Asymmetric Batch Incremental View Maintenance*

Hao He Junyi Xie Jun Yang Hai Yu
Department of Computer Science,
Duke University, Durham, NC 27708, USA
{haohe,junyi,junyang,fishhai}@cs.duke.edu

Abstract

Incremental view maintenance has found a growing number of applications recently, including data warehousing, continuous query processing, publish/subscribe systems, etc. Batch processing of base table modifications, when applicable, can be much more efficient than processing individual modifications one at a time. In this paper, we tackle the problem of finding the most efficient batch incremental maintenance strategy under a refresh response time constraint; that is, at any point in time, the system, upon request, must be able to bring the view up to date within a specified amount of time. The traditional approach is to process all batched modifications relevant to the view whenever the constraint is violated. However, we observe that there often exists natural asymmetry among different components of the maintenance cost; for example, modifications on one base table might be cheaper to process than those on another base table because of some index. We exploit such asymmetries using an unconventional strategy that selectively processes modifications on some base tables while keeping batching others. We present a series of analytical results leading to the development of practical algorithms that approximate an "oracle algorithm" with perfect knowledge of the future. With experiments on a TPC-R database, we demonstrate that our strategy offers substantial performance gains over traditional deferred view maintenance techniques.

1. Introduction

Materialized views [9] have been studied extensively by the database community because of a wide range of traditional applications, such as query processing, data warehousing, caching and replication. Today, view maintenance techniques are also relevant to a number of more recent research areas including streams, continuous query processing, and publish/subscription systems, since all of them in essence deals with the same problem of maintaining *derived data* given a continuous stream of modifications on *base data*. This paper studies the problem of maintaining a materialized view under a *response-time constraint*: That is, at any time, we need to be able to *refresh* a view (i.e., bring its contents up to date with respect to base data) upon request within a specified time limit. Given this response-time constraint, we wish to minimize the cost of view maintenance.

This problem is motivated in part by a publish/subscribe system that we are currently building at Duke University. This system supports a rich subscription language that allows subscribers to define precisely *what* content they want and *when* they want it. Similar language features are also found in OpenCQ [13], NiagraCQ [4], Xyleme [15], and SQL Server Notification Service. Example subscriptions include "tell me the value of my investment portfolio every hour," and "report total gasoline sales in North Carolina if the oil price has changed by more than 10% since the last report." In general, a subscription consists of a *content query* (*what* I want) and a *notification condition* (*when* I want it). Whenever the notification condition is met, the system refreshes the result of the content query and notifies the subscriber of any changes to the result since last notification. In addition, we would like to provide a quality-of-service guarantee to subscribers, which bounds the processing delay when generating notifications.

The result of content query, like a materialized view, can be incrementally maintained when base data is modified. Moreover, the result only needs to be up to date when the notification condition is triggered, and not necessarily after every base data modification. Thus, the system can maintain the result of content query in a *batch incremental* fashion, i.e., a number of modifications are accumulated and processed together as a group. Batch processing is generally more efficient than processing modifications one at a time; the bigger the batch, the more savings it can potentially generate in total maintenance cost. There is plenty of opportunity for batch maintenance of content queries, because while the rate of modification is high (e.g., total gasoline sales figure is constantly changing), the rate of notification may be much lower (e.g., it takes a while for the oil price to change by 10%).

The response-time constraint, however, prevents the system from batching indefinitely. When a view refresh

* This work was supported by a National Science Foundation CAREER Award under grant IIS-0238386.

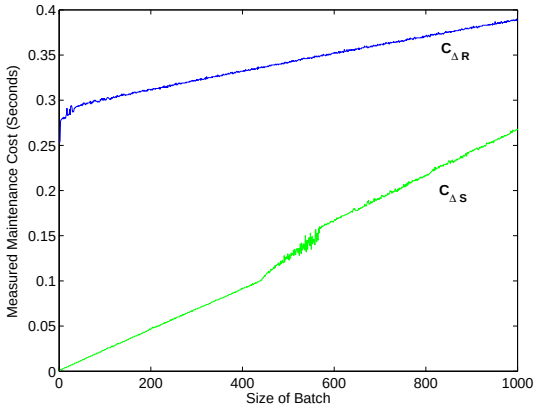

Figure 1. Functions $c_{\triangle R}$ and $c_{\triangle S}$ on a real system.

is needed, if the batch is too big, it may be impossible to process the entire batch within the prescribed response-time constraint. The simplest way to exploit batch maintenance under a response-time constraint is to keep batching modifications affecting a view until the cost of processing the entire batch is about to exceed the response-time constraint; at that point, the system refreshes the view by processing all batched modifications. We call this approach *symmetric batch incremental view maintenance* because it always batch-processes all modifications that have accumulated up to some point. As far as we know, all previous work on batch maintenance of a single view assumes this symmetric approach. However, as we shall see with the following example, this approach is far from optimal.

Example 1 *Consider a subscription content query $R \bowtie S$, where R is indexed on the join attribute while S is not. Suppose that we have been batching database modifications to R and S in $\triangle R$ and $\triangle S$, respectively. Computing the incremental changes to $R \bowtie S$ induced by $\triangle R$ and $\triangle S$ involves, roughly,*[1] *computing $R \bowtie \triangle S$ and $\triangle R \bowtie S$. Because of the index on R, $R \bowtie \triangle S$ can be computed efficiently using an indexed nested-loop join if $\triangle S$ is relatively small; the cost function $c_{\triangle S}$ is linear in $|\triangle S|$. On the other hand, because of the lack of index on S, $\triangle R \bowtie S$ requires scanning the entire S with $\triangle R$ in memory; the cost function $c_{\triangle R}$ starts out to be high for $|\triangle R| = 1$, but increases little with $|\triangle R|$. In Figure 1, we plot these two cost functions for a join query over the TPC-R benchmark database [22] measured on a commercial database system. The horizontal axis shows the size of each batch and the vertical axis shows the cost in running time. The two curves roughly agree with our analysis.*

For simplicity of this example, assume that modifications to R and S arrive at the same rate (the rest of this paper does not make this assumption). Suppose we need to ensure that, whenever a request to refresh $R \bowtie S$ is received, the system is able to complete the refresh operation within 0.35 seconds. With the simple, symmetric approach of refreshing $R \bowtie S$ whenever this response time constraint is reached, the server spends 0.35 seconds of processing time for roughly every 360 modifications (180 in each batch), or 0.97 ms per modification. However, consider an alternative approach where we compute the effect of every modification to S immediately, while batching $\triangle R$ as much as possible. With this alternative approach, we leave the subscription content in an inconsistent state most of the time, but whenever the notification condition is triggered, the server can bring the content into a consistent and up-to-date state in less than 0.35 seconds. In terms of processing costs, the server spends roughly 0.25 ms for each tuple of $\triangle S$; for $\triangle R$, the server spends roughly 0.58 ms per tuple (0.35 seconds every 600 $\triangle R$ tuples, when $c_{\triangle R}$ exceeds 0.35 seconds). Overall, the average cost is only 0.42 per modification, which is substantially lower than the symmetric approach.

Intuitively, we process $\triangle S$ immediately because $c_{\triangle S}$ is roughly linear and hence batching would produce no benefit. Furthermore, processing $\triangle S$ immediately allows the system to defer more of $\triangle R$, which has a lot to gain from batching.

The above example illustrates an asymmetry among different components of the maintenance cost: Some modifications are naturally more amenable to batch processing than others. Such kind of asymmetry arises commonly in practice—not only from availability of different indexes, but also from differences in table sizes, join selectivities, constraints, and operators used in defining a complex subscription content query. To take advantage of this asymmetry, we propose *asymmetric batch incremental maintenance*, which selectively processes modifications on some base tables while keeping batching others. To the best of our knowledge, no previous work on view maintenance or continuous query processing has considered asymmetric batch processing within a single view or query.

The main contributions of this paper can be summarized as follows:

- We formally define the problem of maintaining a materialized view batch-incrementally under a response-time constraint, with the goal of minimizing the total maintenance cost. We demonstrate that asymmetry among components of the total cost leads to asymmetric batch incremental maintenance plans that are more efficient than symmetric ones.
- We show how to reduce the vast search space of all possible asymmetric maintenance plans by considering only *LGM* (*Lazy*, *Greedy*, and *Minimal*) plans. We prove that in general, the best LGM plan is within a factor of two of the optimal plan. Furthermore, for linear cost functions, the best LGM plan is also optimal.
- We propose an optimally efficient A^*-based algorithm to search for the best LGM plan, assuming

1 Additional technicalities are involved in order to avoid the infamous *state bug* [5]; they are omitted here for brevity since they would not affect our argument. For simplicity of presentation, we also assume here that all modifi cations are insertions, though this assumption is easy to drop.

advance knowledge of the modification arrival sequence and the time of view refresh. When the refresh time is not known in advance, we show how to adapt the solution to work with arbitrary refresh time and still achieve provably good performance for linear cost functions. We also provide a heuristic algorithm as a simpler alternative that requires no knowledge of the modification arrival sequence or the view refresh time.

