Scheduling to Minimize Staleness and Stretch in Real-Time Data Warehouses

MohammadHossein Bateni[*]
Dept. of Computer Science
Princeton University
Princeton NJ, USA 08540
mbateni@cs.princeton.edu

MohammadTaghi Hajiaghayi
AT&T Labs–Research
Florham Park, NJ, USA 07932
hajiagha@research.att.com

Lukasz Golab
AT&T Labs–Research
Florham Park, NJ, USA 07932
lgolab@research.att.com

Howard Karloff
AT&T Labs–Research
Florham Park, NJ, USA 07932
howard@research.att.com

ABSTRACT

We study scheduling algorithms for loading data feeds into real time data warehouses, which are used in applications such as IP network monitoring, online financial trading, and credit card fraud detection. In these applications, the warehouse collects a large number of streaming data feeds that are generated by external sources and arrive asynchronously. Data for each table are generated at a constant rate, different tables possibly at different rates. For each data feed, the arrival of new data triggers an *update* that appends the new data to the corresponding table; if multiple updates are pending for the same table, they are batched together before being loaded. At time τ, if a table has been updated with information up to time $r \leq \tau$, its *staleness* is defined as $\tau - r$.

Our first objective is to schedule the updates on one or more processors in a way that minimizes the total staleness. In order to ensure fairness, our second objective is to limit the maximum "stretch", which we define (roughly) as the ratio between the duration of time an update waits till it is finished being processed, and the length of the update.

In contrast to earlier work proving the nonexistence of constant-competitive algorithms for related scheduling problems, we prove that *any* online nonpreemptive algorithm, no processor of which is ever voluntarily idle, incurs a staleness at most a constant factor larger than an obvious lower bound on total staleness (provided that the processors are sufficiently fast). We give a constant-stretch algorithm, provided that the processors are sufficiently fast, for the *quasiperiodic model*, in which tables can be clustered into a few groups such that the update frequencies within each group

vary by at most a constant factor. Finally, we show that our constant-stretch algorithm is also constant-competitive (subject to the same proviso on processor speed) in the quasiperiodic model with respect to total weighted staleness, where tables are assigned weights that reflect their priorities.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms**]: Non-numerical Algorithms and Problems—*Sequencing and scheduling*; H.2.7 [**Database Management**]: Database Administration—*Data warehouse and repository*

General Terms

Algorithms, Theory

Keywords

On-line scheduling, data warehouse maintenance, competitive analysis

1. INTRODUCTION

Data warehouses integrate information from multiple operational databases to enable complex business analyses. In traditional applications, warehouses are updated periodically (e.g., every night or once a week) and data analysis is done off-line [11]. In contrast, real time warehouses [7], also known as *active warehouses* [13], continually load incoming data feeds to support time-critical analyses. For instance, an Internet Service Provider (ISP) may collect streams of network configuration and performance data generated by remote sources in nearly real time. New data must be loaded in a timely manner and correlated against historical data to quickly identify network anomalies, denial-of-service attacks, and inconsistencies among protocol layers. Similarly, online stock trading applications may discover profit opportunities by comparing recent transactions in nearly real time against historical trends. Banks may be interested in analyzing incoming streams of credit card transactions to protect customers against identity theft.

Since the effectiveness of a real time warehouse depends on its ability to ingest new data, we study problems related to data staleness. In our setting, the arrival of a set of new data releases an *update* that seeks to append the data to the

---

corresponding table. Since existing data are not modified, the processing time of an update is at most proportional to the amount of new data. The first update made available for table i contains data accumulated in time period $I_{i1} = (0, r_{i1}]$. The second update made available for table i contains data for time period $I_{i2} = (r_{i1}, r_{i2}]$, and the third, for $I_{i3} = (r_{i2}, r_{i3}]$, etc., $0 = r_{i0} < r_{i1} < r_{i2} < \cdots$. Processing of interval I_{ij} can start at any time greater than or equal to r_{ij} and takes time, at most, proportional to the length of the interval. If updates for time periods $1, 2, 3..., l$, but not $l+1$, have been (fully) processed on table i to date, then we say that table i is *current up to time* r_{il}. The *staleness* of a table i at time τ is defined as $\tau - t$ where the table is current up to time $t \leq \tau$. (It follows that a table may be stale at time τ solely because no update has arrived recently, and hence clearly no update was processed recently.) If multiple updates have accumulated for the same table, then when *any* updates are processed to table i, *all* available updates to table i are processed together, which takes time at most proportional to the sum of the corresponding intervals' lengths. This is what we mean by *batching* updates.

We consider two models that occur in practice. In the (purely) *on-line* model, sources push data to the warehouse at arbitrary times. In what we call the *quasiperiodic* model, updates for any given table arrive with roughly constant frequency. More precisely, a table is said to be B-quasiperiodic, B a positive real, if its update interarrival times vary between $B/2$ and B. We do not study the standard periodic model because it is unrealistic in our context. Even if the warehouse requests updates from the sources at regular intervals, the sources may not always respond promptly.

Since data loads may be long-running and difficult to suspend, our first objective is to nonpreemptively schedule the updates on one or more processors in a way that minimizes the total staleness. Our first contribution answers a question implicit in [8] regarding the difficulty of this problem. We prove that even in the purely online model, *any* on-line nonpreemptive algorithm achieves staleness at most a constant factor times optimal, provided that no processor is ever voluntarily idle and provided that the processors are sufficiently fast. ("Sufficiently fast" means here that the processors be a small constant factor faster than is required for them to "keep up" with the arrival of incoming data.)

Previous scheduling results focus on individual *job* penalties, rather than *table* penalties, and employ no notion of batching, which is crucial to our result. Perhaps the most similar problem studied in the literature (initially by N. Bansal [3]) is that of minimizing the sum of squares of *flow times*, where flow time measures the total time a job spends in the system, including wait time and processing time. For this objective function, no constant competitive algorithm exists. The proof of nonexistence of a competitive algorithm for this problem, as for so many others, relies on the fact that N jobs have to pay N penalties. Specifically, consider a sequence of N consecutive, identical, unit-time jobs arriving starting at time 0 and ending at time N. Even if all N jobs could be (started and) completed instantaneously at time N, the flow-time-squared penalty would be N^2 (for the first job) plus $(N-1)^2$ (for the second) plus $(N-2)^2$, etc., for a total penalty of $\Omega(N^3)$. In our batched model, the staleness of a table depends on the time of the last update and increases linearly over time, until the next batch of updates has been processed. Regardless of whether one long

update or N unit-length updates (which will be batched together) are processed at time N, we will show that the total staleness is $O(N^2)$. Hence, our model prevents an adversary from injecting a long stream of identical short jobs which will hugely cost the algorithm. (Of course, this explanation, while intended to hint at the difference between our batched model and previous job-based models, proves nothing.)

Now we discuss model variants. Even if an algorithm provides a bound on total staleness, it may still starve some tables. To quantify this behavior, we will define the "stretch" to be the maximum, over all updates, (roughly) as the worst-case ratio between the duration of time the update waits till the processing of said update terminates, and the length of that update. In general, no constant bound on maximum stretch, as we have defined it, is possible. However, as our second contribution, we provide a constant-stretch nonpreemptive algorithm for the (practical) quasiperiodic model, in which tables can be divided into relatively few groups such that all the tables in one group are B-quasiperiodic for the same B. (Imagine that some tables are updated roughly every minute, some roughly hourly, and the rest, roughly daily.) Specifically, we assume that the number of groups is at most half the number of processors.

