Managing Change in Large-Scale Data Sharing Systems

Peter Mork   Steven D. Gribble   Alon Y. Halevy

Abstract

The problem of sharing data across multiple sources has received considerable attention in recent years because of its relevance to enterprise data management, scientific data management, and information integration on the WWW. However, the management of updates in such systems has received very little attention. In a data sharing system, the set of sources and clients is not fixed, and therefore the sources publishing the updates do not necessarily know exactly who will consume them. Consequently, the system needs to support a variety of update propagation strategies. In this paper, our approach is based on identifying two kinds of objects of interest, which are treated as first-class citizens in the system: *updategrams*, which are descriptions of updates over base relations, and *boosters*, which complement updategrams to speed up the processing of join views. We derive a complete set of rules governing the production, combination, and reconciliation of updategrams and boosters over differing time intervals. Our rules cover both GAV and LAV style mediation, as well as views involving certain forms of aggregation. We show how to use our rules to produce efficient query execution plans by extending the System-R style query optimizer, and present experiments that evaluate several heuristics for pruning the search space of plans that use updategrams and views for query evaluation.

1 Introduction

The problem of sharing data among multiple sources within or between enterprises has received significant attention in research and in the commercial world. Over the years, several data sharing architectures have been investigated, such as federated databases [35], data integration systems [19, 25, 38], data exchange [10, 31] and peer-data management systems [4, 17, 20, 22]. Abstractly, a data sharing system consists of a set of distributed nodes, each serving one or more of the following roles: a data producer, a client issuing queries, a mediator translating between schemas of other nodes, or a cache of materialized views over other nodes' data. Data producers may generate updates individually or in batches, and at varying rates, perhaps even emitting a continuous data stream. While data sharing has led to a rich body of research concerning architectural and semantic mapping issues, the *dynamic* aspects of data in such systems has received much less attention. As systems become more loosely coupled, and data is subject to frequent changes, the problem of managing updates in these systems becomes more pronounced.

In this paper, we present a framework for managing updates in such data sharing systems. We formalize updates to data sources as *updategrams*, a collection of insertions, removals, and modifications to the data source. Applying an updategram to a data source updates that source from an initial version to a more recent version. Our framework allows different data sharing systems and applications to have differing policies for *when* to propagate updategrams: eagerly at update time, lazily at query time, or using a hybrid policy.

In addition to the updates themselves, some nodes may publish auxiliary information to speed up the computation of join queries. Previous literature [32] has proposed *auxiliary views* that speed up the computation of certain views in a data warehouse. Here we introduce *update boosters*, that are a refinement of auxiliary views in that they are built w.r.t. a particular updategram. Specifically, suppose one node publishes an updategram over its relation R, and suppose many other nodes rely on the join $R \bowtie S$. We formalize the subset of S that is *relevant* to the update to R w.r.t. $R \bowtie S$ as an update booster. Since boosters are built w.r.t. an updategram, they are much smaller that an auxiliary view. Update boosters may be published in order to speed up the computation of $R \bowtie S$, or to reduce the load on the producers of S. As with updategrams, different nodes may have differing policies on which boosters should be published, and when.

1.1 Motivating Scenarios

Our data sharing framework can support highly diverse systems, each with different policies governing the production and distribution of updategrams and boosters. For example:

Data warehouses: In this simple scenario, there are a number of data providers who publish updategrams and boosters according to a fixed schedule (e.g., daily or weekly), and a data warehouse that maintains a collection of views over the data providers' base relations. Upon receipt of new updategrams and boosters, the data warehouse immediately updates its views. Updating a view is equivalent to generating an updategram for it, using the provided base relations' updategrams and boosters.

Peer data management systems: In this more complex scenario, data providers publish base relations and updategrams over their relations. Additionally, third parties manage caches of materialized views (e.g., answers to popular queries) at strategic locations across the system, though these materialized views may be out of date. When a new query is issued, the system selects an appropriate cache to answer the query, and refreshes the materialized views within the cache as necessary. Doing so requires the system to identify an optimal refresh strategy, which may involve the base relations' updategrams, or updategrams and boosters produced and shared by other caches in the system.

Mobile peers: In this scenario, data providers again publish updategrams when their base relations change. However, the clients in the system are mobile, occasionally establishing connections to the data providers, but also occasionally establishing connections to each other. Because of this, it is not possible to propagate changes eagerly to all participants. Instead, some form of lazy or incremental propagation (such as epidemic propagation [36]) will be necessary. As the peers come into contact with each another, they can exchange updategrams and boosters, helping each other gradually become up-to-date. Moreover, if certain join conditions are particularly common within the peers' query workload, it may be economical for some peers to voluntarily promote themselves to become dedicated booster providers, in a manner similar to how peers within file-sharing systems voluntarily promote themselves to become "supernodes." [21]

In all of these scenarios, treating updates as first-class objects facilitates update propagation, and sharing boosters helps to amortize the cost of view maintenance. This is especially true in scenarios in which the base relations are not always accessible.

1.2 Contributions

Given a system that explicitly publishes updategrams and boosters, an obvious initial question arises:

- **Efficient query evaluation:** Given a query at a node in the system, how can that node use updategrams available to it to answer the query most efficiently?

This initial question has been studied in depth by the view-maintenance literature (see [15] for a survey). However, a rich set of previously unanswered questions presents itself when one considers updategrams and boosters within the spectrum of policies that are possible to control their generation and publication in a distributed system. For example:

- **Booster relevance and use:** Boosters are often relevant not only w.r.t. the query for which they were produced, but other current and future queries as well. When can boosters be used to answer queries, and what is the most efficient way to use them?
- **Updategram and booster creation:** a node in the network may want to create a set of updategrams and boosters to best support a particular set of queries. For example, consider a node that serves as a gateway to a company providing boosters and updategrams that are relevant to a small set of simple queries. How can these be used to create new updategrams and boosters that are relevant for different, more complex queries?
- **Evaluating queries over previous time points:** some applications may want to support queries about previous versions of data. When and how can updategrams and boosters of one version interval be used when answering a query on a different, but related, interval?
- **Other mediation formalisms:** how can all of the above questions be answered under various mediation languages (e.g., GAV, LAV, GLAV), or in the presence of aggregates?

