$ for each task T_i, which is a bound that follows from results in [12].

Finally, in order to accommodate all clusters using m processors, we require

$$\sum_{k=1}^{K} m(C_k) \leq m. \tag{10}$$

Figure 4 shows an algorithm for partitioning a task set τ into a set of clusters satisfying (10). The procedure starts with m clusters. The well-known K-means heuristic [1] is then used to group tasks based upon their worst-case execution times. First, the execution times of the K tasks are initially selected as cluster centers using a randomized procedure (line 2). Second, each cluster C_i is populated with tasks that have an execution time closest to the center c_i (lines 4–6). Third, new cluster centers are determined (lines 6–8). This procedure is then repeated until the set of cluster centers does not change. Finally, after the clusters are found, if (10) holds (line 11), then the procedure returns. Otherwise, the tentative number of clusters is decreased. The algorithm is guaranteed to finish and produce a valid set of clusters because, by (9), the tasks in τ can be grouped into one cluster satisfying (10).

```
CONSTRUCT-CLUSTERS(τ)
1   for K = m to 1 do
2     Choose K initial centers 𝒞 = {c_1, ..., c_K}
      as described in [1];
3     while 𝒞 changes do
4         for i = 1 to K do
5           Set the cluster C_i to be the set of tasks in τ
            for which |e_i − c_i| < |e_i − c_j| for each j ≠ i
6         od;
7         for i = 1 to K do
8           c_i = (Σ_{T_i ∈ C_i} e_i) / |C_i|
9         od;
10    od
11    If (10) holds, then return {C_1, ..., C_K}
12  od
```

Figure 4. Cluster construction procedure.

Example 3. To illustrate the clustering procedure, consider a task set $T_1, \ldots, T_6$ with worst-case task execution times $1, 1, 4, 5, 7$, and 10, respectively. Suppose that $K = 3$ and the initial set of centers is $\mathcal{C} = \{5, 7, 10\}$. The clusters C_1, C_2, and C_3 and their centers for different iterations are shown in Table 1. The column "$\mathcal{C}$" represents the set $\mathcal{C}$ before the program enters the loop at line 4 in Figure 4. The "C_i" columns represent the respective clusters C_i before the program enters the loop at line 7. Each row corresponds to an iteration of the loop in lines 3–10. Note that during the first iteration, task T_4 is moved from cluster C_1 to cluster C_2 because the difference between $e_4 = 5$ and $c_2 = 7$ is smaller than that between e_4 and the center of cluster C_1, $c_1 = 2.75$. After that, the set of cluster centers does not change.

5. Experimental Evaluation

In this section, we report on experiments that were conducted to evaluate the performance of several real-time scheduling algorithms in a real-time warehouse context. We compared their performance to the state-of-the-art PRP algorithm and assessed the accuracy of their guaranteed staleness bounds derived as described in Sections 3 and 4.

Scheduling algorithms. We examined the NP-GEDF and C-NP-GEDF schedulers described in Section 4, the global Rate-Monotonic (RM) scheduler, and the state-of-the art

Table 1. Illustration of Example 3.

$\mathcal{C}$	C_1	C_2	C_3
$\{5, 7, 10\}$	$\{T_1, T_2, T_3, T_4\}$	$\{T_5\}$	$\{T_6\}$
$\{2.75, 7, 10\}$	$\{T_1, T_2, T_3\}$	$\{T_4, T_5\}$	$\{T_6\}$
$\{2, 6, 10\}$	$\{T_1, T_2, T_3\}$	$\{T_4, T_5\}$	$\{T_6\}$
$\{2, 6, 10\}$	$\{T_1, T_2, T_3\}$	$\{T_4, T_5\}$	$\{T_6\}$