In practice, some tables are more critical than others. Our third contribution involves minimizing total *weighted* staleness, when each table is accompanied by a weight and the total staleness of each table is multiplied by its weight. We prove that our constant-stretch algorithm is also constant competitive with respect to total weighted staleness in the quasiperiodic model.

1.1 Related Work

A great deal of work exists on scheduling to achieve various objectives. In particular, no competitive algorithms exist for the two problems closest to ours: as mentioned above, [3] proves a negative result for minimizing the sum of squares of flow times, and [4] discusses minimizing the maximum stretch, defined as flow time/processing time. (Their definitions of flow time and stretch differ from ours.)

In the database literature, the closest work to ours is [8], which presented an empirical study of update scheduling algorithms in a real time warehouse and showed that most reasonable algorithms work well with respect to staleness. A simple boolean definition of staleness was used in [6, 12] to model updates that overwrite existing data, in contrast to the append-only model assumed in this paper, which is more appropriate for streaming data. According to this definition, at any time, an object is either fresh or stale. There has also been work on scheduling in real-time databases [1, 10], but the focus was on scheduling queries to meet some quality-of-service requirements (e.g., deadline or response time), not updates. Similarly, related work on scheduling in Data Stream Management Systems only deals with scheduling query operators, e.g., to maximize throughput or minimize memory usage [2, 5, 9, 14].

1.2 Organization

The remainder of this paper is organized as follows. In Section 2, we explain our scheduling model. Section 3 presents a competitiveness proof with respect to staleness. Section 4 discusses stretch, Section 5 deals with weighted staleness, and Section 6 concludes the paper.

2. SCHEDULING MODEL

In our real-time data warehouse model, each table i receives updates from an external source at times $r_{i1} < r_{i2} < \cdots < r_{i,k_i}$ with $0 < r_{i1}$; define $r_{i0} = 0$. These times are unknown in advance to the algorithm. At time r_{ij} an update arrives for table i, and only then does the system know the value of r_{ij}. The update that arrives at time r_{ij} contains data for the time period $(r_{i,j-1}, r_{ij}]$ [1]; however, those data do not become available to be processed till time r_{ij}. We define the *release time* or *arrival time* of the update to be r_{ij} and the *start time* of the update to be $r_{i,j-1}$. That is, the update contains data for the time period (start time, release time]. We define the *length* L_{ij} of the jth update to table i to be $r_{ij} - r_{i,j-1}$.

Let t be the number of tables and $p \le t$ be the number of available identical processors. At any point in time, any idle processor may choose to update any table, so long as this table has at least one pending update and is not currently being updated by another processor. Suppose that we are at time τ and table i is picked, and that it is current up to time r_{ij}. Since it is cheaper to execute an update of length L than l updates of length L/l each [2], all the pending updates for table i with release times in the interval $[r_{ij}, \tau]$ are batched together and processed nonpreemptively by the given processor. We refer to all of these pending updates as a *batch* and define its length as the sum of the lengths of the pending updates. Any update for table i that is released after the start of the processing of a batch is not included in that batch, and waits its turn to be processed in the next batch.

The *wait interval* of a batch is the time between the arrival (e.g., release) of its first update and the time it starts being processed. Zero or more additional updates may arrive during the wait interval of a batch. (Keep in mind that the data in the first update of the batch corresponds to a time period predating the update's release time.) The *processing time* of a batch is the length of its processing or execution interval—the time during which it is actually being processed. The *wait time* of a batch is the duration between the release of its first update and the time when processing of that batch begins.

We assume that the processing time of a batch is at most proportional to the quantity of new data, with a constant of proportionality which depends on the table (since an hour's worth of updates to table i might be far more voluminous than an hour's worth for table i'). Formally, for each table i, we define a real $\alpha_i \le 1$ such that processing a batch of length L takes time at most $\alpha_i L$. (That processing an update containing an hour's worth of data should take significantly less than one hour implies that α_i should be at most 1.) In order to keep up with the inputs, we assume that there is an $\alpha \le p/t$ such that each $\alpha_i \le \alpha$. Any t tables, from time 0 to some time T, receive data for tT time units. In order that the processors not "fall behind," it is necessary that p processors be able to process tT units of data in T time steps. Since T time units of data on table i can be processed in $T\alpha_i$ time, we need $\sum_{i=1}^{t}(T\alpha_i) \le pT$, since there are p processors, or $\sum \alpha_i \le p$. If one wants a

---

[1] Technically, an update arriving at time r_{ij} contains data up to time $r_{ij} - \epsilon$, where ϵ is the transmission delay between the source and the warehouse. For simplicity, we set $\epsilon = 0$.
[2] There are nontrivial fixed costs associated with updating a warehouse table, such as preprocessing the raw data.
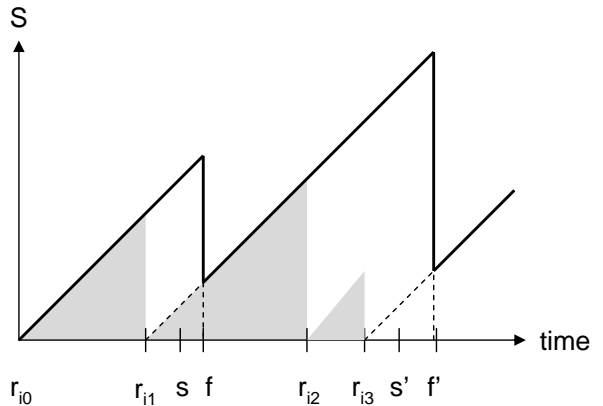

Figure 1: A plot of the staleness of table i over time.

bound in terms of the maximum α_i alone, one would need to impose $t\alpha \le p$, where $\alpha = \max \alpha_i$. Hence $\alpha \le p/t$ is required. Our results only hold if $\alpha \le Cp/t$ for some constant $C < 1$, but this upper bound is only a small constant factor away from what is necessary.

(Note: Even when $\alpha > p/t$, one conceivably could give an on-line algorithm which is constant competitive against the best off-line algorithm. However, since even the adversary would badly fall behind, we do not consider this case interesting.)

2.1 Staleness and Strech

At any time τ, the *staleness* $S_i(\tau)$ of table i is defined to be $\tau - r$, where the table is current up to time r. The total staleness of table i in the time interval $[\tau_0, \tau_1]$ is $\int_{\tau_0}^{\tau_1} S_i(\tau)d\tau$. It is important to study Figure 1, which illustrates the staleness of table i over time. Suppose that table i is initialized at time $r_{i0} = 0$. Ignore the shaded triangles for now. Suppose that a processor becomes available for table i at time s. Only one update, the first, which contains data for interval $(r_{i0}, r_{i1}]$, is available to be processed at time s. The processing of the update takes time at most $\alpha_i(r_{i1} - r_{i0})$ and finishes at time $f \le s + \alpha_i(r_{i1} - r_{i0})$. From time $r_{i0} = 0$ until the processing of the first update finishes at time f, the staleness increases linearly, with slope 1, starting at 0 and ending at time f at value f. As soon as the processing of that update finishes, at time f, the staleness drops to $f - r_{i1}$, since at time f the table is current to time r_{i1}. Immediately after time f the staleness increases linearly, with slope 1, once again.