The main contribution of this paper is to develop a set of rules governing the use of updategrams and boosters. The rules demonstrate 1) how updategrams and boosters of complex views can be computed from those of simpler views, 2) how to manage and reconcile updategrams and boosters describing changes to data over different (possibly overlapping) time intervals, and 3) they point out important properties for query optimization in the presence of updategrams and boosters. Our rules consider several mediation languages, including GAV, LAV, and their combination, GLAV; in fact, this is the first paper to discuss updates in a LAV-based system. We describe extensions to the rules to support certain kinds of grouping and aggregation.

Next, we describe how to use our rules to produce efficient query execution plans. In particular, we describe a modification to a System-R style query optimizer that makes use of available updategrams and boosters in query join enumeration. We describe two heuristics for pruning the space of plans searched, and describe an experiment validating the effectiveness of the heuristics. We show that our rules are complete, in the sense that they enable us to explore the complete space of possible query execution plans that exploit a given set of updategrams and boosters. Together, our rules and query optimization algorithms provide a base set of mechanisms that enable the management of updates in a wide variety of data sharing systems and policies.

The rest of this paper is organized as follows. Section 2 formally defines our problem setting. Section 3 describes our rule set for GAV, and Section 4 describes the rules for LAV, GLAV and aggregate views. Section 5 describes how a query optimizer can use updategrams and boosters. Section 6 discusses related work, and Section 7 concludes.

2 Problem definition

Base relations:

$N1$: Genes(GeneID int, GeneName string, ChromosomeLocus string)
$N2$: Variants(VariantID int, GeneID int, Sequence string, Frequency double)
$N3$: Proteins(ProteinID int, VariantID int, ProteinName string)

View relations:

$V1$:
```
SELECT * FROM Genes, Variants
WHERE Genes.GeneID = Variants.GeneID
AND Frequency > .01;
```
$V2$:
```
SELECT * FROM Genes, Variants, Proteins
WHERE Genes.GeneID = Variants.GeneID
AND Variants.VariantID = Proteins.VariantID
AND Frequency > .01;
```
$V3$:
```
SELECT * FROM Variants, Proteins
WHERE Variants.VariantID = Proteins.VariantID;
```

Figure 1: An example data sharing system, in which three nodes publish base relations, and the three publish view relations.

A data sharing system consists of a set of nodes. Nodes can perform one or more functions: (1) they provide some base data, (2) they store data that is computed over other nodes (i.e., views), or (3) they do not contain any data, but serve only as logical mediators between other sets of nodes. For example, data integration systems are an instance of such a system where there is one node of type (3), called the mediator, and the rest are of type (1). In a data warehouse context, the mediator node is of type (2) and stores views over the data sources. Recently, there has been increased interest in *peer-data management systems* [17, 4, 20, 22, 29], where there is no single logical mediated schema, and peers in the system are related by local semantic mappings. In this context any node can fulfill some or all of the various roles.

We assume that participants use the relational data model, i.e., each peer has its own relational schema. We assume that queries and view definitions are select-project-join (SPJ) queries. Furthermore, for convenience we assume that the join and selection conditions in the WHERE clause are already closed under logical deduction (e.g., if $A = B$ and $B = C$ are in the clause, then so is $A = C$). (Note that closure can always be computed efficiently).

A major difference between data sharing systems is the particular formalism that is used to describe the semantic relationships between different nodes. In the simplest case, often referred to as global-as-view (GAV), there is a set of nodes with base data, and other nodes' contents are defined as views over the base data (or over other views). Hence, query answering at any node can be performed by first unfolding all the view definitions, and then evaluating the query over the base data. In the second formalism, called local-as-view (LAV), the contents of the base data are defined as a view over the relations of other nodes. Lenzerini [25] surveys the theoretical aspects of both formalisms, and [17] shows how (and when) queries can be answered when the two formalisms can be combined in a network of nodes. Together, these formalisms provide a rich language for specifying mappings between highly heterogeneous data.

For brevity, we consider GAV mappings first and defer LAV mappings to Section 4.1. We note that to the best of our knowledge, this is the first work that deals with updates in a LAV context.

Here we are *not* concerned with the rich problem of which views to maintain at a particular node in the network, or with the problem of *how* the mappings between the nodes are obtained (see [33] for a recent survey on that topic). For convenience, we assume there is a shared meta-data repository containing all schema information for all nodes; this allows the optimizer to know about all of the objects (relations/updategrams/boosters) in the system. Finally, note that our results are applicable to both sets and multi-sets because the information contained in updategrams and boosters is sufficient for the counting algorithm presented in [16] for updating views.

Example 2.1 One of the domains driving our work is the ad-hoc sharing of data among scientists. The following example, used throughout the paper, is extracted from that domain. Consider a system with 6 nodes, three that each publish one base relation and three that publish views defined over the base relations (see Figure 1). One node publishes a relation Genes that tracks named genes and their locations on the human genome. Given that a gene can appear in many forms (e.g., blue eyes vs. brown eyes), the second node publishes a relation Variants that tracks known variants, including the frequency and DNA sequence of each variant. Finally, a given gene variant can produce multiple proteins, so the third node publishes a relation Proteins that maps variants to named proteins.

The view relations defined by the other three nodes are as follows. One node publishes a view V1 that is of interest to a genetic counselor; V1 shows common variants that the counselor might expect to see frequently. The second publishes V2 that is of interest to a biochemist, who wants to determine which genes and proteins are related. Finally, the third publishes V3 that is of interest to a genomics researcher, who wants to know for which proteins a DNA sequence has already been identified. □

2.1 Updategrams

As explained in the introduction, our goal is to support a variety of scenarios in which updates need to be propagated from the data producers to clients. We achieve this goal by treating updates as first class citizens, and separating between the policies governing their production, and those governing their use.

In our discussion, updates originate from the owners of the base data and are initially expressed in terms of changes to base relations. Nodes that export base relations publish updates as they see fit, with a frequency and granularity chosen to suit the application. For example, a node can choose to publish an update after every committed transaction, or according to set schedule in which updates are published daily, regardless of the number of transactions that have occurred. Updates are published in the form of *updategrams*, which specify the tuples in the relation that have been inserted, deleted or modified. We note that in some cases it is possible to determine that a view is independent of a particular update [26], and therefore the update process can be skipped entirely. In our discussion, we assume that our updategrams have already been checked for these conditions.