- We present preliminary experimental results using the TPC-R benchmark [22] that validate our proposed algorithms and heuristics.

2. Problem Formulation

Consider the problem of maintaining a materialized view V defined over n base tables $R_1, \ldots, R_n$ over the time period $[0, T]$. At each discrete time step $t \in [0, T]$, any number of base table modifications may arrive; we use an n-vector $\mathbf{d}_t$ to represent the arrivals at t, where the i-th component, $\mathbf{d}_t[i]$, denotes the number of modifications on base table R_i at time t. New modifications are applied immediately to the base tables upon arrival. On the other hand, the content of V does not need to be consistent with the base tables all the time; instead, modifications are appended to *delta tables* $\triangle R_1, \ldots \triangle R_n$ for possible batch processing.

Plan and states. A *maintenance plan* $\mathcal{P}$ is a sequence $\mathbf{p}_0, \ldots, \mathbf{p}_T$, where each $\mathbf{p}_t$ is an n-vector representing the maintenance action (or non-action) taken at time t: For each delta table $\triangle R_i$, we remove the earliest $\mathbf{p}_t[i]$ modifications from $\triangle R_i$ and process them, i.e., compute and apply their resulting incremental changes to the materialized view. If $\mathbf{p}_t = \mathbf{0}$, the plan takes no action at t.

The state of the system also can be characterized by an n-vector whose components denote the sizes of the delta tables $\triangle R_1, \ldots, \triangle R_n$. A zero state vector means that the materialized view is up to date, because no delta table contains any modifications yet to be processed. The *pre-action state* of the system at time t, $\mathbf{s}_t$, denotes the state after the modifications at t have arrived and before any action is taken. The *post-action state*, $\mathbf{s}_{t^+}$, denotes the state after the action at t (if any) has been taken but before the modifications at $t + 1$ arrive. By definition, $\mathbf{s}_{t+1} = \mathbf{s}_{t^+} + \mathbf{d}_{t+1} = \mathbf{s}_t - \mathbf{p}_t + \mathbf{d}_{t+1}$.

Cost functions. We assume that the cost of batch-processing k modifications from delta table $\triangle R_i$ can be expressed by a function $f_i(k)$. As motivated in Section 1, these cost functions may be very different for different delta tables; our approach aggressively exploits this asymmetry. On the other hand, our assumption does ignore cases where certain modifications may be more expensive to process than others even among modifications to the same table. It will be interesting to exploit asymmetry at such a fine granularity as future work, although for this paper we shall concentrate on asymmetry across tables.

We further assume that each cost function $f_i : \mathbb{Z}^+ \to \mathbb{R}$ satisfies the following two properties:

- **Monotonicity**: $f_i(x) \geq f_i(y)$ for any $x \geq y \geq 0$.
- **Subadditivity**: $f_i(0) = 0$, and $f_i(x + y) \leq f_i(x) + f_i(y)$ for any $x \geq 0$ and $y \geq 0$.

Subadditivity captures the benefit of batch processing. Processing $x + y$ modifications cannot be more expensive than processing x of them in one batch and y of them in another; in the worst case, the option of processing them in two different batches is still available when all $x + y$ modifications are given together.

Also note that subadditivity does not necessarily imply concavity. Interestingly, although most subadditive cost functions in practice may be concave, some cost functions that arise in database processing are not. For example, the I/O cost of scanning a table of size x stored compactly on a disk with block size B is $\lceil x/B \rceil$, which is subadditive but not concave because of sudden jumps in value whenever x exceeds a multiple of B.

Response-time constraint. To ensure that an update-to-date view can be made available to users in a timely manner, we require that the maintenance plan never leaves too much modifications unprocessed, so that whenever a refresh is needed, it can be completed within a prescribed cost limit C. This requirement can be formally stated as follows. Suppose the post-action state of the system at time t is $\mathbf{s}_{t^+}$. The cost of refreshing the view in this state is the sum of the costs of processing all modifications in all delta tables, i.e., $\sum_{i=1}^{n} f_i(\mathbf{s}_{t^+}[i])$. As a shorthand, we define $f(\mathbf{v}) = \sum_{i=1}^{n} f_i(\mathbf{v}[i])$, where $\mathbf{v}$ is an n-vector with non-negative components. Thus, the response-time constraint can be expressed as:

$$f(\mathbf{s}_{t^+}) \leq C, \text{for all } t.$$

Note that most database systems today lack the support for hard real-time performance guarantees. However, this limitation does not diminish our contribution, because these systems can still use our techniques to meet soft response-time constraints in a best-effort manner.

Valid maintenance plan. Suppose that view is refreshed at time T. We now define what a *valid* maintenance plan is: It must meet the response-time constraint during $[0, T)$ and completely empty all delta tables at time T. An additional technicality is that the plan cannot remove more modifications than what have been accumulated. A formal definition is given below.

Definition 1 *Consider plan $\mathcal{P} = \mathbf{p}_0, \ldots, \mathbf{p}_T$ over time period $[0, T]$ with view refreshed at time T. Let $\mathbf{s}_t$ denote the pre-action state of the system at time t. An action $\mathbf{p}_t$ ($t < T$) is* valid *if it meets the following criteria:*

- *For all $i \in [1, n]$, $0 \leq \mathbf{p}_t[i] \leq \mathbf{s}_t[i]$.*
- *$f(\mathbf{s}_t - \mathbf{p}_t) \leq C$.*

$\mathcal{P}$ is valid *if all its actions are valid, and $\mathbf{p}_T = \mathbf{s}_T$.*

We say that a state $\mathbf{s}$ of the system is *full* if the cost of refreshing the view in this state exceeds the prescribed limit, i.e., $f(\mathbf{s}) > C$. The response-time constraint basically requires all post-action states to be not full. Therefore, if the pre-action state at time t becomes full (because of the new modifications arriving at t), then a valid plan must take an action at t (i.e., $\mathbf{p}_t \neq \mathbf{0}$) so that the post-action state is not full.

Problem statement. Given a view defined over base tables $R_1, \ldots, R_n$, functions $f_1, \ldots, f_n$ that measure the costs of processing modifications to these base tables, a modification arrival sequence $\mathbf{d}_0, \ldots, \mathbf{d}_T$ over the time period $[0, T]$, and a response-time constraint $C \geq 0$, we want to find a valid maintenance plan $\mathcal{P} = \mathbf{p}_0, \ldots, \mathbf{p}_T$, with view refreshed at time T, such that the total maintenance cost incurred by $\mathcal{P}$, given below, is minimized:

$$\sum_{t=0}^{T} f(\mathbf{p}_t) = \sum_{t=0}^{T} \sum_{i=1}^{n} f_i(\mathbf{p}_t[i]).$$

As a shorthand, we define $f(\mathcal{P}) = \sum_{t=0}^{T} f(\mathbf{p}_t)$.

In practice, the cost functions can be provided a database optimizer, or measured by experiments or from past experience. The modification arrival sequence and the time of refresh can be projected using past observations. Section 4 discusses how to handle the case when the refresh time is unknown, and Section 5 experiments with both unknown refresh time and non-uniform arrival sequence.

3. Reducing the Search Space

Naively, one needs to search among all valid plans in order to find an optimal maintenance plan. However, the space of all valid plans is prohibitively large, which renders any exhaustive search algorithm impractical. In this section, we show that, instead of looking at all valid plans, one only has to consider a significantly smaller subset of plans in order to find an optimal or near-optimal plan.

3.1. Lazy Plans

Recall from Section 2 that a valid plan must take an action if the pre-action state is full, in order to meet the response-time constraint. On the other hand, if the pre-action state is not full, a valid plan may still take an action; such possibilities are a major factor contributing to the large size of the plan space. Thus, our first step towards reducing this space is to consider *lazy* plans, which only take actions when forced to, i.e., when pre-action states are full.

Definition 2 *An action at time t is* lazy *if the pre-action state at t is full. A plan $\mathcal{P} = \mathbf{p}_0, \ldots, \mathbf{p}_T$ is* lazy *if for all $t \in [0, T)$, $\mathbf{p}_t$ is either lazy or $\mathbf{0}$.*

The next lemma shows that any non-lazy maintenance plan can be transformed into a lazy plan without increasing its cost. Therefore, the best lazy plan is also a globally optimal plan.