Now suppose no processor is available again till time s'. At that time, two more updates are available, one containing data for time $(r_{i1}, r_{i2}]$ and one containing data for time $(r_{i2}, r_{i3}]$. These two are batched together. This means that the processor processes both updates, starting at time s' and finishing at time $f' \le s' + \alpha_i(r_{i3} - r_{i1})$. By time f', the staleness has risen to $f' - r_{i1}$. Immediately after time f', however, the staleness drops to $f' - r_{i3}$, since the table is then current up to time r_{i3}. It is important to note that the staleness function would not change if instead of these two updates, a single update with data for time $(r_{i1}, r_{i3}]$ arrived at time r_{i3}.

Traditionally, the *flow time* of a job is defined as the difference between its completion time and release time, and its stretch is the flow time divided by its length. However, our updates start accumulating data before they are released, which affects the staleness of the corresponding table. We thus define the flow time of the update released at time r_{ij} to be $f - r_{i,j-1}$, where the processing of the batch containing this update finishes at time f, i.e., its completion time minus its *start* time, not its completion time minus its release time (the *completion time* of a batch being the time when the processing of the batch finishes). Further, we define the *stretch* to be the maximum, over all updates, of the flow time of the update divided by the length of the update. Stretch indicates how much additional staleness is accrued while an update is waiting and being processed. For instance, in Figure 1, the stretch of the first update is $\frac{f - r_{i0}}{r_{i1} - r_{i0}}$, the stretch of the second update is $\frac{f' - r_{i1}}{r_{i2} - r_{i1}}$, and the stretch of the third update is $\frac{f' - r_{i2}}{r_{i3} - r_{i2}}$.

2.2 Lower and Upper Bounds on Staleness

Let $\mathsf{LOW} := \sum_{j>0} (r_{ij} - r_{i,j-1})^2$; then $(1/2)LOW$ is a lower bound on the total staleness of any run, even of the optimal, prescient run. (It is a lower bound on the staleness of any way to execute the jobs, since the staleness at any time x is at least the duration between the most recent release time τ and x, and $\int_0^{r_{ij} - r_{i,j-1}} x \, dx = (1/2)(r_{ij} - r_{i,j-1})^2$.) In Figure 1, the area of the shaded triangles is exactly $(1/2)LOW$. We will show that under mild conditions, the staleness achieved by any algorithm exceeds $(1/2)LOW$ by at most a constant factor.

We also define a *penalty*, half of which is an upper bound on the total staleness, as the sum of squares of the batch flow times. In Figure 1, the penalty is $(f - r_{i0})^2 + (f' - r_{i1})^2$. (Notice that the flow time for the first batch, which includes only update 1, is $f - r_{i0}$; that of the second batch, which includes updates 2 and 3, is $f' - r_{i1}$.) Thus, half the penalty is the sum of the areas of the triangles based at intervals $(r_{i0}, f]$ and $(r_{i1}, f']$, which, because the triangles overlap on the dotted triangles, is at least as great as the staleness.

Now we argue formally that twice staleness cannot be larger than the penalty we pay, as defined above. For each specific table, partition the time frame into intervals demarcated by the endpoints of execution intervals. For instance, one such partition in Figure 1 corresponds to the interval $(f, f']$. The integration diagram for each of these updates consists of a trapezoid, starting and ending, respectively, at, say, times r and r'. We denote by y the staleness value immediately after time r, the end of the previous execution interval. Staleness accrues linearly between times r and r', reaching a value of $y + r' - r$ at time r'. The total staleness for this batch is the area of the trapezoid whose base has a length of $r' - r$ and whose height ranges from y to $y + r' - r$. This amounts to

$$(r' - r) \left[\frac{(y) + (y + r' - r)}{2} \right] \quad (1)$$

$$\leq \frac{(r' - r + y)^2}{2}, \quad (2)$$

as $y \geq 0$ and $xz \leq \left(\frac{x+z}{2} \right)^2$ for $x, z \geq 0$. Let a be the release time of the last update processed in the execution interval which ends at time r. Then $y = r - a$. The penalty is $(r' - a)^2 = (r' - (r - y))^2$, so $(r' - r + y)^2/2$ is exactly half

the penalty paid for this batch according to our objective function.

3. MINIMIZING STALENESS

We call an algorithm *eager*, or *work-conserving*, if it leaves no processor idle while at least one pending update exists. We first state the rather-inscrutable Theorem 3.1, followed by an easy-to-read corollary, which implies that for any $C < (\sqrt{3} - 1)/2$, there is a constant (dependent on C) such that the staleness of any eager algorithm is at most that constant factor times optimal, provided that each α_i is at most Cp/t.

THEOREM 3.1. *Fix p, t. For any β and δ such that $0 < \beta, \delta < 1$, define $C_{\beta,\delta} = \sqrt{\delta}(1 - \beta)/\sqrt{3} > 0$. Given p processors and t tables, pick any α such that $\alpha/[1 - \alpha/(p/t)] \leq C_{\beta,\delta} \cdot p/t$. Then the penalty incurred by an eager algorithm is at most $(1 + \alpha)^2 (1/\beta^4)(1/(1 - \delta))$ times LOW, provided that each $\alpha_i \leq \alpha$.*

COROLLARY 3.2. *Fix p, t. Suppose $\alpha := \max \alpha_i$ satisfies $\alpha = Cp/t$ with $C < (\sqrt{3} - 1)/2 \approx 0.366$. Then the penalty (and hence staleness) incurred by any eager algorithm is at most a constant factor times LOW. Furthermore, as $\alpha \to 0$, the constant factor approaches 1.*

Since LOW is a lower bound on the staleness achieved by any algorithm, even the optimal, prescient one, and *penalty* is an upper bound on the staleness achieved by any eager algorithm, the corollary implies the claimed competitiveness.

PROOF. Define C' by $1/C = 1 + 1/C'$. Because $C < (\sqrt{3} - 1)/2$, it follows that $0 < C' < 1/\sqrt{3}$. Hence $\alpha = [1/(1/C' + 1)](p/t)$. Simple algebra shows that $\alpha/(1 - (\alpha/(p/t)) = C'(p/t)$. Because $0 < C'\sqrt{3} < 1$, there are β, δ in $(0, 1)$ such that $C'\sqrt{3} = \sqrt{\delta}(1 - \beta)$. (Define $\gamma = C'\sqrt{3}$, take any $\beta \in (0, 1 - \gamma)$, and take δ such that $\sqrt{\delta}(1 - \beta) = \gamma$.) Therefore $C' = \sqrt{\delta}(1 - \beta)/\sqrt{3}$. Hence, Theorem 3.1 implies that the penalty is at most $(1 + \alpha)^2 (1/\beta^4)(1/(1 - \delta))$ times LOW, as claimed.