Version numbers and vectors: To define updategrams formally, we need to introduce notation for tracking versions of relations and views. Different versions of relations are denoted by *version numbers*. We use R^t to denote the t-th version of the relation R. The version number is increased every time an updategram is published.

We use *version vectors* to specify the versions of a view. The version vector of a view V contains a version number for each base relation on which V depends. Specifically, since we are considering GAV mappings, we let V' be the unfolding of the definition of V that refers only to the base relations, and let $R_1, \ldots, R_m$ be the relations appearing in V'. The version vector of V contains m elements, where the i-th element refers to a version of R_i. For example, the version vector $V^{t_1,t_2,\ldots,t_m}$ indicates that the current contents of V were computed using the data in $R_1^{t_1}, R_2^{t_2}, \ldots, R_m^{t_m}$.

Note that if a relation R appears multiple times in the expansion V', then it still has only one entry in the version vector; we do not want to consider anomalous view versions that depend on two different versions of the same base relation. For brevity, we will frequently abbreviate the version vector $t_1, t_2, \ldots, t_m$ as $\vec{t}$. We will sometimes slightly abuse notation and not distinguish between a version vector of length one and a version number. We can now define updategrams formally:

Definition 2.1 (**Updategrams**) An *updategram* contains the list of changes (insertions, deletions, and updates) necessary to advance a relation from one version number to a later version number:$\mu_R^{i,j}$ contains the changes that must be applied to advance R^i to R^j. Similarly, $\mu_V^{\vec{i},\vec{j}}$ advances $V^{\vec{i}}$ to $V^{\vec{j}}$. □

Whenever the old state and new state are obvious from the context, we will omit the version numbers/vectors. Additionally, for the purposes of our discussion we do not specify the format of an updategram. It is sufficient to assume that the size of the updategram is (at most) proportional to the number of tuples affected by the update.

Example 2.2 Assume that our example system has been running for some time; the base relations' version numbers are $\textsf{Genes}^3$, $\textsf{Variants}^7$ and $\textsf{Proteins}^5$. $V1$ and $V2$ are current with $V1^{(3,7)}$ and $V2^{(3,7,5)}$, but $V3$ has not been re-materialized recently, and has version vector $V3^{(2,1)}$.

As an example of an updategram, suppose that it has been decided that the gene formerly known as 'PRNC' should be renamed, 'AFKAP'. An updategram($\mu^{3,4}_{\textsf{Genes}}$) is published to reflect this; its contents can be encoded as:
(Delete, <42,'PRNC','17q2'>)
(Insert, <42,'AFKAP','17q2'>) □

2.2 Update Boosters

Updategrams can significantly speed up view maintenance because they save the query processor from having to access the entire base relation. However, as the literature on view *self-maintenance* shows (e.g., [14]), updategrams in isolation are only useful when queries do not involve joins. If a view V involves a join of relations $R_1, \ldots, R_n$, and R_1 has published an updategram, then we still need to access the base relations $R_2, \ldots, R_n$ to update V.

To speed up the recomputation of join views, we also publish *boosters* as first class citizens. Consider a query V that joins $R_1, \ldots, R_n$, and assume R_1 publishes an updategram. The booster of R_2 w.r.t. the updategram and V includes the set of tuples in R_2 that (1) may join with tuples mentioned in the updategram, and (2) satisfy the selection predicates in V. Update boosters are a refinement of the concept of auxiliary views (or view indices [32]). Whereas auxiliary views are the portion of R_2 that is relevant to the computation of a particular query, the booster is the part that is relevant w.r.t. a particular updategram and query. As a result, a booster is likely to be much smaller than the corresponding auxiliary view.

Example 2.3 Given that $\mu^{3,4}_{\textsf{Genes}}$ has been published, it may be desirable to update $V1$. This re-materialization can be speeded up if a booster w.r.t. $V1$ and the new updategram is available. This booster includes all the tuples of the relation Variant, where the GeneID is 42 (corresponding to the tuples in $\mu^{3,4}_{\textsf{Genes}}$), and whose Frequency is more than 0.01 (the condition from $V1$). □

To formally define boosters, we need to define when a tuple is *relevant* to a query w.r.t. an updategram. The notion of relevance depends on *derivations* of answers to the query.

Definition 2.2 (derivation) Let V be an SPJ view whose FROM clause is $R_1, \ldots, R_n$, and let $\bar{a}$ be an answer of V over a database D. A derivation, d, of $\bar{a}$ is a mapping that for every i, $1 \leq i \leq n$, $d(i)$ is a tuple in the relation R_i in D, $d(1), \ldots, d(n)$ satisfy the join and selection predicates in V, and $\bar{a}$ is the result of applying the SELECT clause of V to $d(1), \ldots, d(n)$. □

Note that an answer to V may have multiple derivations from a database.

Definition 2.3 (**relevance**) Let V be an SPJ view whose FROM clause is $R_1, \dots, R_n$. Let $\bar{a}_1$ be a tuple from R_1. We say that the tuple $\bar{a}_2$ of R_2 is relevant to V w.r.t. $\bar{a}_1$ if it is possible to construct a database D such that there is some derivation d of an answer to V where $d(1) = \bar{a}_1$ and $d(2) = \bar{a}_2$. □

Note that relevance is not defined w.r.t. a particular database instance. In a similar fashion, we can define a tuple $\bar{a}_2$ to be relevant to a set $\bar{A}_1$ of tuples from R_1 if it is relevant to *any* of the tuples in $\bar{A}_1$.
We can now define boosters formally:

Definition 2.4 (**booster**) Let V be an SPJ view definition whose FROM clause is $R_1, \dots, R_n$, let D be a database, and let μ_{R_1} be an updategram for R_1. The booster of R_2 w.r.t. μ_{R_1} and V is the subset of tuples of R_2 in D that are relevant to some tuple mentioned in μ_{R_1}. We denote the booster by $\beta_V(\mu_{R_1}, R_2)$; when V and μ_{R_1} are obvious, we abbreviate $\beta(R_2)$. Any subset of R_2 in D that is a superset of $\beta_V(\mu_{R_1}, R_2)$ is called a *super-booster*. □