Lemma 1 *For any valid maintenance plan $\mathcal{P}$, there exists a valid lazy plan $\mathcal{Q}$ such that $f(\mathcal{Q}) \leq f(\mathcal{P})$.*

Proof: Using the following procedure, we construct $\mathcal{Q} = \mathbf{q}_0, \ldots, \mathbf{q}_T$ from $\mathcal{P} = \mathbf{p}_0, \ldots, \mathbf{p}_T$.

```
MAKELAZYPLAN(p_0, ..., p_T)
 1: p ← 0; {used to accumulate actions in P}
 2: for t = 0 to T do
 3:    p ← p + p_t;
 4:    s_t ← pre-action state under plan Q at t;
 5:    if f(s_t) > C or t = T then {action forced}
 6:       q_t ← p; {apply all accumulated actions in P}
 7:       p ← 0;
 8:    else {action unnecessary}
 9:       q_t ← 0;
10: return q_0, ..., q_T;
```

It is very easy to verify that $\mathcal{Q} = \text{MAKELAZYPLAN}(\mathcal{P})$ is lazy. To see that $\mathcal{Q}$ is also valid, note that whenever an action is forced, say at time t, $\mathcal{Q}$ basically applies all accumulated actions from $\mathcal{P}$ up to t; therefore, the resulting post-action state at t will be identical to that under $\mathcal{P}$ at the same time instant, which satisfies the response-time constraint because $\mathcal{P}$ is valid.

Next, we show that $f(\mathcal{Q}) \leq f(\mathcal{P})$. To this end, note that each non-zero $\mathbf{q}_t$ is the sum of one more $\mathbf{p}_t$'s. Because all cost functions are subadditive, the cost of the combined $\mathcal{Q}$ action must be no greater than the sum of costs of individual $\mathcal{P}$ actions. □

3.2. Lazy, Greedy, and Minimal Plans

Lemma 1 shows that we only have to consider lazy plans when looking for an optimal plan. However, the space of lazy plans is still very large. When an action is needed, a lazy plan has a lot of freedom in choosing the amount of processing to do to make the post-action state non-full: It may process a single modification from one delta table just to get the post-action state below full; it may process all modifications in all delta tables; or it may process a moderate amount between these two extremes. In this section, we consider more restricted forms of plans in order to reduce the search space even further. This time, however, we will have to sacrifice the global optimality of the solution in the general case.

Definition 3 *A plan $\mathcal{P} = \mathbf{p}_0, \ldots, \mathbf{p}_T$ is* LGM *(*Lazy, Greedy, Minimal*) if it has the following properties:*

(Laziness) *$\mathcal{P}$ is lazy.*

(Greediness) *All actions of $\mathcal{P}$ are* greedy*, i.e., each action processes either all modifications accumulated in a delta table or none at all. Formally, for all $t \in [0, T]$ and $i \in [1, n]$, either $\mathbf{p}_t[i] = \mathbf{s}_t[i]$ or $\mathbf{p}_t[i] = 0$, where $\mathbf{s}_t$ is the pre-action state at time t.*

(Minimality) *All actions except* $\mathbf{p}_T$ *are* minimal, *i.e., for each action* $\mathbf{p}_t$ *where* $t < T$*, it is impossible to set any non-zero component of* $\mathbf{p}_t$ *to* 0 *and still have it satisfy* $f(\mathbf{s}_t - \mathbf{p}_t) \leq C$.

Intuitively, greediness means that each action $\mathbf{p}_t$ completely empties a set of delta tables and leaves others alone; minimality means that we cannot meet the response-time constraint by emptying just a proper subset of the delta tables emptied by $\mathbf{p}_t$.

Let OPT be the cost of a globally optimal plan, and $\mathrm{OPT}^{\mathrm{LGM}}$ be the cost of the best LGM plan. In general, one can no longer hope that the best LGM plan is also globally optimal. Fortunately, it turns out that their costs only differ by a constant factor. Indeed, in the remainder of this section, we will show that $\mathrm{OPT}^{\mathrm{LGM}} \leq 2 \cdot \mathrm{OPT}$. To this end, let $\mathcal{P}$ be any valid plan. We shall construct an LGM plan $\mathcal{Q}$ whose cost is bounded by twice that of $\mathcal{P}$.

The following procedure describes how to construct $\mathcal{Q} = \mathbf{q}_0, \ldots, \mathbf{q}_T$ from $\mathcal{P} = \mathbf{p}_0, \ldots, \mathbf{p}_T$. The procedure uses an auxiliary function MINIMIZEACTION$(\mathbf{q}, \mathbf{s})$, which takes an action $\mathbf{q}$ and a pre-action state $\mathbf{s}$ as input, and returns a minimal action $\mathbf{q}'$ that empties a subset of the delta tables emptied by $\mathbf{q}$ while still satisfying $f(\mathbf{s} - \mathbf{q}') \leq C$.

```
MAKELGMPLAN(p_0, ..., p_T)
 1: for t = 0 to T − 1 do
 2:    s_{Q_t} ← pre-action state under plan Q at t;
 3:    s_{P_{t+}} ← post-action state under plan P at t;
 4:    if f(s_{Q_t}) > C then {action forced}
 5:       for i = 1 to n do {decide for each delta table}
 6:          if s_{Q_t}[i] > s_{P_{t+}}[i] then
 7:             q'_t[i] ← s_{Q_t}[i];
 8:          else
 9:             q'_t[i] ← 0;
10:       q_t ← MINIMIZEACTION(q'_t, s_{Q_t});
11:    else {action unnecessary}
12:       q_t ← 0;
13: q_T ← pre-action state under plan Q at T;
14: return q_0, ..., q_T;
```

Lemma 2 *Plan* $\mathcal{Q} = \text{MAKELGMPLAN}(\mathcal{P})$ *is valid and LGM.*

Proof: It is easy to verify that our construction guarantees that $\mathcal{Q}$ is LGM. We now show that $\mathcal{Q}$ is valid. Consider the temporary action $\mathbf{q}'_t$ ($t < T$) computed by the loop on Line 5 of MAKELGMPLAN. Obviously, for all $i \in [1, n]$, $0 \leq \mathbf{q}'_t[i] \leq \mathbf{s}_{\mathcal{Q}t}[i]$, i.e., this action does not remove more modifications than what have been accumulated. Furthermore, regardless of whether $\mathbf{s}_{\mathcal{Q}t}[i] > \mathbf{s}_{\mathcal{P}t^+}[i]$, we have $\mathbf{s}_{\mathcal{Q}t}[i] - \mathbf{q}'_t[i] \leq \mathbf{s}_{\mathcal{P}t^+}[i]$ by construction of $\mathbf{q}'_t$. Thus, by monotonicity of the cost functions and the fact that $\mathcal{P}$ is valid, we have

$$f(\mathbf{s}_{\mathcal{Q}t} - \mathbf{q}'_t) \leq f(\mathbf{s}_{\mathcal{P}t^+}) \leq C.$$

In other words, action $\mathbf{q}'_t$ also meets the response-time constraint and is therefore valid. Since MINIMIZEACTION does not make a plan invalid, $\mathbf{q}_t$ is valid too. □

Recall that our goal is to prove that the cost of $\mathcal{Q}$ is within twice the cost of $\mathcal{P}$. As the total cost of a plan is the sum of the costs paid for processing modifications from each base table, we only need to prove the (stronger) claim that for each base table R_i, the cost of processing all modifications from R_i in $\mathcal{Q}$ is within twice the total such cost in $\mathcal{P}$. In subsequent discussion, we fix a base table R_i, and let $\mathcal{P}(i) = \{(t, \mathbf{p}_t[i]) \mid t \in [0, T] \wedge \mathbf{p}_t[i] \neq 0\}$ be a set of pairs representing actions in $\mathcal{P}$ restricted to modifications from the single base table R_i. We similarly define $\mathcal{Q}(i) = \{(t, \mathbf{q}_t[i]) \mid t \in [0, T] \wedge \mathbf{q}_t[i] \neq 0\}$.

We define a bipartite graph $G = (V_{\mathcal{P}(i)}, V_{\mathcal{Q}(i)}, E)$, where each node in $V_{\mathcal{P}(i)}$ represents an element in $\mathcal{P}(i)$, each node in $V_{\mathcal{Q}(i)}$ represents an element in $\mathcal{Q}(i)$, and E is the set of edges to be specified shortly. For ease of exposition, we will not distinguish between a node in G and an element in $\mathcal{P}(i)$ or $\mathcal{Q}(i)$ any more.

Without loss of generality, assume that for every delta table the modifications are processed in FIFO order; that is, modifications that arrive earlier are processed no later than those that arrive later. This assumption does not affect the generality of our analysis because we assume that the total cost of processing a set of modifications depends only on the size of the set, and not on individual modifications in the set. For any node x, let $U(x)$ be the set of modifications processed by x. The edges in G are defined by the following rule: Two nodes $x_1 \in V_{\mathcal{P}(i)}$ and $x_2 \in V_{\mathcal{Q}(i)}$ are connected by an edge if and only if

$$U(x_1) \cap U(x_2) \neq \varnothing.$$

That is, they process some modification in common. Figure 3.2 shows an example of the graph $G = (V_{\mathcal{P}(i)}, V_{\mathcal{Q}(i)}, E)$. Recall that each node is a pair (t, k), where t is time of the action and k is the number of R_i modifications processed by the action. Nodes are listed from left to right in increasing order of the action time.