Given any $\epsilon > 0$, first choose $0 < \beta < 1$ such that $1/\beta^4 \leq 1 + \epsilon/4$. Then choose $0 < \delta < 1$ such that $1/(1-\delta) \leq 1 + \epsilon/4$. Define $C_{\beta,\delta} = \sqrt{\delta}(1 - \beta)/\sqrt{3} > 0$. Choose $\alpha > 0$ such that $(1 + \alpha)^2 \leq 1 + \epsilon/4$ and $\alpha/(1 - \alpha/(p/t)) \leq C_{\beta,\delta} \cdot p/t$. Now if each $\alpha_i \leq \alpha$, the penalty is at most $(1+\alpha)^2 (1/\beta^4)(1/(1-\delta))$ times LOW. Since $(1 + \alpha)^2$, $1/\beta^4$, and $1/(1 - \delta)$ are all at most $1 + \epsilon/4$, and $(1 + \epsilon/4)^3 \leq 1 + \epsilon$ for $\epsilon \leq 1$, we are done. □

To start the proof of Theorem 3.1, let us look at the penalty a particular run of the algorithm pays. Let $\mathcal{B}$ be the set of batches in this run. For some batch $B_i \in \mathcal{B}$, let c_i be the length of the first update, d_i be the wait time, and b_i be the total length of the batch, i.e., the sum of the lengths of its updates. Clearly,

$$c_i \leq b_i \leq c_i + d_i, \quad (3)$$

since $c_i \leq b_i$ is obvious and since $c_i + d_i$ is the duration in time from the start (not release) time of the first job in the batch till the update for the batch starts, and this duration is clearly at least the length b_i of the batch. For the penalty of this batch, denoted by ρ_i, we take the square of the flow time, i.e., the length c_i of the first update plus the wait time

d_i plus the processing time of the entire batch:

$$\rho_i \le [(c_i + d_i) + \alpha b_i]^2, \quad \text{by the definition of penalty,} \quad (4)$$

$$\le (1+\alpha)^2 (c_i + d_i)^2, \quad \text{by (3).} \quad (5)$$

Let $\mathcal{A}$ be the set of all updates. From the definition of *LOW*, each update $i \in \mathcal{A}$ has a budget of a_i^2 units, where a_i is the length of update i. Our proof requires the use of a "charging scheme." A *charging scheme* specifies what fraction of its budget each update pays to a certain batch. Let us call a batch B_i *tardy* if $c_i < \beta(c_i + d_i)$ (where β comes from Theorem 3.1); otherwise it is *punctual*. Let us denote the corresponding sets by $\mathcal{B}_t$ and $\mathcal{B}_p$ respectively. More formally, a charging scheme is a matrix (v_{ij}) of nonnegative values, where v_{ij} shows the extent of dependence of batch i on the budget available to batch j, with the following two properties.

1. For any batch $B_i \in \mathcal{B}$,

$$(c_i + d_i)^2 \le \sum_{j \in \mathcal{B}_p} v_{ij} b_j^2, \quad (6)$$

   both i and j referring to batches, and

2. there exists a constant $\lambda > 0$ such that, for any punctual batch B_j,

$$\sum_{i \in \mathcal{B}} v_{ij} \le \lambda. \quad (7)$$

LEMMA 3.3. *The existence of a charging scheme with parameters β and λ gives a competitive ratio of at most $(1+\alpha)^2 \lambda / \beta^2$ for any eager algorithm.*

PROOF. We have

$$\rho_i \le (1+\alpha)^2 (c_i + d_i)^2 \qquad \text{by (5)} \quad (8)$$

$$\le (1+\alpha)^2 \sum_{j \in \mathcal{B}_p} v_{ij} b_j^2 \qquad \text{by (6)} \quad (9)$$

$$\le (1+\alpha)^2 \sum_{j \in \mathcal{B}_p} v_{ij} \frac{1}{\beta^2} c_j^2, \quad (10)$$

the last inequality following by 3 and the definition of punctuality. Hence, the total penalty of a solution is

$$\sum_{i \in \mathcal{B}} \rho_i \le (1+\alpha)^2 \frac{1}{\beta^2} \sum_{j \in \mathcal{B}_p} c_j^2 \sum_{i \in \mathcal{B}} v_{ij} \qquad \text{by (10)} \quad (11)$$

$$\le (1+\alpha)^2 \frac{1}{\beta^2} \lambda \sum_{j \in \mathcal{B}_p} c_j^2 \qquad \text{by (7)} \quad (12)$$

$$\le (1+\alpha)^2 \frac{1}{\beta^2} \lambda \cdot \mathsf{LOW} \quad (13)$$

$$\le (1+\alpha)^2 \frac{1}{\beta^2} \lambda \cdot \mathsf{OPT}. \quad (14)$$

$\square$

Say batch B *blocks* batch B' if B's execution interval has intersection of positive length with the wait interval of B' (note that many batches can block a given batch B'). We now introduce a charging scheme with the desired properties by defining how the v_{ij} values are computed. If a batch B_i is punctual, this is simple: all v_{ij} values are zero except for $v_{ii} = 1/\beta^2$. Take a tardy batch B_i. In this case d_i is large compared to c_i. During the wait interval of length d_i during

which B_i is waiting, all p processors should be busy. Let $(r, r']$ denote this interval. The total time, summed over p processors, is pd_i. Claim 3.5 establishes a relaxed version of this bound to draw the conclusion.

Build a weighted directed graph with one node for each batch. Punctual batches are sinks, i.e., have no out-arcs. Any tardy batch has arcs to all the batches blocking it, and there is at least one, since it has positive d_i. Even though punctual batches may be blocked by other batches, they have no out-arcs.

The result is a directed acyclic graph (DAG), because along any directed path in the graph, the execution start times of batches are decreasing. The weight w_e on any such arc $e = (B, B')$ is the fraction, between 0 and 1, of the execution interval of the blocking batch B' which is inside the wait interval of the blocked batch B. We also have a parameter γ,

$$\gamma := \frac{3t\alpha^2}{p^2} \left(\frac{1}{(1-\beta)[1 - \alpha/(p/t)]} \right)^2. \quad (15)$$

Then, for any two batches i and j, v_{ij} is defined as

$$v_{ij} = \frac{1}{\beta^2} \sum_{p \in P_{ij}} \prod_{e \in p} (\gamma w_e^2), \quad (16)$$

where P_{ij} denotes the set of directed paths from i to j. The dependence along any path is the square of the product of weights on the path multiplied by γ to the power of the length of the path. This definition includes as a special case the definition of the v_{ij}'s for punctual batches i, since there is a path of length zero between any batch i and itself and no other path from i to itself (giving $v_{ii} = 1/\beta^2$) and no path from batch i to any batch j for $j \ne i$ (giving $v_{ij} = 0$ if $j \ne i$).

Next we show that such a charging scheme satisfies the desired properties. The penalty paid for each batch should be accounted for using the budget it secures, as (6) requires.

LEMMA 3.4. *For any batch $B_i \in \mathcal{B}$, $(c_i + d_i)^2 \le \sum_{j \in \mathcal{B}_p} v_{ij} b_j^2$.*

We will prove Lemma 3.4 shortly. We need the following claim in the proof of Lemma 3.4.