Example 2.4 Now consider updating $V2$. It would be beneficial to have a booster w.r.t. $V2$ and $\mu^{3,4}_{\mathsf{Genes}}$defined over both Variants and Proteins (i.e., $S_1 = \{\mathsf{Genes}\}$ and $S_2 = \{\mathsf{Variants}, \mathsf{Proteins}\}$). The tuples in this booster include all the tuples in the join of Variants and Proteins for which the GeneID is 42 and the Frequency is more than 0.01. □

A booster of the form $\beta_V(\mu_{R_1}, R_2)$ is clearly useful for computing V. Moreover, we emphasize two properties of boosters that play an important role when we consider booster production policies. First, as we will show, boosters can be computed as a side-effect of using an updategram to update a view. Second, boosters are actually useful for a much larger set of views than the one for which they are computed; the booster of R_2 w.r.t. the updategram of R_1 and V will be useful for any view that joins R_1 and R_2 with a subset of the join and selection predicates of V. We note that Definition 2.4 can easily be extended to the case where R_1 and R_2 are replaced by two disjoint subsets S_1 and S_2.

3 Updategram and booster rules

We now describe a set of rules that governs the way updategrams and boosters can be used. The rules answer the basic questions raised at the outset, such as: How are updategrams and boosters produced and combined? How can they be used in query execution plans? When can we prune unneeded updategrams and boosters? How do we deal with varying version intervals? These rules can be used in a variety of settings, thereby being applicable to a range of strategies for producing, storing, integrating and reconciling updates.

This section shows that the set of rules is *complete* in the context of GAV-style mediation; it enables exploring all possible ways of using a given set of updategrams and boosters. The next section considers LAV and mediation involving aggregation.

Our rules describe the operations that combine updategrams with relations, updategrams with boosters, and boosters with boosters. Since boosters are relations, the latter operation is simply a join. When we combine an updategram μ_R with a relation S or with a booster, we also join the tuples in the updategram with those of the relation, but depending on the desired result, we need to do some additional bookkeeping. If the result is a booster, then the output includes the set of tuples in S that join with *some* tuple in μ_R. If the result is an updategram, then we label every tuple in the result with the label of the tuple in μ_R that it came from. For simplicity of exposition, we denote all these operations with the $\bowtie$ symbol despite the slight differences. We use the notation $R \overset{V}{\bowtie} S$ to specify that we join R and S using the join and selection conditions in the view V. In most cases, the left-hand side of each rule describes the operations necessary to produce the right-hand side.

Updategram and booster creation: the first set of rules tell us how to create boosters and how to create updategrams for views. Recall that $\beta_V(\mu_R, S)$ is a booster for S w.r.t. the updategram μ_R and the view V.

A1. $\mu_R \overset{V}{\bowtie} S = \mu_{R \overset{V}{\bowtie} S}$

A2. $\mu_R \overset{V}{\bowtie} \beta_V(\mu_R, S) = \mu_{R \overset{V}{\bowtie} S}$

A3. $\Pi_{attributes(S)}(\mu_R \overset{V}{\bowtie} S) = \beta_V(\mu_R, S)$

Rule A1 is the basic rule specifying how updategrams are used. If the goal is to update $R \overset{V}{\bowtie} S$ (i.e., to produce $\mu_{R \overset{V}{\bowtie} S}$), then one can join μ_R with S. A2 shows that the same result is achieved by using the booster $\beta_V(\mu_R, S)$ instead of the entire relation S. Rule A3 shows how boosters are created by combining updategrams with a relation, followed by the appropriate projection. (Under multi-set semantics, it is necessary to count the number of times a tuple appears in S; the booster must contain the same count.) Hence, A3 implies that a booster can be created as a side-effect of using the updategram, which is important when we consider the tradeoffs involved in creating and maintaining boosters.

A4. $\beta_V(\mu_R, S) \overset{V}{\bowtie} \beta_V(\mu_R, T) = \beta_V(\mu_R, S \overset{V}{\bowtie} T)$

A5. $(\mu_R \overset{V}{\bowtie} \beta_V(\mu_R, S)) \overset{V}{\bowtie} \beta_V(\mu_R, T) =$
$\mu_R \overset{V}{\bowtie} (\beta_V(\mu_R, S) \overset{V}{\bowtie} \beta_V(\mu_R, T))$

Rule A4 shows that two boosters w.r.t. the same updategram can be combined in order to create a booster for a view. Rule A5 shows that combining updategrams and boosters is an associative operation. This rule is key to the join enumeration algorithm we present later – the rule entails that during the join enumeration algorithm, we do not have to keep plans for a sub-expression V_1 of V if we have a booster for V_1 that is cheaper to access or compute.

Example 3.1 Consider the scenario in which $\mu^{3,4}_{\mathsf{Genes}}$ and $\beta = \beta_{V2}(\mu^{3,4}_{\mathsf{Genes}}, \mathsf{Variants})$ have been published. One plan to recompute $V2$ is to join $\mu^{3,4}_{\mathsf{Genes}}$ with Variants, and then to join that result with Proteins. A more attractive alternative might be to join $\mu^{3,4}_{\mathsf{Genes}}$ with β, and then join the result with Proteins. □

Using super-boosters: one of the key benefits of boosters is that they can be used across queries. That is, a booster created as a result of updating V_1 may be useful for a subsequent view V_2. However, the booster created for V_1 may not be exactly a booster w.r.t. V_2, but rather be a superset of the booster of V_2, i.e., a super-booster. The following rules show how super-boosters are used. In the rules, we use $(R \overset{V}{\bowtie} S) \subseteq (R \overset{W}{\bowtie} S)$ to denote that the query $(R \overset{V}{\bowtie} S)$ is *contained* in the query $(R \overset{W}{\bowtie} S)$ (i.e., produces a subset of the tuples for any given database instance).