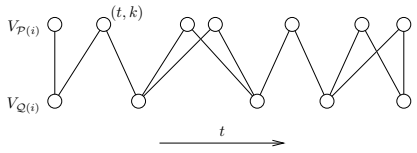

Figure 2. $G = (V_{\mathcal{P}(i)}, V_{\mathcal{Q}(i)}, E)$.

Lemma 3 *In the graph* $G = (V_{\mathcal{P}(i)}, V_{\mathcal{Q}(i)}, E)$*, the degree of each node in* $V_{\mathcal{P}(i)}$ *is at most* 2.

Proof: Consider a node $x = (t, k) \in V_{\mathcal{P}(i)}$. Its degree is the number of neighboring nodes in $V_{\mathcal{Q}(i)}$ connected to x. We are going to show that x has at most one neighbor to its right (with an action time no less than t), and at most one neighbor to its left (with an action time less than t).

Case 1: neighbors to the right. Let $x' = (t', k') \in V_{\mathcal{Q}(i)}$ be the first neighbor of x to its right ($t' \geq t$), if any. Since $\mathcal{Q}$ is greedy, x' processes all modifications accumulated up to t'. Thus, any subsequent node x'' in $V_{\mathcal{Q}(i)}$ can process only modifications that arrive strictly after t'. On the other hand, it is impossible for x to process any modifications that arrive strictly after t, because x happens at t. Since $t \leq t'$, x and x'' cannot process any modification in common. Hence there is at most one neighbor of x to its right.

Case 2: neighbors to the left. Suppose x has at least two neighbors in $V_{\mathcal{Q}(i)}$, $x'' = (t'', k'')$ and $x' = (t', k')$, with $t'' < t' < t$. Let t_u be the arrival time of a modification u that is processed by both x and x''. Obviously, $t_u \leq t''$ because x'' cannot process something that has not yet arrived. The post-action state of $\triangle R_i$ under $\mathcal{P}$ at t', $\mathbf{s}_{\mathcal{P}t'^+}[i]$, must contain all modifications that arrive during $(t'', t']$: Since they arrive later than u, in FIFO order, they must be processed together with or after u, and therefore can only be processed by x at t or by a later action. On the other hand, the pre-action state of $\triangle R_i$ under $\mathcal{Q}$ at t', $\mathbf{s}_{\mathcal{Q}t'}[i]$, contains only modifications that arrive during $(t'', t']$: Because $\mathcal{Q}$ is greedy, x'' has processed all modifications accumulated up to t''. Comparing the two states, we have $\mathbf{s}_{\mathcal{P}t'^+}[i] \geq \mathbf{s}_{\mathcal{Q}t'}[i]$. However, by construction (Lines 6–7 of MAKELGMPLAN), $\mathcal{Q}$ only process R_i modifications at t' if $\mathbf{s}_{\mathcal{Q}t'}[i] > \mathbf{s}_{\mathcal{P}t'^+}[i]$, a contradiction.

Combining the two cases above, n can have at most 2 neighbors. □

Lemma 4 *For any node $x \in V_{\mathcal{Q}(i)}$,*

$$f_i(x) \leq \sum_{y \in N(x)} f_i(y),$$

where $N(x)$ denotes the set of neighbors of x in $V_{\mathcal{P}(i)}$, and $f_i(x) = f_i(k)$ is the cost of action $x = (t, k)$.

Proof: Let $x = (t, k)$ and $N(x) = \{(t_1, k_1), \ldots, (t_m, k_m)\}$. By construction of G, it is easy to see that $k \leq \sum_{j=1}^{m} k_j$. Because f_i is monotone and subadditive, we have

$$f_i(k) \leq f_i(\sum_{j=1}^{m} k_j) \leq \sum_{j=1}^{m} f_i(k_j),$$

which proves the lemma. □

We are now ready to state and prove the main theorem of this section.

Theorem 1 $\mathrm{OPT}^{\mathrm{LGM}} \leq 2 \cdot \mathrm{OPT}$.

Proof: Let $\mathcal{P}$ be a globally optimal maintenance plan, and $\mathcal{Q}$ the LGM plan returned by MAKELGMPLAN($\mathcal{P}$). For each R_i, we construct the bipartite graph $G(V_{\mathcal{P}(i)}, V_{\mathcal{Q}(i)}, E)$ as described above. By Lemma 4, the total cost of $\mathcal{Q}(i)$ is:

$$\sum_{x \in V_{\mathcal{Q}(i)}} f_i(n) \leq \sum_{x \in V_{\mathcal{Q}(i)}} \sum_{y \in N(x)} f_i(y).$$

By Lemma 3, the degree of each node $y \in V_{\mathcal{P}(i)}$ is at most 2. Therefore, the right-hand side of the above inequality is bounded by $2 \cdot \sum_{y \in V_{\mathcal{P}(i)}} f_i(y)$. Hence we can write:

$$\sum_{x \in V_{\mathcal{Q}(i)}} f_i(n) \leq 2 \cdot \sum_{y \in V_{\mathcal{P}(i)}} f_i(y).$$

The total cost of a plan can be calculated by the sum of the cost of processing each base table R_i. We complete the proof by summing up the above inequality for $i = 1, \ldots, n$: $\mathrm{OPT}^{\mathrm{LGM}} \leq \sum_{i=1}^{n} \sum_{x \in V_{\mathcal{Q}(i)}} f_i(x) \leq 2 \cdot \sum_{i=1}^{n} \sum_{y \in V_{\mathcal{P}(i)}} f_i(y) = 2 \cdot \mathrm{OPT}$. □

Next, we show that Theorem 1 is tight by constructing an instance of the problem for which $\mathrm{OPT}^{\mathrm{LGM}} \geq (2 - \varepsilon) \cdot \mathrm{OPT}$ for any $\varepsilon > 0$. The example instance involves a subadditive but non-concave cost function. It would be interesting to investigate as future work whether Theorem 1 can be improved under more stringent assumptions, e.g., when all cost functions are concave and modifications always arrive one by one at discrete time steps.

Example 2 *Let $T = 2m - 1$ for some $m \in \mathbb{N}$. Without loss of generality, assume that $1/\varepsilon$ is an integer. Consider one single base table R. The cost function f for processing modifications on R is defined as follows:*

$$f(x) = \begin{cases} (\varepsilon x/2) \cdot C & 0 \leq x \leq 2/\varepsilon \\ (1 + \varepsilon/2) \cdot C & x > 2/\varepsilon. \end{cases}$$

Note that f is monotone and subadditive.

Suppose $2/\varepsilon + 1$ modifications on R arrive at each time step $0, 1, \cdots, 2m - 1$. The cost of processing all new arrivals at each time step is thus $f(2/\varepsilon + 1) = (1 + \varepsilon/2) \cdot C > C$. Therefore, an LGM plan is forced to process all new arrivals at each time step, yielding a total cost of $\mathrm{OPT}^{\mathrm{LGM}} = (2 + \varepsilon)m \cdot C$. On the other hand, a non-LGM plan can process one modification and leave the other $2/\varepsilon$ modifications. The cost of processing the post-action state is $f(2/\varepsilon) = C$, still within the constraint. In the next time step, the plan processes the $2/\varepsilon$ old modifications and $2/\varepsilon + 1$ new ones together. Overall, $\mathrm{OPT} \leq (f(1) + f(4/\varepsilon + 1))m = (1 + \varepsilon)m \cdot C$. Consequently, $\mathrm{OPT}^{\mathrm{LGM}} \geq (2 - \varepsilon)\mathrm{OPT}$.

3.3. Linear Cost Functions

In practice, many cost functions often exhibit as a linear form $f(k) = a \cdot k + b$, where $b \geq 0$ and $a > 0$. In Section 5 we show by experiments that such linear cost functions indeed arise in real systems. A linear cost function can be interpreted as a fixed initial processing cost b followed by a per-modification processing cost a incurred for each modification processed. In practice, b reflects the cost of pre-processing, e.g., parsing, optimization, setting up execution, sorting or constructing hash tables for joining base tables, or loading parts of the index structures. Once the pre-precessing step is done, the cost is roughly a constant a for each modification. Clearly, linear cost functions are both monotone and subadditive.

In this section, we show that for linear cost functions, the best LGM plan is actually globally optimal. Suppose that $f_i(k) = a_i \cdot k + b_i$ for all $i \in [1, n]$. For any plan $\mathcal{P}$, the total cost spent on processing modifications from R_i can be written as $a_i K_i + b_i |\mathcal{P}(i)|$, where K_i is the total number of modifications from R_i arrived over the entire period, and $|\mathcal{P}(i)|$ is the total number of actions $\mathcal{P}$ has ever taken on modifications from R_i. Since K_i is fixed for a given problem instance, $|\mathcal{P}(i)|$ is the decisive factor in $\mathcal{P}$'s cost. In particular, a plan $\mathcal{P}$ is optimal if and only if it minimizes $\sum_{i=1}^{n} b_i |\mathcal{P}(i)|$.