CLAIM 3.5. *If $B_1, \ldots, B_k$ are the children of tardy batch B_0, with arcs $(B_0, B_1), (B_0, B_2), ..., (B_0, B_k)$ having weights $w_{01}, w_{02}, \ldots, w_{0k}$, respectively, in a run of any eager algorithm, then $\sum_{j'=1}^{k} (w_{0j'} b_{j'})^2 \ge p^2 d_0^2 [1 - \alpha/(p/t)]^2 / (3t\alpha^2)$.*

PROOF. By the definition of the w_e's, the construction of the graph, the fact that $B_1, B_2, ..., B_k$ are all the batches blocking B_0, and the fact that (parts of) the k blocking batches are executed on p processors in a wait interval of length d_0 (so that their actual lengths must sum to at least $1/\alpha$ times as much), we know that

$$\sum_{j'=1}^{k} w_{0j'} b_{j'} \ge pd_0/\alpha. \quad (17)$$

We now remove all but $3t$ of the batches, such that the sum of sizes of the remaining batches is relatively large.

Choose r, r' such that $(r, r']$ is the wait interval, of length $d_0 = r' - r$, corresponding to batch B_0. Among $B_1, B_2, ..., B_k$, there might be one batch per processor whose execution starts before r and does not finish until after r.

We keep these (at most) p batches, and in addition the first at-most-two other batches for each table whose execution interval has a positive-length intersection with this interval (r, r'), at most $p + 2t \leq 3t$ in total. We show that the contribution of the other batches, however many they might be, is small. Consider the third (and later) batches at least part of which were executed during (r, r'), from one fixed table. Clearly the release times are at most r'. We now prove that their start times are no smaller than r. Suppose that a processor executes a batch, say, Q_0, whose execution starts before r and terminates after r. Enumerate the batches at least part of each of which was executed on this table during time interval (r, r'), calling them $Q_0, Q_1, Q_2, Q_3, ...$, in order of execution. Now the key point is that when batch Q_l is executed, $l = 1, 2, 3, ...$, all updates to the same table which were released at or after the beginning of Q_{l-1}'s execution interval and prior to the beginning of Q_l's execution interval, and no others, are executed. Therefore all updates performed in batch Q_3 were released at or after the beginning of Q_2's execution interval. These updates had start times no earlier than the beginning of the execution interval of Q_1. But all of Q_1 was executed after time r, implying that no update of Q_3 has a start time which is less than r. The same statement is true for batches Q_l, $l \geq 4$.

Let K be the set of omitted batches and let K' be the set of at-most-$3t$ remaining batches. Summing over the t tables, it follows that $\sum_{j' \in K} w_{0j'} b_{j'} \leq t d_0$. This means, in conjunction with (17), that

$$\sum_{j' \in K'} w_{0j'} b_{j'} \geq p d_0 (1/\alpha) - t d_0 \geq d_0 p / \alpha [1 - \alpha/(p/t)]. \quad (18)$$

Now we use the general fact that

$$\sum_{i=1}^{N} x_i^2 \geq \left(\sum_{i=1}^{N} x_i \right)^2 / N \quad (19)$$

to infer that

$$\sum_{j' \in K'} (w_{0j'} b_{j'})^2 \quad \geq ([1 - \alpha/(p/t)] p d_0 / \alpha)^2 / (3t) \quad (20)$$

$$\geq p^2 d_0^2 [1 - \alpha/(p/t)]^2 / (3t\alpha^2). \quad (21)$$

$\square$

Now we are ready to prove Lemma 3.4, which assures that each batch receives a sufficient budget.

PROOF OF LEMMA 3.4. Let the *depth* of a node be the maximum number of arcs on a path from that node to a node of outdegree 0. (The punctual nodes are the only nodes of outdegree 0.) We use induction on the depth of nodes to prove the lemma, i.e., we want to prove, for any Δ, for any node B_i of depth at most Δ, that $(c_i + d_i)^2 \leq \sum_{j \in \mathcal{B}_p} v_{ij} b_j^2$. For sinks, i.e., nodes of outdegree 0 (these are the punctual batches), the claim is obvious, since

$$(c_i + d_i)^2 \leq \frac{1}{\beta^2} c_i^2 \quad \text{by definition of punctuality} \quad (22)$$

$$\leq \frac{1}{\beta^2} b_i^2 \quad \text{by (3)} \quad (23)$$

$$= v_{ii} b_i^2 \quad \text{by definition of } v_{ii} \quad (24)$$

$$= \sum_{j \in \mathcal{B}_p} v_{ij} b_j^2 \quad \text{because } v_{ij} = 0 \text{ if } j \neq i. \quad (25)$$

Take a tardy batch B_0 of depth Δ whose immediate children—and it has at least one—are $B_1, \ldots, B_k$. For any child B_i of B_0, whose depth has to be less than Δ, we have

$$b_i^2 \leq (c_i + d_i)^2 \quad \text{by (3)} \quad (26)$$

$$\leq \sum_{j \in \mathcal{B}_p} v_{ij} b_j^2 \quad \text{by the inductive hypothesis.} \quad (27)$$

Now we prove that the inductive assertion holds for B_0 as follows.

$$(c_0 + d_0)^2 \leq \left(\frac{1}{1 - \beta} \right)^2 d_0^2 \quad (28)$$

by definition of tardiness,

$$= \gamma \frac{p^2 d_0^2 [1 - \alpha/(p/t)]^2}{3t\alpha^2} \quad (29)$$

by the choice of γ,

$$\leq \gamma \sum_{j'=1}^{k} (w_{0j'} b_{j'})^2 \quad (30)$$

from Claim 3.5,

$$\leq \sum_{j'=1}^{k} \gamma w_{0j'}^2 \sum_{j \in \mathcal{B}_p} v_{j'j} b_j^2 \quad (31)$$

by (26) and (27),

$$= \sum_{j \in \mathcal{B}_p} \sum_{j'=1}^{k} (\gamma w_{0j'}^2) v_{j'j} b_j^2 \quad (32)$$

$$\leq \sum_{j \in \mathcal{B}_p} v_{0j} b_j^2, \quad (33)$$

by (16) and because, for $j \in \mathcal{B}_p$, we can "factor out" the first arc of the paths to get $v_{0j} = \sum_{j'=1}^{k} (\gamma w_{0j'}^2) v_{j'j}$. $\square$

The second property of a charging scheme says that the budget available to a batch should not be overused.

LEMMA 3.6. *For any batch B_j, $\sum_{i \in \mathcal{B}} v_{ij} \leq \lambda := 1/(\beta^2 (1 - t\gamma))$.*

We will see shortly that $t\gamma < 1$. We need the following claim in the proof of Lemma 3.6.

CLAIM 3.7. *The wait intervals corresponding to batches of a single table are disjoint.*

PROOF. The wait interval of a batch Q starts no earlier than the beginning of the execution interval of a previous batch and extends to the beginning of the execution interval of Q. These intervals are disjoint. $\square$

Let us now prove the second property of the charging scheme.

PROOF OF LEMMA 3.6. Let the *height* of a node be the maximum number of arcs on a path from any node to that node. We do induction on the height of nodes to prove the lemma, i.e., for any H and any node B_j of height H, $\sum_{i \in \mathcal{B}} v_{ij} \leq \lambda$.

By (15),

$$t\gamma = [3t^2 \alpha^2 / p^2] (1/[1 - \alpha/(p/t)]^2)(1/(1 - \beta))^2 \quad (34)$$

$$\leq \delta < 1 \quad (35)$$

(by definition of α, δ in Theorem 3.1). For a batch B_j at height zero (a source, i.e., a node of indegree 0), the definition of v_{ij}, which involves a sum over all $i \to j$ paths, is 0 unless $i = j$, in which case $v_{ij} = 1/\beta^2$. Now the claim that $\lambda \geq 1/\beta^2$ follows from the definition of λ and the fact that $t\gamma < 1$.