A6. Let W be a view such that $(R \overset{V}{\bowtie} S) \subseteq (R \overset{W}{\bowtie} S)$.
Then, $\mu_R \overset{V}{\bowtie} \beta_W(\mu_R, S) = \mu_{R \overset{V}{\bowtie} S}$

A7. If $(S \overset{V}{\bowtie} T) \subseteq (S \overset{U}{\bowtie} T)$ and $(S \overset{V}{\bowtie} T) \subseteq (S \overset{W}{\bowtie} T)$,
then $\beta_U(\mu_R, S) \overset{V}{\bowtie} \beta_W(\mu_R, T) \supseteq \beta_V(\mu_R, S \overset{V}{\bowtie} T)$.

Rule A6 tells us that a booster created for a more general view can still be used to create an updategram (i.e., generalizes rule A2). Rule A7 says that if we join two boosters, each constructed for a query containing V (U and W), then the result is a super-booster for V.

Boosters for overlapping intervals: at a given point in time, there may be several updategrams available for a particular relation R, each associated with different start and end versions of R. Similarly, boosters will be created w.r.t. these updategrams with corresponding version intervals. The following rules show how to use and combine multiple updategrams and boosters over related intervals:

A8. If $i \leq j < k \leq l$, then $\mu_R^{j,k} \overset{V}{\bowtie} \beta_V(\mu_R^{i,l}, S) = \mu^{j,k}_{R \overset{V}{\bowtie} S}$

A9. If $(m = max(i,k)) < (n = min(j,l))$,
then $\beta_V(\mu_R^{i,j}, S) \overset{V}{\bowtie} \beta_V(\mu_R^{k,l}, T) =$
$\beta_W(\mu_R^{m,n}, S \overset{V}{\bowtie} T)$ where $V \subseteq W$

Rule A8 says that we can use a booster for a containing interval. Rule A9 says that if we combine two boosters of overlapping intervals, then we get a super-booster of their intersection interval.

Coalescing updategrams and boosters: Finally, one can bundle multiple updategrams and boosters into objects that span larger time intervals. In the following rules, we use the union symbol for coalescing, though the actual operation requires some simple additional bookkeeping:

A10. $\mu_R^{i,j} \cup \mu_R^{j,k} = \mu_R^{i,k}$

A11. If $V \subseteq W, V \subseteq X$ and $i < k \leq j < l$,
then $\beta_W(\mu_R^{i,j}, S) \cup \beta_X(\mu_R^{k,l}, S) =$
$\beta_Y(\mu_R^{i,l}, S)$ where $V \subseteq Y$

Rule A10 allows us to merge the contents of adjacent updategrams. Rule A11 says that boosters can also be merged, although the bookkeeping required depends on whether one is considering set or multi-set semantics (for example, UNION ALL cannot be used because this may inflate the multiplicity of certain tuples).

The following theorem shows that our rules are complete in that they enable exploring the entire space of possible plans that exploit a set of updategrams and boosters.

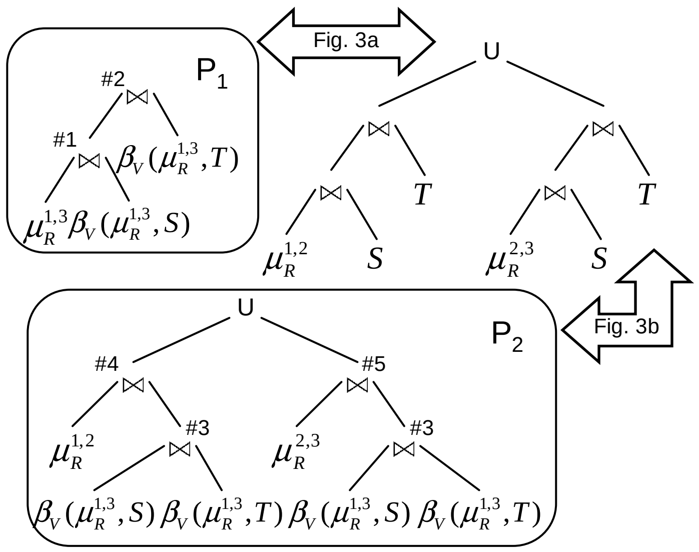

Figure 2: The figure shows three equivalent plans for updating $V = R \bowtie S \bowtie T$. The plan at the top-right is the *canonical* plan.

Equivalence	(Rule)	Description
#1= $\mu^{1,3}_{R \bowtie S}$ #2= $\mu^{1,3}_V$ = $\mu^{1,2}_V \bigcup \mu^{2,3}_V$	(A2) (A2) (A10)	(a) P_1 can be transformed into the canonical plan by splitting it into two updates.
#3= $\beta(S \bowtie T)$ #4= $\mu^{1,2}_V$ #5= $\mu^{2,3}_V$	(A4) (A8) (A8)	(b) P_2 can be transformed into the canonical plan by reordering operations.

Figure 3: The figure demonstrates how to transform plans P_1 to the canonical plan (upper half) and P_2 to the canonical plan (lower half).

Theorem 3.1 *Let V be a view over relations $R_1, \ldots, R_n$, and let $\mu^{i,j}_{R_1}$ be an updategram for R_1. Let P_1 and P_2 be two (possibly bushy) query trees for computing $\mu^{i,j}_V$, using the base relations or boosters. Then, using the equivalences in rules A1–A11 and the commutativity and associativity of joins, we can transform P_1 to P_2 and vice versa.*

Proof Sketch: The proof proceeds by showing that both P_1 and P_2 can be transformed to and from a canonical series of left-linear join trees that reference only the base relations and original update (see Figure 2). Each subsequent tree increments the version number by one. To show this, first expand each updategram that spans multiple versions into a collection of updategrams across incremental versions (using rules A10 and A11); this produces a series of incremental plans. Next, expand each booster in a plan P so it references a single base relation (using rules A4, A7 and A9), and then transform P into a left-linear join tree (using rule A5). Finally, replace each booster with the corresponding base relation (with rules A1, A2, A6 and A8). $\square$

Example 3.2 Figure 3 demonstrates the equivalence of two sample plans P_1 and P_2 (shown in Figure 2). Using rules A2 and A10, we can transform P_1 to/from the canonical plan by splitting the plan into two updates. Using rules A4 and A8, we can similarly transform P_2 to/from the canonical plan by reordering operations. □

4 Extended mediation languages

Building on the rules presented in the previous section, we now describe how to handle updategrams and boosters in LAV-style mediation (Section 4.1), and mediation involving grouping and aggregation (Section 4.2).