PRP algorithm [6]. Under RM, tasks with shorter update periods have higher priority, which seems like a natural prioritization for multiple streams with different update rates. PRP is a modification of RM with the following features. First, tasks with shorter periods have higher priority, with ties broken in favor of jobs with the largest ratio of the resulting freshness increase to the job's worst-case execution time. Second, in order to prevent starvation, tasks are grouped into clusters based upon their period lengths but, in contrast to C-NP-GEDF, a task can be *promoted* if there is an available processor. Third, 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, 14, and 196 tables have update periods of 900, 3600, and 28800 seconds, respectively. Because there is currently an interest in supporting warehouse tables with five minute update periods, we created *four* classes of update tasks — C_1, C_2, C_3, and C_4 — with periods 300, 900, 3600, and 28800 seconds, respectively. Note that the periods are multiples of each other, and the period 28800 corresponds to update tasks running on an eight-hour basis. The execution time parameters S_i and R_i (see Definition 7) were taken from [6]. Particularly, we set $S_i = 0.01 \cdot P_i$ and $R_i = 0.1$ for all tasks. (In [6], all tasks are normalized to have $S_i = 1$ and $P_i = 100$.) 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 $e_{i,j} = S_i + R_i \cdot \mathsf{L}_{i,j}$, and $b = 0.2$ is the variability. (In [6], $b = 0.5$ is used. We now have empirical evidence that suggests that a smaller value is more appropriate.) Note that, by Definition 7, the worst-case job execution time in our model becomes $e_i = (1+b) \cdot (S_i + R_i \cdot p_i)$ and the utilization becomes $(1+b) \cdot (S_i + R_i \cdot p_i)/p_i$, respectively.

We examined processor counts of $m = 4, 8, 16, 24$, and 32. For each m, we added tasks to the task set as follows. First, we set the utilization fill level $U_f = m/(1+b)$ to cap the total task utilization and account for execution time variability. Second, we capped the total utilization of tasks in classes C_1, C_2, C_3 and C_4, respectively, by 0.1, 0.1, 0.1, and 0.7 times U_f and added the corresponding number of tasks to each class. We chose these caps in order to resemble an actual warehouse configuration. Each generated task set τ thus contained several dozen tasks. For each m, we simulated a schedule for a total of 2,000,000 scheduling events.

Evaluation metrics. In our experiments, we considered two metrics. First, to assess the performance of various schedulers, we measured the weighted sum of the maximum table staleness observed during the simulation on m processors, denoted as

$$\mathsf{MWS}(m) = \sum_{T_i \in \tau} \frac{\max_{t \geq 0}(\mathsf{S}(V_i, t))}{p_i}. \tag{11}$$

We used reciprocal periods as weights because data in tables with short update periods must be as fresh as possible. For example, a ten-minute staleness increase is not critical for a table updated daily; however, it may be critical for a table to be updated every five minutes.

Second, we computed the maximum weighted staleness bound

$$\widehat{\mathsf{MWS}}(m) = \sum_{T_i \in \tau} \frac{A(V_i)}{p_i},$$

where $A(V_i)$ is defined in Definition 5. By Theorem 1 and (11), $\widehat{\mathsf{MWS}}(m)$ upper-bounds $\mathsf{MWS}(m)$. Thus, comparing $\widehat{\mathsf{MWS}}(m)$ to $\mathsf{MWS}(m)$ can help in assessing the pessimism of the analysis presented in Sections 3 and 4. To do that, we computed the ratio $\frac{\widehat{\mathsf{MWS}}(m) - \mathsf{MWS}(m)}{\mathsf{MWS}(m)}$ for NP-GEDF and C-NP-GEDF and each processor count m. (Note that, the smaller this ratio is, the better the analysis.)