As in the previous proof, we can factor out the last arc of the path, except for the zero-length trivial path. Say we are considering B_0 whose immediate ancestors are $B_1, \ldots, B_k$ with arcs $e_1 = (B_1, B_0), \ldots, e_k = (B_k, B_0)$, respectively. These incoming arcs may come from batches corresponding to different tables. However, we show that the sum $\sum_{i=1}^{k} w_{e_i}$ of the weights of these arcs is at most t. More precisely, we show that the contribution from any table is no more than one. Remember that $w_{e_i} = w_{(B_i, B_0)}$ denotes the fraction of batch B_0 which is in the delay interval of batch B_i. As the delay intervals of these batches, for one fixed table, are disjoint, by Claim 3.7, their total weight cannot be more than 1 and hence the total sum over all tables cannot exceed t.

Further, for any e, we know that $w_e \leq 1$. So

$$\sum_{i=1}^{k} w_{e_i}^2 \leq t. \tag{36}$$

As the height of any ancestor B_i of B_0 is strictly less than H, the inductive hypothesis ensures that $\sum_{l \in \mathcal{B}} v_{li}$ is no more than λ. Hence

$$\sum_{i \in \mathcal{B}} v_{i0} = \frac{1}{\beta^2} + \sum_{i \in \mathcal{B}, i \neq 0} \sum_{i'=1}^{k} \gamma w_{i'0}^2 v_{ii'} \tag{37}$$

by definition of v_{ij} in (16), noting that $v_{00} = 1/\beta^2$, and by "factoring out" the last arc, this equals

$$\frac{1}{\beta^2} + \gamma \sum_{i'=1}^{k} w_{i'0}^2 \big(\sum_{i \in \mathcal{B}, i \neq 0} v_{ii'} \big), \text{ by } (37). \tag{38}$$

Now

$$\sum_{i \in B} v_{ii'} \leq \lambda \text{ by the inductive hypothesis applied to } i', \text{ and} \tag{39}$$

$$\sum_{i'=1}^{k} w_{(i',0)}^2 \leq t \text{ by } (36), \text{ and hence} \tag{40}$$

$$\sum_{i \in B} v_{i0} \leq \frac{1}{\beta^2} + \gamma t \lambda \text{ by } (36) \text{ and the inductive hypothesis,} \tag{41}$$

$$= \lambda \text{ by the choice of } \lambda. \tag{42}$$

$\square$

Lemmas 3.4 and 3.6 show that the matrix (v_{ij}) is indeed a charging scheme. Lemma 3.3 concludes the proof of Theorem 3.1, in light of the fact that $t\gamma \leq \delta$ is proven near the beginning of the proof of Lemma 3.6.

4. BOUNDING THE MAXIMUM STRETCH

In general, no constant bound on stretch is possible. For consider the case of any number of tables and any number of processors, in which, on table 1, an enormous update of length S_1 is released at time S_1 followed by the release of a minuscule update of length S_2 at time $S_1 + S_2$. At some point both updates have be processed. Whether they are batched together (possibly with other updates) or not, the minuscule update will not finish being processed possibly until time $S_1 + \alpha_1 S_1$, meaning that its flow time would be at least $\alpha_1 S_1$, and hence its stretch at least $\alpha_1 S_1 / S_2$. Choosing S_2 positive but arbitrarily small would make the stretch arbitrarily large. It follows that to obtain bounded stretch one must somehow restrict the inputs.

For the remainder of the paper we make the (technical) assumption that no two updates arrive at exactly the same time. We suspect (but have not tried to prove) that judicious tie breaking would obviate the need for such an assumption.

Recall that a table is *B-quasiperiodic*, B a positive real, if all updates to the table have length between $B/2$ and B. (All results we prove would go through, *mutatis mutandis*, if the "1/2" were changed to any other constant c, $0 < c < 1$.)

Now we assume that the set of tables can be divided into a small number, say, g, of groups, such that all the tables in one group are B-quasiperiodic for the same B. We need at least as many processors as the number of groups; otherwise, we can come up with examples to produce arbitrarily large stretch values. We assume instead that $p \geq 2g$. In this scenario, by assigning to each group a fraction of the p processors which is proportional to the number of tables in the associated group, we are able to bound the stretch by using the rather naive algorithm GROUPANDRUN given in Figure 2. (Actually, because of rounding, instead of assigning $(p/t)T$ processors to a group of T tables, we assign $\lceil ((p - g)/t)T \rceil$.)

First we describe algorithm MYOPIC. Say an update to a table is *pending* if, at the current time, it has been released yet no processor is processing that update. (Some processor may, of course, be processing some *other* update to that table.) Say a table is *available* if, at the current time, some update for that table is pending yet no processor is processing any update (that or any other) on that table. Number the processors $P_1, P_2, ..., P_p$ arbitrarily. When a processor of MYOPIC becomes idle, it looks at all available tables. On each, it looks, among the potentially many pending updates, at the pending update U with the earliest release time, and it chooses to process the available table for which the pending update with the earliest release time is earliest. (Of course it batches all pending updates to that table, so that U may be only one of the many updates MYOPIC processes on that table.) If multiple processors of MYOPIC become idle at exactly the same time, the idle processors, in increasing order by index, choose pending updates to process. Algorithm GROUPANDRUN naively divides the tables into groups, allocates processors roughly proportionally to the group sizes, and then runs algorithm MYOPIC on each.

Now each group forms an independent instance; from now on, there is no interaction whatsoever in the algorithm between the different groups. Prior to Theorem 4.3, there is no interaction in the analysis, either.

DEFINITION 4.1. *A* tight interval *is a maximal closed interval of time during which all the processors associated with a group are busy. A* loose interval *is a maximal open interval of time during which not all processors associated with a group are busy.*

The tight intervals can be ordered chronologically. Let I_k

Figure 2: Algorithm GroupAndRun.

be the kth tight interval and let its length be θ_k.

First we need a crucial lemma.

LEMMA 4.2. *Let ω_i denote the wait time of the ith update J_i. Let θ_k be the length of the kth tight interval I_k. Suppose that all the tables in a group with say, t' tables and p' processors, are B-quasiperiodic. If $\alpha \leq p'/(8t')$ and $\alpha \leq 1/8$, then all ω_i and all θ_k are at most $B/6$.*

PROOF. Order the set of updates associated with these t' tables according to their release times. We establish recursive bounds ω_i and θ_k and use them inductively to prove that each is at most $B/6$. There is a dependence between them, but the dependence is not circular. θ_k depends on ω_i for updates J_i which are released before I_k starts; ω_i depends on $\omega_{i'}$ for $i' < i$ and on θ_k for I_k in which J_i is released.

Let i_k be the index of the last update released before I_k starts. The dependence is as follows. Write down the sequence $<$"ω_1", "ω_2", "ω_3", ...$>$ of symbols. Place the symbol "θ_k" between "ω_{i_k}" and "ω_{1+i_k}". Let σ denote the resulting sequence. The formulas avoid circular dependence by bounding each ω_i and θ_k in terms of symbols preceding it in σ.