4.1 Schema mediation with LAV

The preceding section assumed that the relationships between data sources were described using the GAV formalism, i.e., there is a set of nodes providing base data, and other nodes are defined as views over the base nodes (perhaps with a hierarchy of views). In the local-as-view formalism (LAV), the relationships are described in reverse—the contents of base data nodes are described as views over the schemas of other nodes. LAV's key advantage is that it is easier to map a large set of data sources into a mediated schema, because they can be mapped *independently* of each other.

Example 4.1 Consider a *mediator* node that does not store any data, but just contains mappings to a set of other nodes that provide data. Suppose the mediator node has two relations: GeneNames(GeneID int, GeneName string) and GeneLoci(GeneID int, Locus string). Given this virtual schema, we may describe the data in Genes as:

```
SELECT N.GeneID, GeneName, Locus AS ChromosomeLocus
FROM   GeneNames AS N,  GeneLoci AS L,
WHERE  N.GeneID = L.GeneID
```

In this example, when Genes is updated, both GeneNames and GeneLoci may need updating. For example, if μ_{Genes} is the insertion of a single tuple (with non-null GeneID), the appropriate action (under multi-set semantics) is to add one new tuple to each of the virtual relations. □

In general, LAV and GAV mappings can be combined into a network of mappings (see [17] for a precise analysis of the restrictions we need to impose on such combinations in order to still be able to answer queries efficiently). We now extend our rule set to handle LAV-style mappings. Because we reduce LAV to GAV, we can also accommodate GLAV [11, 10], a formalism that combines the benefits of LAV and GAV.

Given the rules we have already described, the key to handling LAV is to reformulate the mappings using *inverse rules* [9]. Given a view V, we can use inverse rules to rewrite V as a non-recursive datalog program V' over the relations of the nodes with the base data[1]. The program V' provides a basis for extending the notation of version vectors – any relation that

[1]In general, if we consider functional dependencies, then the datalog program resulting from the inverse-rule technique may be recursive [8]. However, we do not consider functional dependencies here.

appears as a base relation in V' is part of the version vector of V. As the following example shows, it is important we express V in terms of the base relations, and not just the relations over which it is expressed:

Example 4.2 Continuing the previous example, assume the view is:

```
CREATE VIEW V AS
SELECT N.GeneID, GeneName, Locus AS ChromosomeLocus
FROM   GeneNames AS N,  GeneLoci AS L,
WHERE  N.GeneID = L.GeneID
```

When expressed in terms of the actual base data, V is just `SELECT * FROM Genes`. As such, μ_{Genes} can be used to update V directly. However, if we consider V to be dependent on $\mathsf{GeneNames}$ and $\mathsf{GeneLoci}$, then it will appear as if two relations have been changed. One may be tempted to apply each updategram in succession by joining $\mu_{\mathsf{GeneNames}}$ with $\mathsf{GeneLoci}^{old}$ and then joining μ_{GeneLoci} with $\mathsf{GeneNames}^{new}$. This will produce the correct result, but it is much more expensive. □

Since V' is non-recursive, we only need to extend our rules to deal with views that involve unions. Hence, we provide the following rule, that combines the updategrams of a set of views to an updategram for their union:

A12. If $V = V_1 \cup V_2 \cup \ldots \cup V_n$, then $\bigcup_{k=1}^{n} \mu_{V_k}^{\vec{\imath},\vec{\jmath}} = \mu_{V}^{\vec{\imath},\vec{\jmath}}$

This rule indicates that an update to a union of views is the union of the updates to the respective views. Note that elements of the $\mu_{V_k}^{\vec{\imath},\vec{\jmath}}$'s may interact: if $\mu_{V_k}^{\vec{\imath},\vec{\jmath}}$ inserts a tuple that is deleted from one or more other updategrams, then the net is that the tuple is inserted by the updategram. When multi-set semantics are considered, we also need to keep track of the multiplicities to get the final result.

4.2 GAV with Aggregation

Queries with grouping and aggregation are very useful in large-scale data sharing systems. We now describe how to extend our rules for GAV-style mediation where some views involve grouping and aggregation. For brevity, we assume the grouping and aggregation are over a single relation, but in practice that relation may result from a join query as well.

The key to extending our rules is the intuition that a booster is the subset of an auxiliary view that is relevant w.r.t. an update. Recall that auxiliary views were introduced in the view-maintenance literature in order to allow views to be self-maintainable [32]. Auxiliary views can be thought of as boosters relevant to *all* updates. Auxiliary views have been extended to include views with cumulative aggregates (SUM, COUNT, AVG) [28] and also top-k queries (including MAX, MIN) [39]. To illustrate, consider the view V, defined as follows:

```
SELECT A, AVG(C) FROM R GROUP BY A, B;
```

The auxiliary view V_A would be the following, designed to keep track of the sum and count on the `C` column:

```
SELECT A, B, SUM(C) AS D, COUNT(C) AS E
FROM R GROUP BY A, B;
```

Given a grouping query V defined over R, auxiliary view V_A and an update $\mu_R^{i,j}$, we define a *grouping booster,* $\gamma_V(\mu_R^{i,j})$, to be the subset of V_A that is relevant to the update. In our example, when (a, b, c) is inserted into R, the appropriate grouping-booster is:

SELECT A, D, E FROM V_A WHERE A=a AND B=b;

With this information, we can compute an updategram for V (as well as an updategram for V_A): delete $(a, d/e)$ from V and insert $(a, (d + c)/(e + 1))$.

A more interesting use of grouping boosters arises when we consider top-k views. In order to support deletions, the auxiliary view for a top-k view is a top-k' view, where $k' > k$. When the maximum value is deleted, the k-th value can be extracted from the auxiliary view. The auxiliary view now contains $k' - 1$ values. We can choose not to refresh the auxiliary view, or we can retrieve the k'-th value from some other location (or the base relation). This affords great flexibility: nodes can maintain various windows, which are refilled according to differing policies. Any peer with a larger window can be used to refill a peer with a smaller window.