Results. Inset (a) of Figure 5 shows the ratio $\mathsf{MWS}(m)/\mathsf{MWS}(|\tau|)$ for $m = 4$, 8, 16, 24, and 32 processors. $\mathsf{MWS}(|\tau|)$ is the maximum observed staleness in the contention-free case, in which each task is running on a dedicated processor. This figure shows that the performance of C-NP-GEDF is better than that of PRP for all values of m except $m = 8$. As m increases, clusters containing tasks with similar periods (execution times) as constructed by C-NP-GEDF can be mapped onto non-intersecting processor sets. This, in turn, reduces the starvation experienced by tasks with short periods. For $m = 8$, NP-GEDF and C-NP-GEDF perform nearly identically and worse than PRP. This is because C-NP-GEDF can produce only one cluster satisfying (10), wherein tasks with short periods can experience starvation. PRP, which employs a different clustering heuristic, also attempts to limit the starvation of tasks with short periods. However, in contrast to C-NP-GEDF, under PRP, the number of processors reserved to a cluster is not fixed and a task can be scheduled if there are additional available processors.

A bound on maximum table staleness that can be guaranteed for NP-GEDF and C-NP-GEDF is quite reasonable, as illustrated in Figure 5(b), where the ratio $\frac{\widehat{\mathsf{MWS}}(m) - \mathsf{MWS}(m)}{\mathsf{MWS}(m)}$ is shown. For large platforms, $\widehat{\mathsf{MWS}}(m)$ is better for C-NP-GEDF because the clustering algorithm eliminates interference among tasks with substantially different execution times. However, some pessimism remains in the estimation so that the predicted

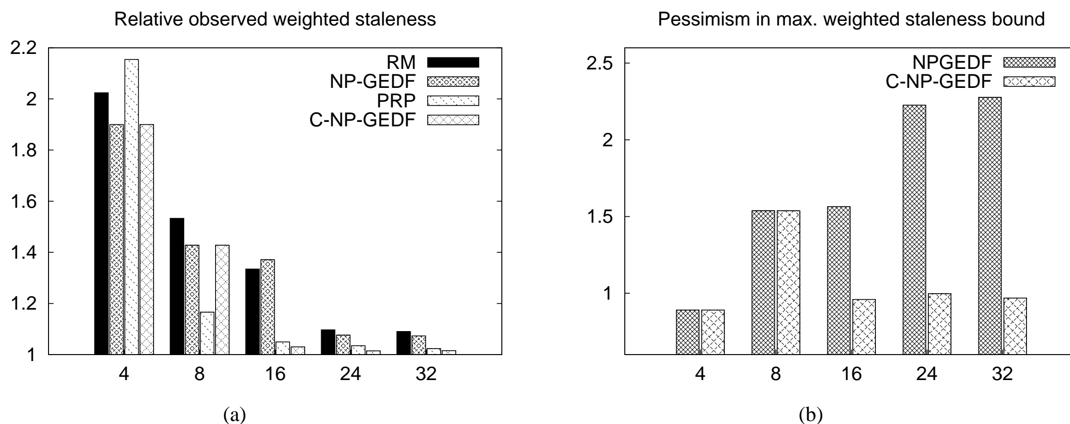


Figure 5. **(a)** Observed staleness and **(b)** staleness bound pessimism for different platform sizes.

$\widehat{\mathsf{MWS}}(m)$ is approximately two times larger than the observed $\mathsf{MWS}(m)$.

These experiments show that NP-GEDF and its clustered variant exhibit performance close to or better than that of the state-of-the-art PRP scheduler in terms of the observed maximum staleness bound. However, in contrast to PRP, NP-GEDF and C-NP-GEDF provide *a priori guarantees* on individual table freshness.

6. Conclusion

In this paper, we have presented a general framework for deriving upper bounds on data staleness in a real-time data warehouse running on a multiprocessor platform. This framework is the first to enable guarantees on data freshness to be made. Our experimental evaluation showed that the performance of simple schedulers (when used within our framework) is comparable to and often better than that of sophisticated heuristics.

Several interesting avenues for further work exist. The most important open problem is to model different aspects of update jobs more accurately. This includes incorporating memory bus contention, disk accesses, etc., in the analysis. It is also important to account for changes in query workloads and aperiodic user-induced queries, which may be a result of configuration changes.