Fix a k. We start by proving an upper bound on θ_k, the length of the kth interval, in terms of those ω_i's having $i \leq i_k$. Let us examine the batches processed inside I_k, where r and r' denote the start and end time of the interval I_k. The release times of all of the updates in all of these batches are no greater than r'.

Let the *load* at time r be the total amount of updates with starting times at or before r. If an update is released after r, we only include in load the portion of it which occurs at or before r. Furthermore, if part of an update has been processed before time r, we only consider the portion of the update which is yet to be processed at time r.

Fix one table and let X denote its contribution to the overall load. We claim that the contribution X to load by any single table is at most $B + \max_{i \leq i_k} \omega_i$. There are three cases to consider. In the first two, no update is being processed at time r. In the third, one is.

- Suppose that at time r no update is being processed. Suppose, furthermore, that there is no pending update available at time r. The contribution to load in this case is at most the length of the update next to be released, at some time exceeding r. (Its start time is less than r; its release time exceeds r.) Since each update has length at most B, the contribution X to load in this case is at most B.

- Suppose that at time r no update is being processed but that there *is* some pending update available at time r. In this case the contribution to load is at most $B + \max_{i \leq i_k} \omega_i$, the "$B+$" appearing because the wait time ω_i is measured relative to release time, yet the contribution to load is measured relative to start time.

- Suppose that at time r some update *is* being processed. Let z be the time at which the processing of that update started. From time z to time r, as the batch was being processed, the load decreased. In fact, each unit of execution time decreased the load by at least $1/\alpha_i \geq 1/\alpha$. Obviously, $z \leq r$ and the processing of the batch continues up to at least time r. Hence

$$X \leq (r - z) + (B + \max_{i \leq i_k} \omega_i) - (r - z)/\alpha, \qquad (43)$$

where the first term correspond to a batch, possibly prior to its release time, which is being formed while the other batch is being executed, the second term bounds the length of the batch being executed, and the last term reflects the decrease in contribution to load as the batch is processed. Noting that $\alpha \leq 1$, equation (43) gives $X \leq B + \max_{i \leq i_k} \omega_i$.

Summing over t' tables, the total load at time r is at most $t'(B + \max_{i \leq i_k} \omega_i)$. This means, since interval I_k has length θ_k, that the total load at any time during I_k is at most

$$Bt' + t' \max_{i \leq i_k} \omega_i + t' \theta_k. \qquad (44)$$

Because all p' processors are busy during I_k, and because during the θ_k time units of I_k the p' processors decrease the load by at least $\theta_k p'/\alpha$, we have

$$Bt' + t' \max_{i \leq i_k} \omega_i + t' \theta_k \geq \theta_k p'/\alpha. \qquad (45)$$

Rearranging, we get

$$\theta_k \leq \frac{(B + \max_{i \leq i_k} \omega_i) t'}{p'/\alpha - t'}. \qquad (46)$$

This is the desired upper bound on θ_k in terms of ω_i's.

Now we write two upper bounds for ω_i, one in terms of $\omega_{i'}$'s for $i' < i$, the other in terms of the same $\omega_{i'}$'s and also in terms of θ_k. Without loss of generality, we only consider the wait time for the first update J_i of a batch. This obviously has the largest wait time among all the updates from the same batch.

Recall that we have assumed that no two updates arrive at exactly the same time.

A crucial point is that an update can wait (and be pending) for only two reasons: the obvious one that all the p' processors are busy, and the slightly subtle one that some processor is already executing an earlier batch associated with the same table.

We consider the latter case first. This case is the one in which J_i (which is the first update in its batch) is released in a loose interval. Let τ' be J_i's release time. If $\omega_i > 0$, then J_i must be waiting for an earlier batch from the same table (because not all p' processors are busy). Clearly the length of the other batch is at most $B + \max_{i' < i} \omega_{i'}$. We claim that as soon as this batch finishes being processed, which will occur at some time $\tau \leq \tau' + [\alpha(B + \max_{i' < i} \omega_{i'})]$, J_i will immediately start being processed, and hence that

$\omega_i \leq \alpha(\max_{i'<i} \omega_{i'} + B)$. At time τ', not all p' processors are busy; hence all these, say, N, other updates favored by MYOPIC over J_i at time τ' are either running or waiting for one batch of their own table. There must be one processor working on the table corresponding to J_i (on the batch which is delaying J_i), one for each of the N updates favored by MYOPIC over J_i, and one idle processor. Now $1 + N + 1 \leq p'$ implies that $N \leq p' - 2$. In other words, there are at most $p' - 2$ tables which MYOPIC favors to the table containing J_i at time τ' and hence also at time τ. Furthermore, MYOPIC favors J_i over any updates released in the time interval $(\tau', \tau]$. Because $p' - 2 < p'$, it follows that J_i cannot be blocked at time τ. Hence

$$\omega_i \leq \alpha \left(\max_{i'<i} \omega_{i'} + B \right). \tag{47}$$

Now we consider the case in which the update J_i is released inside a tight interval, say, I_k. If processing of J_i does not start at or before the end of I_k, then a "blocking" batch Z from the same table has to be undergoing processing at the moment I_k finishes; otherwise, J_i would start at that point. However, processing of this batch must have started before J_i was released; or else J_i would have been part of it. Moreover, as in the argument for the first case, we will prove that as soon as the processing of the blocking batch Z from the same table is done, the batch corresponding to update J_i will start to be processed.

Let τ be the time when interval I_k ends and let τ' be the time at which processing of Z is finished. Since Z is still being processed at the time when I_k finishes, $\tau' \geq \tau$. We show that exactly at time τ', the algorithm starts processing J_i on some processor. Note that if τ' is in a loose section, then the claim is trivial since there is no longer a blocking batch to prevent J_i's processing. So we may assume that τ' is in a tight section.

The release times of the first updates of all batches that are favored by MYOPIC over J_i at time τ', and might prevent the algorithm from starting the processing of J_i at time τ', are less than J_i's release time (note that we have assumed that all jobs have distinct release times). Since at time τ, we start a loose section, each of the these batches should have a batch from the same table, undergoing processing at τ. Let $B_1, B_2, \ldots, B_s$, $s \geq 1$, be the batches the release times of the first updates of which are less than J_i's release time, for which there are batches $Z_1, Z_2, \ldots, Z_s$, respectively, from the same tables undergoing processing at τ and finishing no later than τ'. Let $B_0 = J_i$ and $Z_0 = Z$. Let $\tau_0 < \tau_1 < \tau_2 < \cdots < \tau_{s'}$, $s' \leq s$, be the distinct times at which at least one Z_k, $0 \leq k \leq s$, finishes. We prove that at time $\tau_{k'}$, for any k', $1 \leq k' \leq s'$, each B_k whose corresponding blocking batch Z_k is finished exactly at time $\tau_{k'}$ starts to be processed. If not, choose k' to be the smallest counterexample for this claim. Assume that at time $\tau_{k'}$ there are exactly $r \geq 1$ blocking batches which finish. Note that by the definitions of $B_1, B_2, \ldots, B_s$, any batch whose processing has not been started at τ, whose blocking batch will be finished not later than τ', and finally favored by MYOPIC over some B_k, $1 \leq k \leq s$, should be also among $B_1, B_2, \ldots, B_s$ (since by transitivity the release time of its first update is also less than J_i's release time). Now by minimality of k', each batch B_k whose blocking batch Z_k finishes (strictly) earlier than $\tau_{k'}$ has already started its processing. On the other hand, each batch B_k whose corresponding blocking

batch Z_k has not finished yet cannot start to be processed at $\tau_{k'}$. Since there is no other batch favored by MYOPIC over these r batches whose blocking batches finish precisely at time $\tau_{k'}$, we can start processing all these r batches, including $B_{k'}$, on at least these r available processors at time $\tau_{k'}$. This is a contradiction to the assumption that $B_{k'}$ does not start to be processed at time $\tau_{k'}$. Thus, at time τ', the algorithm starts processing J_i on some processor. Hence,

$$\omega_i \leq \max \left\{ \theta_k, \alpha \left(\max_{i'<i} \omega_{i'} + B \right) \right\}, \tag{48}$$

since it either waits for the tight interval to end, or for a batch of its own table whose length cannot be more than $\max_{i'<i} \omega_{i'} + B$.