Theorem 2 *If for all $i \in [1, n]$, $f_i(k) = a_i \cdot k + b_i$, where $b_i \geq 0$ and $a_i > 0$, then $\mathrm{OPT}^{\mathrm{LGM}} = \mathrm{OPT}$.*

Proof: Based on the discussion above, given an optimal plan $\mathcal{P}$, if we can construct an LGM plan $\mathcal{Q}$ such that for each R_i, $|\mathcal{Q}(i)| \leq |\mathcal{P}(i)|$, then $\mathcal{Q}$ would also be optimal and hence $\mathrm{OPT}^{\mathrm{LGM}} = \mathrm{OPT}$. The plan returned by MAKELGMPLAN($\mathcal{P}$) serves our purpose. In fact, let $G = (V_{\mathcal{P}(i)}, V_{\mathcal{Q}(i)}, E)$ be the graph defined as in the previous section. We can show that $|V_{\mathcal{P}(i)}| \geq |V_{\mathcal{Q}(i)}|$, which is an easy consequence of Lemmas 5 and 6 below and the observation that the first action in $\mathcal{Q}(i)$ occurs no earlier than the first action in $\mathcal{P}(i)$. This observation follows from the construction of $\mathcal{Q}$: $\mathcal{Q}$ would not act on $\triangle R_i$ unless the pre-action state of $\triangle R_i$ under $\mathcal{Q}$ is strictly larger than the post-action state of $\triangle R_i$ under $\mathcal{P}$, but unless $\mathcal{P}$ has already taken an action on $\triangle R_i$, the two states will be identical. □

Lemma 5 *For any two nodes $x_1, x_2 \in V_{\mathcal{P}(i)}$ representing two consecutive actions of $\mathcal{P}$ on $\triangle R_i$ with action times $t_1 < t_2$, there exists at most one node $y \in V_{\mathcal{Q}(i)}$ with action time t such that $t_1 \leq t < t_2$.*

Proof: We prove by contradiction. Suppose there exist $y, y' \in V_{\mathcal{Q}(i)}$ with action times t and t' such that $t_1 \leq t < t' < t_2$. The post-action state of $\triangle R_i$ under $\mathcal{P}$ at time t' must contain all modifications on R_i arrived during $(t_1, t']$, because x_1 at t_1 cannot process any modification that has not yet arrived, and there is no action in $\mathcal{P}(i)$ between x_1 and x_2. On the other hand, the pre-action state of $\triangle R_i$ under $\mathcal{Q}$ at time t' only contains modifications on R_i arrived during $(t, t']$, because y has greedily processed all modifications accumulated up to t. Comparing the two states, we have $\mathbf{s}_{\mathcal{P}t'^+}[i] \geq \mathbf{s}_{\mathcal{Q}t'}[i]$. However, by construction $\mathcal{Q}$ only process R_i modifications at t' if $\mathbf{s}_{\mathcal{Q}t'}[i] > \mathbf{s}_{\mathcal{P}t'^+}[i]$, a contradiction. □

Lemma 6 *Let $x \in V_{\mathcal{P}(i)}$ be the last action of $\mathcal{P}$ on $\triangle R_i$ with action time t. There exists at most one action $x' \in V_{\mathcal{Q}(i)}$ with action time $t' \geq t$.*

Proof: If $t = T$, then obviously there can be at most one action in $V_{\mathcal{Q}(i)}$ with action time of T. If $t < T$, the fact that x is the last action in $V_{\mathcal{P}(i)}$ implies that no modifications on R_i arrive after t. Since $\mathcal{Q}$ is greedy, x' will process all modifications arrived at or before $t' \geq t$, so there will be no new modifications for a subsequent action to process. □

Lemmas 5 and 6 do not depend on any property of the cost function, so they apply to any cost functions.

4. Algorithms

Depending on whether we have knowledge about the modification arrival sequence and the view refresh time T, we propose different algorithms: Section 4.1 discusses how to model the space of LGM plans as a graph and search this graph efficiently, assuming perfect knowledge of the modification sequence and T. Sections 4.2 and 4.3 present two algorithms that relax these assumptions.

4.1. An A^* Algorithm

Plan space as a graph. We model the space of all LGM plans as a weighted directed acyclic graph G. Each node represents a *possible* post-action state of the system after an action ($\mathbf{q}_t \neq \mathbf{0}$) is taken at time $t \in [0, T]$. By "possible," we mean that this state is reachable if we follow a certain valid LGM plan from time 0 to time t. We annotate the node by t and the post-action state. Every possible post-action state following an action has a unique representative node in G. In addition, there are two special nodes in G: *Source* is the only node with special timestamp -1 representing the initial state of the system before any modifications arrive at time 0; *destination* is the only node with timestamp T representing a post-action state equal to $\mathbf{0}$, i.e., all delta tables are empty and the view is refreshed.

We determine the set of all directed edges in G as follows. Each node x with timestamp $t_1 < T$ representing state $\mathbf{s}$ has one or more outgoing edges to nodes with timestamp t_2, which is the first time after t_1 when the pre-action state becomes full because of the new modifications arriving during $[t_1 + 1, t_2]$. More precisely, $t_2 = \min\{t' \mid f(\mathbf{s} + \sum_{t=t_1+1}^{t'} \mathbf{d}_t) > C\}$, where $\mathbf{d}_t$, defined in Section 2, is a vector representing the modifications arriving at t. Each edge from x represents a possible greedy, minimal, and valid action $\mathbf{q}$ at t_2 given the pre-action state $\mathbf{s} + \sum_{t=t_1+1}^{t_2} \mathbf{d}_t$. We assign the edge a weight of $f(\mathbf{q})$, and we annotate the node that this edge points to by t_2 and $\mathbf{s} + \sum_{t=t_1+1}^{t_2} \mathbf{d}_t - \mathbf{q}$. As a special case, if $t_2 = T$ or the pre-action state at T is still not full, then x has a single outgoing edge to *destination* with weight $f(\mathbf{s} + \sum_{t=t_1+1}^{T} \mathbf{d}_t)$, representing the action of refreshing view at T.

Figure 4.1 shows an example of the graph G. It is easy to see that every path from *source* to *destination* corresponds to an LGM plan, and vice versa. The cost of the plan is the sum of all edge weights on the corresponding path. Therefore, searching for the best LGM plan amounts to finding a shortest path (in terms of total edge weight) from *source* to *destination*.

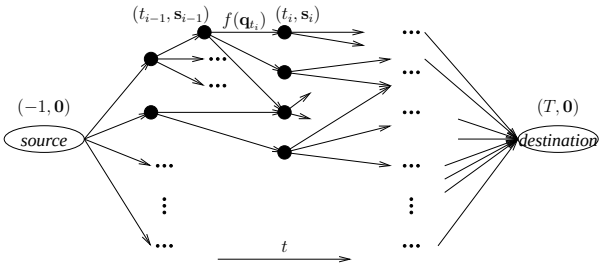

Figure 3. Graph modeling the space of all LGM plans.

Theorem 3 *Each shortest path between* source *and* destination *in G corresponds to an LGM plan with the lowest possible cost. Suppose that the nodes on this path are annotated by the sequence of (timestamp, state) pairs $(t_0, \mathbf{s}_0), (t_1, \mathbf{s}_1), \ldots, (t_m, \mathbf{s}_m)$, where $-1 = t_0 < t_1 < \cdots < t_m = T$. The corresponding LGM plan $\mathcal{Q}$ is specified by:*

- $\mathbf{q}_t = \mathbf{0}$ *if* $t \notin \{t_1, \ldots, t_m\}$.
- $\mathbf{q}_{t_i} = \mathbf{s}_{t_{i-1}} + \sum_{t=t_{i-1}+1}^{t_i} \mathbf{d}_t - \mathbf{s}_{t_i}$ *for* $i = 1, \ldots, m$.

$\mathrm{OPT}^{\mathrm{LGM}}$ *is equal to the sum of all edge weights on the shortest path.*

It should be apparent now why the LGM properties are effective in reducing the search place. By ensuring that actions are spaced out in time, the laziness property decreases out-degrees of nodes and lengths of paths to *destination*. Greediness and minimality properties further decrease out-degrees of nodes by limiting the number of possible actions in each state. The graph would have been significantly larger had all valid plans been considered instead.

Searching the graph. Although we have significantly reduced the search space, constructing and exhaustively searching the entire graph still can be very expensive. We now present an efficient A^* algorithm that can find the optimal LGM plan without constructing or exploring the entire graph.

The algorithm performs an informed best-first search through the space, constructing parts of the graph on demand only when necessary. Specifically, we maintain a priority Q containing all nodes we have seen met but not expanded yet. Nodes in Q are ordered by an evaluation function $d(x) = g(x) + h(x)$ where $g(x)$ is the minimum path cost from *source* to node x and $h(x)$ is a heuristic estimate of the cost from x to *destination*.