Another important direction for future work is scheduling updates during overload situations. In real data warehouses, overloads happen and the scheduler must ensure that important tables remain consistent. Updates to less important tables can be deferred, but eventually all updates must be applied.

References

[1] D. Arthur and S. Vassilvitskii. K-means++ the advantages of careful seeding. In *Proc. of Symposium on Discrete Algorithms (SODA)*, pages 1027–1035, 2007.

[2] M. Bateni, L. Golab, M. Hajiaghayi, and H. Karloff. Minimizing staleness and stretch in real-time data warehouses. In *Proc. of SPAA*, 2009. To appear.

[3] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(4):553–566, 2009.

[4] L. Colby, A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. Supporting multiple view maintenance policies. In *Proc. of SIGMOD*, pages 405–416, 1997.

[5] U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2006.

[6] 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*, pages 1207–1210, 2009.

[7] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003.

[8] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[9] K. Jeffay, D. Stanat, and C. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proc. of the 12th IEEE Symposium on Real-Time Systems*, pages 129–139, 1991.

[10] A. Karakasidis, P. Vassiliadis, and E. Pitoura. ETL queues for active data warehousing. In *Proc. IQIS*, pages 28–39, 2005.

[11] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-time Systems*, 2008. In submission.

[12] H. Leontyev and J. Anderson. A unified hard/soft real-time schedulability test for global EDF multiprocessor scheduling. In *Proc. of the 29th IEEE Real-Time Systems Symposium*, pages 375–384, 2008.

[13] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.-E. Frantzell. Supporting streaming updates in an active data warehouse. In *Proc. of ICDE*, pages 476–485, 2007.

[14] X. Song and J.W. S. Liu. Maintaining temporal consistency: Pessimistic vs. optimistic concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):786–796, 1995.

[15] 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*, pages 2–7, 1996.

Appendix

In this appendix, we prove Lemma 2. We first prove three auxiliary claims.

Claim A1. *$Z_{i,k,j} + p_k \geq Z_{i,k,j+1}$ for all $j \geq 1$.*

Proof.

$$\begin{aligned} & Z_{i,k,j} + p_k \\ & \quad \{\text{by Definition 6}\} \\ & = \max(0, r_{k,j} - \max(0, \phi_k - p_k) - A(V_i)) + p_k \\ & = \max(p_k, r_{k,j} + p_k - \max(0, \phi_k - p_k) - A(V_i)) \\ & \quad \{\text{because } r_{k,j+1} = r_{k,j} + p_k.\} \\ & \geq \max(0, r_{k,j+1} - \max(0, \phi_k - p_k) - A(V_i)) \\ & = Z_{i,k,j+1} \end{aligned}$$

□

Claim A2. *If $\mathsf{F}(V_i, t) \geq \max(0, t - A(V_i))$ for all $t \geq 0$, then $\mathsf{F}(V_i, s_{k,j}) \geq Z_{i,k,j}$.*

Proof.

$$\begin{aligned} & \mathsf{F}(V_i, s_{k,j}) \\ & \quad \{\text{by the condition of the claim}\} \\ & \geq \max(0, s_{k,j} - A(V_i)) \\ & \quad \{\text{because } s_{k,j} \geq r_{k,j}\} \\ & \geq \max(0, r_{k,j} - A(V_i)) \\ & \geq \max(0, r_{k,j} - \max(0, \phi_k - p_k) - A(V_i)) \\ & \quad \{\text{by Definition 6}\} \\ & = Z_{i,k,j} \end{aligned}$$

□

Claim A3. *If $\mathsf{F}(V_k, f_{k,j}) \geq \min_{V_i \in pred(V_k)}\{Z_{i,k,j}\}$, then $\mathsf{F}(V_k, f_{k,j}) + p_k \geq \min_{V_i \in pred(V_k)}\{Z_{i,k,j+1}\}$*