Using $\alpha \leq 1/8$ and $p'/\alpha \geq 8t'$, we prove by induction on position in σ that each ω_i and each θ_k is at most $B/6$. The basis is the case for the first update. Released in a loose interval, it will have $\omega_1 = 0 \leq B/6$. For equation (47), the right-hand side is at most $(1/8)(B/6+B) = (7/48)B < B/6$, as desired. For equation (48), the right-hand side is at most the maximum of $B/6$ and the $(7/48)B < B/6$ which we just got for the right-hand side of equation (47). Last, for equation (46), $p'/\alpha \geq 8t'$ implies that $t'/(p'/\alpha-t') \leq 1/(8-1)$, giving an upper bound on the right-hand side of equation (46) of $(7B/6) \cdot 1/7 = B/6$. $\square$

Now, we can prove that our algorithm keeps the stretch low.

THEOREM 4.3. *Algorithm* GROUPANDRUN *keeps the stretch below 3 if $\alpha \leq (p - g)/(8t)$ and $\alpha \leq 1/8$.*

Specifically, since $g \leq p/2$, $\alpha \leq p/(16t)$ and $\alpha \leq 1/8$ suffices.

PROOF. Since the sum of $(p-g)|\mathcal{T}_i|/t$ for different groups is $p-g$, $\sum_i \lceil (p-g)|\mathcal{T}_i|/t \rceil \leq p$. Thus, in total we are not using more than the p processors we have. Now $\alpha \leq (p - g)/(8t)$ implies that in a group with p' processors and t' tables, $\alpha \leq p'/(8t')$, as required by Lemma 4.2. By that lemma, any waiting time is at most $B/6$. Any update has length at most B and at least $B/2$, giving a stretch of at most $[B + B/6 + (7B/6)\alpha]/(B/2) = [(7/6)(1 + \alpha)]/(1/2) < 3$, because $\alpha \leq 1/8$. $\square$

5. BOUNDING THE WEIGHTED STALENESS

We now study *weighted staleness*. That is, each table T has a weight w_T which is multiplied by the overall staleness of table T. These weights reflect the varying priorities of different tables.

THEOREM 5.1. *In the quasiperiodic case, with $\alpha \leq (p - g)/(8t)$ and $\alpha \leq 1/8$, the weighted staleness of algorithm* GROUPANDRUN *is no more than 4.5 times the minimum possible weighted staleness.*

PROOF. For all $i \in A$, define w'_i to equal w_T, where the ith update is an update to table T. Now the total weighted staleness is

$$\sum_T w_T \sum_{\text{batches } B \text{ processed on } T} \text{(staleness charged for } B\text{)} \tag{49}$$

Using the fact that the staleness charged when processing batch B is at most half the square of the flow time of the

first update in batch B, we infer that the total weighted staleness is at most half the sum, over tables T, of w_T times $\sum_{\text{batches } B \text{ processed on table } T}$(flow time of the first update in batch B)^2. This is at most

$$(1/2) \sum_i w_i'(\text{flow time of the } i\text{th update})^2, \quad (50)$$

which is at most

$$(1/2) \sum_i w_i'(3a_i)^2 \quad (51)$$

by Theorem 4.3. This equals $4.5 \sum_i w_i' a_i^2 \leq 4.5 \cdot$ (optimal weighted staleness), since $\sum_i w_i' a_i^2$ is a lower bound on the optimal weighted staleness. $\square$

6. CONCLUSIONS

In this paper, we studied the complexity of scheduling data-loading jobs to minimize the staleness of a real time stream warehouse. We proved that any on-line nonpreemptive algorithm that is never voluntarily idle achieves a constant competitive ratio with respect to the total staleness of all tables in the warehouse, provided that the processors are sufficiently fast. We also showed that stretch and weighted staleness can be bounded under certain conditions on the processor speed and on the arrival times of new data.

One interesting direction for future work is to relax the assumption on processor speed from Section 2, which requires that each α_i be bounded by $\frac{p}{t}$. For instance, we may have a workload consisting of one large table that is very time-consuming to update, and many small tables that can be updated very quickly. Even though the α_i of the large table may be very large (larger than $\frac{p}{t}$), the data warehouse may be able to keep up with the incoming data if the *average* α_i is bounded by $\frac{p}{t}$. We also want to investigate whether replacing MYOPIC in Algorithm GROUPANDRUN by an arbitrary eager algorithm gives an algorithm which guarantees bounded stretch.

7. REFERENCES

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

[2] B. Babcock, S. Babu, M. Datar, and R. Motwani, Chain: Operator Scheduling for Memory Minimization in Data Stream Systems, SIGMOD 2003, 253-264.

[3] N. Bansal, Algorithms for Flow Time Scheduling, Ph.D. Thesis, Carnegie Mellon University, December 2003.

[4] M. Bender, S. Chakrabarti, and S. Muthukrishnan, Flow and Stretch Metrics for Scheduling Continuous Job Streams, SODA 1998, 270-279.

[5] D. Carney, U. Cetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker, Operator Scheduling in a Data Stream Manager, VLDB 2003, 838-849.

[6] J. Cho and H. Garcia-Molina, Synchronizing a Database to Improve Freshness, SIGMOD 2000, 117-128.

[7] L. Golab, T. Johnson, J. S. Seidel and V. Shkapenyuk, Stream Warehousing with DataDepot, SIGMOD 2009, to appear.

[8] L. Golab, T. Johnson, and V. Shkapenyuk, Scheduling Updates in a Real Time Stream Warehouse, ICDE 2009, 1207-1210.

[9] M. Hammad, M. Franklin, W. Aref, and A. Elmagarmid, Scheduling for Shared Window Joins over Data Streams, VLDB 2003, 297-308.

[10] B. Kao and H. Garcia-Molina, An Overview of Real Time Database Systems, *Advances in Real-Time Systems* (ed. S.H. Son), 463-486, Prentice Hall, 1995.

[11] W. Labio, R. Yerneni, and H. Garcia-Molina, Shrinking the Warehouse Update Window, SIGMOD 1999, 383-394.

[12] A. Labrinidis and N. Roussopoulos, Update Propagation Strategies for Improving the Quality of Data on the Web, VLDB 2001, 391-400.

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

[14] M. Sharaf, P. Chrysanthis, A. Labrinidis, and K. Pruhs, Algorithms and Metrics for Processing Multiple Heterogeneous Continuous Queries, *Trans. On Database Sys.*, 33(1) (2008).