We define $h(x)$ as follows. Suppose that x has timestamp t and post-action state $\mathbf{s}$. We can compute b_i, the maximum number of R_i modifications that can be processed in one batch under the response-time constraint, as $b_i = m_i + \max\{b \mid f_i(b) \leq C\}$, where m_i is the maximum number of R_i modification that can arrive together in one time step. Let K_i be the total number of R_i modifications arrived during $(t, T]$. Then,

$$h(x) = \sum_{i=1}^{n} \left\lfloor \frac{\mathbf{s}[i] + K_i}{b_i} \right\rfloor f_i(b_i).$$

Intuitively, $h(x)$ computes a conservative estimate of the cost of processing all remaining R_i modifications. This estimate is conservative because for each R_i, the calculation ignores modifications on other base tables, which can make states full faster and force more frequent actions. Formally, we can show that $h(x)$ is *consistent*:

Lemma 7 *The heuristic $h(x)$ is* consistent*, i.e., for every node x and every successor x' of x generated by action $\mathbf{q}$, $h(x) \leq f(\mathbf{q}) + h(x')$.*

Proof: Omitted. □

A consistent heuristic is also *admissible*, i.e., it never overestimates the cost to reach *destination*. An admissible heuristic guarantees that the A^* search is optimal, and a consistent heuristic further guarantees that the optimal path to any node is always the first one followed by the A^* search [20]. Therefore, in the algorithm presented below, any node that has been visited before does not need to be expanded again.

A^*SEARCH

```
1:  g(source) = 0;
2:  d(source) = h(source);
3:  Q ← {source};
4:  while Q ≠ ∅ do
5:      x ← DEQUEUE(Q);
6:      if x = destination then
7:          return g(x);
8:      for each (x', q) ∈ EXPAND(x) do
9:          if x' has been expanded before then
10:             continue;
11:         g(x') ← min{g(x'), g(x) + f(q)};
12:         d(x') ← g(x') + h(x');
13:         ENQUEUE(Q, x');
```

In the above, EXPAND(x) returns the set $\{(x', \mathbf{q})\}$ where x' can be reached from x by an LGM action $\mathbf{q}$. An exception to the laziness requirement is when $x' =$ *destination*, in which case $\mathbf{q}$ must process all remaining modifications. Conceptually, we set $g(x)$ initially to $+\infty$ for all nodes; in reality, $g(x)$ and $d(x)$ only need to be maintained for nodes in Q. The algorithm returns the cost of the optimal LGM plan. To obtain the plan itself, during the course of A^*SEARCH, we can maintain a backward pointer from each node x to its predecessor node on the shortest path from *source* to x; then, we can reconstruct the plan by following backward pointers from *destination*.

Finally, we note that the A^* algorithm is *optimally efficient* for the given heuristic; that is, no other optimal algorithm is guaranteed to expand fewer nodes than it [20].

4.2. Adapting to Unknown Refresh Time

So far, our discussion has been based on the assumption that we know the time T when the view needs to be refreshed. In practice, the refresh time is usually not known in advance, which is also the reason why we enforce the response-time constraint at all times. In this section, we describe a way to adapt the solution found for a

specific refresh time T_0 so that it works for any arbitrary refresh time. This approach allows us to bound the cost of the resulting plan in terms of the optimal plan cost for linear functions.

Suppose that an estimation of the refresh time, T_0, is given to us in advance. We find an optimal LGM plan $\mathcal{Q}_{T_0}$ for the interval $[0, T_0]$ as described in the previous section. At runtime, we simply execute $\mathcal{Q}_{T_0}$. Suppose the view is actually refreshed at time T. If T happens to be T_0, we have executed the optimal LGM plan. Otherwise, there are two cases. If $T < T_0$, we simply stop executing $\mathcal{Q}_{T_0}$ and process all remaining modifications at time T. If $T > T_0$, we execute $\mathcal{Q}_{T_0}$ repeatedly until T, at which point we process all remaining modifications.

Note that this approach still requires knowledge of the modification arrival sequence to compute $\mathcal{Q}_{T_0}$. In the case of $T > T_0$, we must also assume that the arrival sequence is periodic with period T_0 in order to guarantee the quality of the adapted plan. In practice, if the sequence exhibits a periodic pattern, T_0 should be chosen as a multiple of the period. In Section 4.3, we will provide an alternative algorithm that requires neither knowledge of the refresh time nor the precise modification arrival sequence.

Let $\mathcal{Q}_{T_0,T}$ be the maintenance plan obtained by adapting $\mathcal{Q}_{T_0}$ (an LGM plan optimized for T_0) for a refresh time of T as described above. Note that $\mathcal{Q}_{T_0,T}$ may not be LGM. Let OPT_T be the cost of an optimal plan over the time interval $[0, T]$. We conjecture the cost of $\mathcal{Q}_{T_0,T}$ is close to $2 \cdot \mathrm{OPT}_T$. While we are unable to prove this conjecture in general, we can obtain tight bounds if all cost functions are linear.

Theorem 4 *Suppose $f_i(k) = a_i \cdot k + b_i$ for all $i \in [1, n]$. If $T < T_0$, the cost of $\mathcal{Q}_{T_0,T}$ is upper-bounded by $\mathrm{OPT}_T + \sum_{i=1}^{n} b_i$. If $T > T_0$, the cost of $\mathcal{Q}_{T_0,T}$ is upper-bounded by $\mathrm{OPT}_T + \lceil T/T_0 \rceil \cdot \sum_{i=1}^{n} b_i$.*

Proof: Here we concentrate on the case of $T < T_0$. The proof for $T > T_0$ is similar and thus omitted for brevity.

To facilitate our discussion, for a maintenance plan $\mathcal{P}$ and a time interval $[t_1, t_2]$, we use the notation $\mathcal{P}[t_1, t_2]$ to denote $\mathcal{P}$'s actions during $[t_1, t_2]$, and use $\mathcal{P}[t_1, t_2](i)$ to represent $\mathcal{P}$'s actions on R_i modifications during $[t_1, t_2]$, i.e., $\mathcal{P}[t_1, t_2](i) = \{(t, \mathbf{p}_t[i]) \mid t \in [t_1, t_2] \wedge \mathbf{p}_t[i] \neq 0\}$.

Let $\mathcal{Q}_T$ and $\mathcal{Q}_{T_0}$ be the best LGM plans optimized for refresh times T and T_0, respectively. We construct an LGM plan $\mathcal{N}$ for the period $[0, T_0]$ by combining $\mathcal{Q}_T$ and $\mathcal{Q}_{T_0}$ as follows:

- $\mathcal{N}[0, T-1] = \mathcal{Q}_T[0, T-1]$.
- At time T, if the pre-action state under $\mathcal{N}$ is full, let $\mathcal{N}$ take any greedy and minimal action that results in a non-full post-action state.
- $\mathcal{N}[T+1, T_0] = \textsc{MakeLGMPlan}(\mathcal{Q}_{T_0}[T+1, T_0])$.

Now, let us compare $\mathcal{N}[T, T_0]$ and $\mathcal{Q}_{T_0}[T+1, T_0]$ in terms of the number of actions on R_i modifications. Suppose the first action of $\mathcal{Q}_{T_0}[T+1, T_0](i)$ occurs at T'. By Lemmas 5 and 6, $|\mathcal{N}[T', T_0](i)| \leq |\mathcal{Q}_{T_0}[T+1, T_0](i)|$. The remaining task is to bound $|\mathcal{N}[T, T'-1](i)|$. Let $\mathbf{s}_t$ and $\mathbf{s}_{t^+}$ denote the pre- and post-action state at time t under $\mathcal{N}$, respectively. Consider the first time $T'' \in [T, T'-1]$ when $\mathbf{s}_{T''^+}(i) = 0$. If there is no such T'', then $\mathcal{N}[T, T'-1](i)$ contains no action, because actions are greedy and always result in a post-action state of 0. If $\mathbf{s}_T(i) = 0$, then obviously $\mathcal{N}(i)$ takes no action at T, and $T'' = T$. Otherwise, T'' is the time of the first action in $\mathcal{N}[T, T'-1](i)$. Either way, at T'', $\mathbf{s}_{T''^+}[i] = 0$ must be less than or equal to that of the corresponding post-action state under $\mathcal{Q}_{T_0}$. At any time $t \in [T''+1, T'-1]$, $\mathbf{s}_t[i]$ must remain less than or equal to the size of post-action state of $\triangle R_i$ under $\mathcal{Q}_{T_0}$ at t, because $\mathcal{Q}_{T_0}$ takes no action during $[T''+1, T'-1]$. By construction with MAKELGMPLAN, $\mathcal{N}[T+1, T_0]$ would not take any action during $[T''+1, T'-1]$. In conclusion, we have:

$$|\mathcal{N}[T, T_0]| \leq |\mathcal{Q}_{T_0}[T+1, T_0]| + \delta, \tag{1}$$

where $\delta = 0$ if $\mathbf{s}_T[i] = 0$; otherwise $\delta = 1$.