Proof. By the condition of the claim, we have

$$\begin{aligned} & \mathsf{F}(V_k, f_{k,j}) + p_k \\ & \quad \geq \min_{V_i \in pred(V_k)}\{Z_{i,k,j}\} + p_k \\ & \quad = \min_{V_i \in pred(V_k)}\{Z_{i,k,j} + p_k\} \\ & \quad \quad \{\text{by Claim A1}\} \\ & \quad \geq \min_{V_i \in pred(V_k)}\{Z_{i,k,j+1}\}. \end{aligned}$$

□

Lemma 2 *Let V_k be a derived table. If $\mathsf{F}(V_i, t) \geq \max(0, t - A(V_i))$ for all $V_i \in pred(V_k)$ and $t \geq 0$, then $\mathsf{F}(V_k, f_{k,j}) \geq \min_{V_i \in pred(V_k)}\{Z_{i,k,j}\}$ for all $j \geq 1$.*

Proof. We prove this lemma by induction on job index j. By (3), we have

$$\begin{aligned} & \mathsf{F}(V_k, f_{k,j+1}) \\ & = \min(\mathsf{TE}(V_k, s_{k,j+1}), \mathsf{F}(V_k, s_{k,j+1}) + p_k) \\ & \quad \left\{ \begin{array}{l} \text{because } V_k \text{ is derived, by Definition 3,} \\ \mathsf{TE}(V_k, s_{k,j+1}) = \min_{V_i \in pred(V_k)}\{\mathsf{F}(V_i, s_{k,j+1})\} \end{array} \right\} \\ & = \min\left(\min_{V_i \in pred(V_k)}\{\mathsf{F}(V_i, s_{k,j+1})\}, \mathsf{F}(V_k, s_{k,j+1}) + p_k\right) \\ & \quad \{\text{by the condition of the lemma and Claim A2}\} \\ & \geq \min\left(\min_{V_i \in pred(V_k)}\{Z_{i,k,j+1}\}, \mathsf{F}(V_k, s_{k,j+1}) + p_k\right) \\ & \quad \{\text{by Rule 2}\} \\ & = \min\left(\min_{V_i \in pred(V_k)}\{Z_{i,k,j+1}\}, \mathsf{F}(V_k, f_{k,j}) + p_k\right). \end{aligned} \tag{12}$$

Base case: $j = 0$. By Rule 2, $\mathsf{F}(V_k, f_{k,0}) = 0$. By Definition 5, $A(V_k) > 0$. Thus, by Definition 6,

$$\min_{V_i \in pred(V_k)}\{Z_{i,k,1}\} \leq \max(0, r_{k,1} - \max(0, \phi_k - p_k)). \tag{13}$$

By (1), $r_{k,1} = \phi_k \geq 0$, and hence, $r_{k,1} - \max(0, \phi_k - p_k) = \min(r_{k,1}, r_{k,1} - \phi_k + p_k) = \min(\phi_k, p_k) \geq 0$. From this and (13), we have

$$\begin{aligned} \min_{V_i \in pred(V_k)}\{Z_{i,k,1}\} & \leq r_{k,1} - \max(0, \phi_k - p_k) \\ & = r_{k,1} - \max(p_k, \phi_k) + p_k \\ & = \min(r_{k,1} - p_k, r_{k,1} - \phi_k) + p_k \\ & \leq p_k, \end{aligned}$$

where the last inequality follows from (1). We thus have from (12), $\mathsf{F}(V_k, f_{k,1}) \geq \min\left(\min_{V_i \in pred(V_k)}\{Z_{i,k,1}\}, \mathsf{F}(V_k, f_{k,0}) + p_k\right) \geq \min_{V_i \in pred(V_k)}\{Z_{i,k,1}\}$.

Induction step: $j > 0$. By the induction hypothesis and Claim A3, $\mathsf{F}(V_k, f_{k,j}) + p_k \geq \min_{V_i \in pred(V_k)}\{Z_{i,k,j+1}\}$. Setting this into (12), we get the required result. □