The following rules formalize the properties of grouping boosters. In the rules, $\overline{V_g}$ denotes the grouping columns of V and $\overline{V_a}$ denotes its aggregate columns. Rule A13 indicates how grouping-boosters can be used in place of auxiliary views (note that in this case we overload the join operator to mean 'can be computed' as in the preceding examples). Rules A14–A16 generalize rules presented in section 3. Finally, rule A17 captures the intuition of the preceding example.

A13. $\mu_R \overset{V}{\bowtie} R = \mu_R \overset{V}{\bowtie} V_A = \mu_R \overset{V}{\bowtie} \gamma_V(\mu_R) = \mu_V$

A14. Let W be an aggregation query such that $\overline{W_g} \subseteq \overline{V_g}$ and $\overline{W_a} = \overline{V_a}$. Then, $\mu_R \overset{W}{\bowtie} \gamma_V(\mu_R) = \mu_W$.

A15. If $i \leq j < k \leq l$, then $\mu_R^{j,k} \overset{V}{\bowtie} \gamma_G(\mu_V^{i,l}) = \mu_V^{j,k}$.

A16. If $i < k \leq j < l$, then $\gamma_V(\mu_R^{i,j}) \cup \gamma_V(\mu_R^{k,l}) = \gamma_V(\mu_R^{i,l})$

A17. If V and W are top-k queries and $V_A \subset W_A$, then $\mu_R \overset{V}{\bowtie} \gamma_W(\mu_R) = \mu_{V_A}$.

In summary, the rules we describe allow a wide range of propagation policies in data sharing systems with mediation rules in GAV, LAV and aggregation views.

5 Computing updates

In this section we show how our rules can actually be used to optimize the propagation of updates. Any system that uses updategrams and boosters will face a variant of the following problem: given a set of views (with specified version vectors), base relations, updategrams, and boosters, compute an efficient plan for using the base relations, updategrams, and boosters to update one or more of the views. The point at which the optimization problem is addressed and the particular variant depends on the strategy the system uses to propagate updates.

In this section, we describe a System-R style join enumeration algorithm to make use of updategrams and boosters. We first consider the case in which we need to advance a view from one version to the next one, and then describe how to find a plan that advances a view over multiple versions. We discuss here only the case of updategrams to a single relation; the extension to multiple relations is conceptually similar.

5.1 Advancing a single step

We begin with the case where we want to compute the updated version of a view, V, after one of the relations on which it depends, R, has published an updategram μ_R. In addition to the updategram, some set of boosters with respect to that updategram may also be published. The goal of optimization is to calculate the cheapest plan for updating V (i.e., for producing μ_V) given μ_R, the base relations, and a set of boosters w.r.t. μ_R.

Calculating the cheapest plan can be done using System-R-style dynamic programming [34]. With no boosters, finding the cheapest plan can be done simply by replacing R by μ_R in the System-R algorithm (under the reasonable assumption that accessing μ_R is cheaper than accessing the entire relation R). In the presence of boosters, the System-R algorithm needs to be modified to be able to use them. Specifically, Rule A5 implies that a plan for a sub-expression S of the view V can be replaced by a plan for a booster of S w.r.t. μ_R. Hence, the algorithm needs to be modified to consider the boosters, and to *create* such boosters (using Rule A3). Next, we explain the *augmented* algorithm by contrasting it to the System-R algorithm. We note that the same principles of our algorithm apply to a top-down or transformational optimizer. In addition, we assume that selections and projections are pushed as a later phase and do not consider them further. Our execution model assumes that joins are performed at the querying node, and hence we do not consider a full distributed query processing scenario where semi-joins can be performed anywhere. In our description, we assume that relation R_1 has been updated with updategram μ_1.

The System-R join enumeration algorithm finds the best join order for a query involving a join of n relations, say $R_1, \ldots, R_n$. In its initial iteration, the algorithm finds the best access path to each of the relations mentioned in the FROM clause. The augmented algorithm, in addition, also finds the best access paths to μ_1, and to each of the boosters in $\mathcal{B}$ that are over a single relation. If the algorithm finds that there exists a booster for R_k w.r.t. μ_1 and it is cheaper to access than R_k itself, then it need not consider R_k any further.

In the k-th iteration, the System-R algorithm finds the best plan for joining any sub-expression consisting of k relations in the FROM clause. It does so by trying all possible ways of joining plans of size $k - i$ with non-overlapping plans of size i (or considers only $i = 1$ if it is restricting the search to left-linear trees). The key point is that the algorithm saves a single plan for every sub-expression of size k.[2] The augmented algorithm computes the following in the k-th iteration, also saving only one plan for each construct:

[2]More precisely, it saves one plan for every interesting order on the result, but this is orthogonal to our discussion.

Sub-expressions of size k: in the same way as before, except that if we already have a booster for a sub-expression S that is cheaper than all the plans for S, we do not consider the plans for S any further (Rules A5,A2).

Updategrams for sub-expressions of size k that include R_1: by combining updategrams of size $k-i$ with either boosters or sub-expressions of size i (Rules A1,A2).

Boosters for sub-expressions of size k w.r.t. μ_1 and don't include R_1: these boosters are either already in $\mathcal{B}$, or can be created by combining boosters for sub-expressions of size $k-i$ with boosters for sub-expressions of size i (Rule A4).

We note that super-boosters, which may have been computed for a related view, can always be used instead of boosters at any point in the algorithm. Super-boosters may be more expensive to send than a real booster, but it may still be less expensive than using no boosters.

Note that the running time of the augmented algorithm is no different than that of the basic algorithm. Only one plan is saved for every sub-expression: if the sub-expression includes R_1, the plan will be for an updategram, and if the sub-expression does not include R_1 it will be for the sub-expression *or* a booster. Furthermore, each booster in $\mathcal{B}$ is only considered once during the algorithm.

5.2 Advancing several steps

Our goal now is to advance a view V from version $t_1, \ldots, t_k, \ldots, t_n$ to version $t_1, \ldots, t'_k, \ldots, t_n$, where t'_k may be arbitrarily larger than t_k. We again leverage dynamic programming; the algorithm creates plans for wider steps from the plans of smaller steps. For ease of exposition, assume that V is currently in version 0 and that we need to update it to version x (that is, $t_k = 0$ and $t'_k = x$).