Let K_i be the total number of R_i modification arrived during $[0, T_0]$. Recall from Section 3.3 that $f(\mathcal{Q}_{T_0}) = \sum_{i=1}^{n} a_i K_i + \sum_{i=1}^{n} b_i |\mathcal{Q}_{T_0}(i)|$ and $f(\mathcal{N}) = \sum_{i=1}^{n} a_i K_i + \sum_{i=1}^{n} b_i |\mathcal{N}(i)|$. Since $\mathcal{Q}_{T_0}$ is an optimal LGM plan for the time interval $[0, T_0]$, we have:

$$\sum_{i=1}^{n} b_i |\mathcal{Q}_{T_0}(i)| \leq \sum_{i=1}^{n} b_i |\mathcal{N}(i)|.$$

Note that the left-hand side of this inequality is equal to

$$\sum_{i=1}^{n} b_i |\mathcal{Q}_{T_0}[0, T](i)| + \sum_{i=1}^{n} b_i |\mathcal{Q}_{T_0}[T+1, T_0](i)|,$$

while the right-hand side is equal to

$$\begin{aligned} & \sum_{i=1}^{n} b_i \Big(|\mathcal{N}[0, T-1](i)| + |\mathcal{N}[T, T_0](i)| \Big) \\ \leq \; & \sum_{i=1}^{n} b_i |\mathcal{Q}_T[0, T-1](i)| + \\ & \sum_{i=1}^{n} b_i |\mathcal{Q}_{T_0}[T+1, T_0](i)| + \sum_{\mathbf{s}_T[i] \neq 0} b_i, \end{aligned}$$

where the last inequality follows from the fact that $\mathcal{N}[0, T-1] = \mathcal{Q}_T[0, T-1]$ (by construction) and (1). Therefore,

$$\sum_{i=1}^{n} b_i |\mathcal{Q}_{T_0}[0, T](i)| \leq \sum_{i=1}^{n} b_i |\mathcal{Q}_T[0, T-1](i)| + \sum_{\mathbf{s}_T[i] \neq 0} b_i.$$

Note that the right-hand side is exactly $f(\mathcal{Q}_T) - \sum_{i=1}^{n} a_i K_i$. By Theorem 2, $f(\mathcal{Q}_T) = \mathrm{OPT}_T$. Hence,

$$\sum_{i=1}^{n} b_i |\mathcal{Q}_{T_0}[0, T](i)| + \sum_{i=1}^{n} a_i K_i \leq \mathrm{OPT}_T.$$

Finally, note that by construction, $\mathcal{Q}_{T_0,T}[0, T-1] = \mathcal{Q}_{T_0}[0, T-1]$, and $\mathcal{Q}_{T_0,T}$ may take one more action at T.

Therefore,

$$\begin{aligned} & f(\mathcal{Q}_{T_0,T}) = \sum_{i=1}^{n} a_i K_i + \sum_{i=1}^{n} b_i |\mathcal{Q}_{T_0,T}[0,T](i)| \\ \leq & \sum_{i=1}^{n} a_i K_i + \sum_{i=1}^{n} b_i \Big(|\mathcal{Q}_{T_0}[0,T-1](i)| + 1 \Big) \\ \leq & \ \mathrm{OPT}_T + \sum_{i=1}^{n} b_i. \end{aligned}$$

□

4.3. An Online Heuristic Algorithm

The approach presented in the previous section needs to precompute an optimal LGM plan for an estimated refresh time T_0, which requires precise knowledge of the modification arrival sequence. The cost of precomputing and remembering the plan can be expensive, especially if T_0 is chosen to be large (to provide a better bound when $T > T_0$). In this section, we provide an online heuristic algorithm to produce a LGM plan on the fly, with no precomputation and very little bookkeeping.

Without loss of generality, let 0 be the time when the last view refresh took place. Suppose at the current time t, the response-time constraint is violated, i.e., $f(\mathbf{s}_t) > C$ where $\mathbf{s}_t$ is the pre-action state at t. Clearly, we have to take a valid action at this time. We determine this action as follows. Let F_t be the total maintenance cost incurred by the plan so far during the time interval $[0, t)$. We choose a greedy, minimal, and valid action $\mathbf{q}_t$ that minimizes the following quantity:

$$H(\mathbf{q}_t) = \frac{F_t + f(\mathbf{q}_t)}{t + \textsc{TimeToFull}(\mathbf{s}_t - \mathbf{q}_t)},$$

where $\textsc{TimeToFull}(\mathbf{s})$ predicts the number of time step it would take for incoming modifications to make the pre-action state full again, given an initial state $\mathbf{s}$. Intuitively, $H(\mathbf{q}_t)$ measures the amortized maintenance cost of the plan if $\mathbf{q}_t$ is taken at time t. The algorithm basically attempts to minimize this measure in a greedy manner.

At runtime, the algorithm needs to record the running sum F_t and enough information to compute TIMETOFULL, which can be an n-vector $\mathbf{v}_t$ that records the recent modification arrival rate for each base table. It is straightforward to maintain F_t and $\mathbf{v}_t$ incrementally at runtime.

A straightforward implementation of the above algorithm may require enumerating up to all $2^n - 1$ possible choices of $\mathbf{q}_t$, each corresponding to a non-empty subset of $\{R_1, \ldots, R_n\}$. In practice we have found this implementation acceptable, because n is typically a very small constant, e.g., $n \leq 5$ for the TPC-R views we use for our experiments.

5. Experiment

Data, view, and modifications. We conduct proof-of-concept experiments with the TPC-R benchmark [22]. We choose the following representative view, which is an aggregate over a four-way join:

```
SELECT MIN(PS.supplycost)
FROM PartSupp AS PS, Supplier AS S,
     Nation AS N, Region AS R
WHERE S.suppkey = PS.suppkey
  AND S.nationkey = N.nationkey
  AND N.regionkey = R.regionkey
  AND R.name = 'MIDDLE EAST';
```

The two large base tables, `PartSupp` and `Supplier`, have $800,000$ and $10,000$ rows, respectively. Each modification randomly updates either a `PartSupp` row's `supplycost`, or a `Supplier` row's `nationkey`.

Cost functions. We write SQL statements to maintain the view incrementally given a batch of updates on either `PartSupp` or `Supplier`. Figure 4 shows the cost of processing each type of updates as a function of the size of the update batch. These measurements are obtained on a commercial DBMS running on a Linux server with 2GB of memory.

There are some irregularities in the curves due to the sheer complexity of the SQL maintenance statements; they need to deal with the state bug [5] and the case when `MIN` is not incrementally maintainable, for example. However, both curves are approximately subadditive and follow linear trends. Note that the cost of processing `PartSupp` updates remains fairly stable after an initial increase, probably because all small joining tables are loaded into memory for processing, and the chance that `MIN` is affected by a random `supplycost` update is small. The cost of processing `Supplier` updates is higher, probably because the joining table `PartSupp` is much larger.

Simulation and validation. In order to speed up experiments over long update arrival sequences, we *simulate* the execution of maintenance plans instead of actually running them. We then use the simulation traces and the cost functions measured in Figure 4 to calculate costs of plans. To validate our simulation, we also run the same plans on the real system and measure their actual costs. Figure 5 shows the results of one validation run for three different plans. We see that there is negligible difference between the simulated costs and the actual ones.

Comparison of plans. We compare the performance of four maintenance plans:

- NAIVE is the symmetric approach described in Section 1, which simply processes all modifications whenever the response-time constraint is violated.
- $\mathrm{OPT}^{\mathrm{LGM}}$ is the optimal LGM plan found by A^*SEARCH described in Section 4.1, assuming advance knowledge of the update sequence and refresh time.
- ADAPT is the plan obtained by adapting an optimal LGM plan for a different refresh time, as discussed in Section 4.2.
- ONLINE is the plan obtained by the online heuristic algorithm discussed in Section 4.3, which assumes

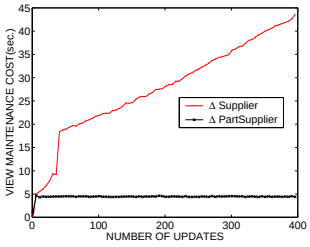

Figure 4. Cost function.

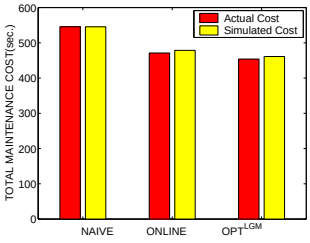

Figure 5. Simulated vs. actual cost.

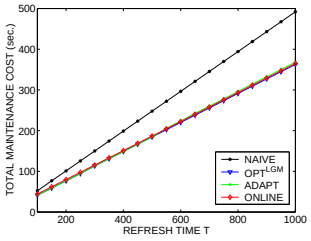

Figure 6. Varying refresh time.

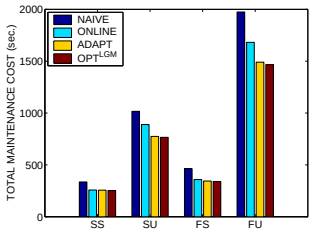

Figure 7. Non-uniform arrivals.

no advance knowledge of either the update sequence or the refresh time.

Figure 6 compares the performance of these plans when the time of refresh varies from 100 to 1000 seconds. One `PartSupp` update and one `Supplier` arrive at every time step. The response-time constraint is always set to 12 seconds. For $\mathrm{OPT}^{\mathrm{LGM}}$, an optimal LGM plan is computed for each refresh time. ADAPT, on other hand, is adapted from the same LGM plan optimized for a refresh time of 500 seconds. From the figure, we see that NAIVE is clearly outperformed by all other approaches. Moreover, ADAPT and ONLINE both perform very close to OPT, even though they use less advance knowledge.

Non-uniform modification arrivals. In the previous experiment, a constant number of modifications always arrive at regular intervals. To see how well ONLINE can handle instabilities in modification streams, we generate non-uniform modification arrival sequences as follows. For each time step t, let $p \in [0, 1]$ be the probability that at least one modification arrives at t, and let X_t be a random variable following the normal distribution with mean μ and variance σ^2. The probability that $d > 0$ modifications arrive at t is set to $p \times \mathbf{Pr}\{\lceil X_t \rceil = d \mid X_t > 0\}$. We control the rate and stability of the update stream by setting parameters p and σ. To model *slow* and *fast* streams, we set $p = 0.5$ and $p = 0.9$, respectively. To model *stable* and *unstable* streams, we set $\sigma = 1$ and $\sigma = 5$, respectively. We let μ remain at 1.

Figure 7 shows the performance of different plans on four types of update streams: slow/stable (SS), slow/unstable (SU), fast/stable (FS), and fast/unstable (FU). The response-time constraint is 20 seconds and refresh time is 1000 seconds. Again, $\mathrm{OPT}^{\mathrm{LGM}}$ is optimized for each given modification sequence and refresh time. From the figure, we see that NAIVE is again outperformed by other approaches for all four streams. ONLINE comes quite close to $\mathrm{OPT}^{\mathrm{LGM}}$ for stable update streams, but is not as good for unstable update streams, probably because of the inaccuracy in TIMETOFULL predictions.

6. Related Work

It is interesting to observe how new data management applications rejuvenate traditional research problems in databases. Materialized views, for example, have witnessed a resurgence in the nineties because of their applications in data warehousing and query optimization. A good survey of materialized views can be found in [9]. Incremental maintenance (e.g., [2, 3, 8, 11, 17]), self-maintenance (e.g., [19, 10]), and data warehousing (e.g., [23, 1, 7]) have been the traditional focuses of much research on materialized views.

Colby et al. [5] propose *deferred view maintenance* instead of refreshing views immediately upon updates. They focus on techniques that avoid the *state bug* (i.e., using the post-update state of base tables incorrectly in maintenance queries) and minimize the time when a view is locked for refresh. Mumick et al. [14] develop efficient batch processing techniques for aggregate views. In addition to batching data modifications, Liu et al. [12] also consider batching schema changes. In [6], Colby et al. discuss how to support a combination of deferred, immediate, and periodic maintenance policies for different views. These techniques are orthogonal to ours, because they do not consider, for each single view, the possibility of using different batching policies for modifications from different base tables in order to exploit natural asymmetries among components of the maintenance cost. Salem et al. [21] propose maintaining a view in small, asynchronous steps in order to reduce the contention between view maintenance and other database operations. Our approach is also asynchronous in the sense that delta tables are not always emptied together. Beyond this similarity, however, our optimization metric is very different from theirs; for example, benefit of batch processing is not really a factor in [21]. To the best of our knowledge, our work is the first to exploit the asymmetric benefits of batch processing using asymmetric maintenance plans.

Another class of problems broadly related to ours is efficiently maintaining some form of derived data under some resource constraint. Some recent examples include maximizing the accuracy of approximate caches given a bandwidth constraint [16], monitoring changes on the Web under resource constraints [18], just to name a few.

7. Conclusion and Future Work

In this paper we consider the problem of maintaining a materialized view batch-incrementally under a response-time constraint. We have identified optimization opportunities that arise from asymmetries among different components of the view maintenance cost. Traditional ap-

proaches to batch view maintenance treat modifications symmetrically, and therefore are unable to take advantage of such opportunities. We have proposed a novel asymmetric approach that leads to more efficient view maintenance plans. We have developed a series of algorithms to search for best asymmetric maintenance plans, with a range of trade-offs among optimization efficiency, amount of prior knowledge required, and optimality of result plans. We have also demonstrated the promise of our approach and effectiveness of our algorithms with preliminary experiments.

We have identified a number of interesting directions for future investigation. First of all, we are interested in developing a cost bound for the online heuristic algorithm in Section 4.3. Second, so far we have only made very general assumptions (monotonicity and subadditivity) about the cost functions; it will be interesting to see whether a stronger assumption, e.g, concavity, can lead to a tighter bound on the quality of LGM plans. We are also interested in cases where the cost may not be subadditive for some parts of the domain, which might occur because of limitations in practical query optimizers. Third, we are currently developing techniques to other types of asymmetries in maintenance cost. Specifically, in the query plan representing a maintenance query, different operators may be more or less amenable to batch processing. Propagating modifications through some operators while batching them in front of others may lead to further savings in total maintenance cost. Finally, we plan to extend our optimization framework to other types of resource constraints, since subadditivity seems to be a very common property in cost functions.

References

[1] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *Proc. ACM SIGMOD Intl. Conf. Management of Data*, pages 349 – 356, 1997.

[2] J. A. Blakeley, N. Coburn, and P. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. Database Systems*, 14(3):369–400, September 1989.

[3] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. 17th Intl. Conf. Very Large Data Bases*, pages 577 – 589, 1991.

[4] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagraCQ: A scalable continuous query system for internet databases. In *Proc. 2000 ACM SIGMOD Intl. Conf. Management of Data*, pages 379–390, Dallas, Texas, USA, May 2000.

[5] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proc. ACM SIGMOD Intl. Conf. Management of Data*, pages 469–480, 1996.

[6] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting multiple view maintenance policies. In *Proc. 1997 ACM SIGMOD Intl. Conf. Management of Data*, pages 405–416, 1997.

[7] H. Garcia-Molina, W. J. Labio, and J. Yang. Expiring data in a warehouse. In *Proc. 24th Intl. Conf. Very Large Data Bases*, pages 500–511, 1998.

[8] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. ACM SIGMOD Intl. Conf. Management of Data*, pages 328 – 339, 1995.

[9] A. Gupta and I. S. Mumick, editors. *Materialized Views: Techniques, Implementations and Applications*. MIT Press, June 1999.

[10] N. Huyn. Multiple-view self-maintenance in data warehousing environments. In *Proc. 23rd Intl. Conf. Very Large Data Bases*, pages 26–35, 1997.

[11] W. J. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom. Performance issues in incremental warehouse maintenance. In *Proc. 2000 Intl. Conf. Very Large Data Bases*, Cairo, Egypt, September 2000.

[12] B. Liu, S. Chen, and E. A. Rundensteiner. Batch data warehouse maintenance in dynamic environments. In *Proc. 11th Intl. Conf. Information and Knowledge Management*, pages 68 – 75, 2002.

[13] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. Knowledge and Data Engineering*, 11(4):610–628, 1999.

[14] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proc. ACM SIGMOD Intl. Conf. Management of Data*, pages 68 – 75, 1997.

[15] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proc. 2001 ACM SIGMOD Intl. Conf. Management of Data*, Santa Barbara, California, USA, May 2001.

[16] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *Proc. ACM SIGMOD Intl. Conf. Management of Data*, pages 73–84, 2002.

[17] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *Proc. 2002 Intl. Conf. Very Large Data Bases*, August 2002.

[18] S. Pandey, K. Dhamdhere, and C. Olston. Wic: A general-purpose algorithm for monitoring web information sources. In *Proc. 2004 Intl. Conf. Very Large Data Bases*, pages 360–371, Toronto, Canada, September 2004.

[19] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. 1996 Intl. Conf. Parallel and Distributed Information Systems*, pages 158–169, December 1996.

[20] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2002.

[21] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How to roll a join: Asynchronous incremental view maintenance. In *Proc. ACM SIGMOD Intl. Conf. Management of Data*, pages 129–140, 2000.

[22] Transaction Processing Council. TPC-R. `http://www.tpc.org/tpcr/`.

[23] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. ACM SIGMOD Intl. Conf. Management of Data*, pages 316–327, 1995.