In the first phase, we identify the best plan (using the algorithm described above) for updating V from version 0 to version 1. In the second phase, we identify the best plan for updating V from version 0 to version 2. This can be accomplished by using $\mu^{0,2}_{R_k}$ to transition directly, or by using $\mu^{1,2}_{R_k}$ to transition from version 1. In phase i, we identify the best plan for updating V from 0 to i. This requires considering every updategram that transitions to i (from *any* earlier version). To prune the search space, we only consider version intervals that involve *landmark* versions. Landmark versions are defined by the following procedure. In the procedure, $L(\mathcal{U})$ is the set of ending points of updategrams in the set $\mathcal{U}$.

```
Let $\mathcal{U}$ be the set of available updategrams
while $\mathcal{U}$ changes
   for each u in $\mathcal{U}$ of the form $(t_1, t_2)$
      if $t_1$ is not in $L(\mathcal{U})$, remove u from $\mathcal{U}$
return $L(\mathcal{U})$
```

5.3 Optimization tradeoffs

In general, the utility of updategrams and boosters when computing updates efficiently depends on the particular context in which they are used, and is therefore beyond the scope of this paper. Our goal is to identify optimization challenges that are novel in the presence of updategrams and boosters.

To get a feel for the optimization tradeoffs, we implemented the algorithms described above and tested them under a variety of conditions. We observed the following trends. As the number of updategrams and boosters available to the system grew, our optimization algorithm produced plans with lower expected cost. In many cases, the resulting plans were more efficient because boosters enabled the optimizer to create bushy plans with small intermediate results, rather than left-linear plans. Our experiments revealed that the most significant factor in the expected cost of a plan was its length: an execution plan to advance a view V from version x to version y may go through several intermediate versions. The number of intermediate versions (+1) is called the *length* of the plan. We observed that the estimated cost of a plan is proportional to its length; longer plans end up scanning the same relations multiple times compared to the shorter plans.

The above observation raises an optimization challenge. Exploring the space of plans of all lengths would require using rules A10 and A11, which allow us to coalesce adjacent updategrams and overlapping boosters. However, exploring the complete space of plans with these rules may be prohibitively expensive, and the opportunities for exploiting them grow with the number of updategrams and boosters available. In what follows we describe two heuristics for pruning the search space and describe a set of experiments that validate their impact. We consider two heuristics:

Coalesce: this heuristic first uses the algorithm in Section 5.2 to identify a plan that moves through multiple versions. We then modify the chosen plan by coalescing updates whenever possible, and we choose the cheaper plan of the two.

Greedy: this heuristic greedily explores the set of plans in a way that is biased towards shorter plans. Specifically, instead of considering every possible path to a given version, only the largest possible transition is considered. For example, suppose we are updating from version 1 to 3. After building a plan for interval (1,2), the greedy heuristic will not consider the interval (2,3) if an updategram for (1,3) exists. The important property of this heuristic is that the cost of optimization only depends on the size of the interval, and not on the number of updategrams and boosters available.

5.3.1 Experimental setup

To test the impact of these two heuristics we ran experiments over a 0.1 scale TPC-H benchmark, which defines 8 base relations and 22 queries over these relations. Our experiments did not consider aggregation, so the 12 TPC-H queries involving aggregation were converted to related SPJ queries that could be used to answer the original query.

We generated 200 updates to the relation LineItem. For each update, we generated boosters for each relation that shares a key with the updated relation (i.e., Orders, Parts and Suppliers). Each update is published as a separate updategram. As we explain shortly, the different experiments will add updategrams to this basic set.

On the query side, we generated 100 version intervals whose end-points lie in the range [0-200]. For each interval, we ran the augmented System-R optimizer to find the optimal plan for advancing each query in our collection across the interval. That is, for a given interval

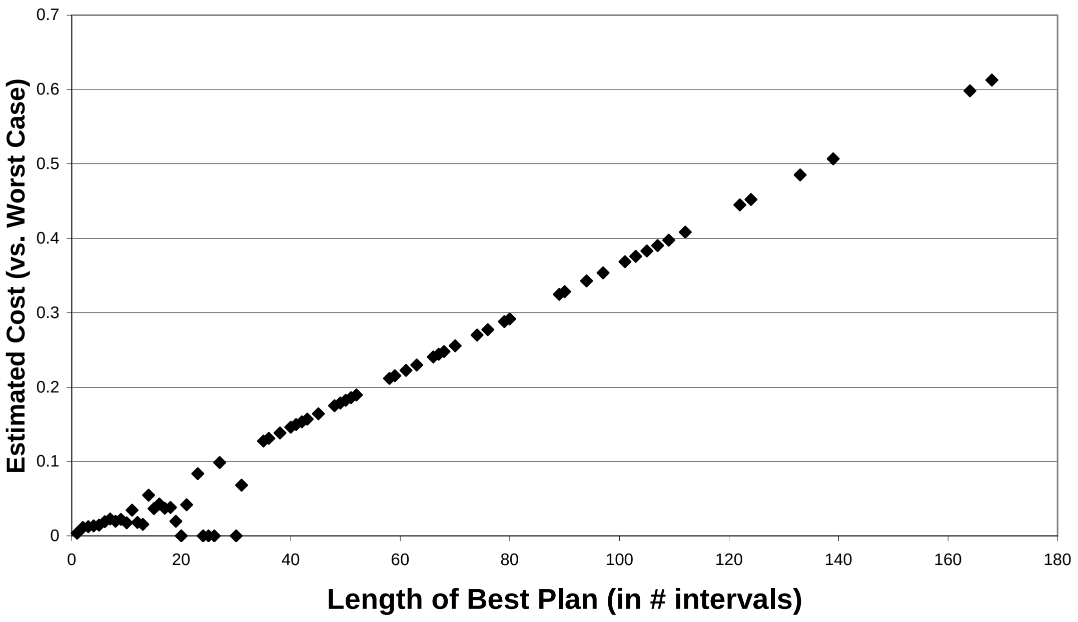

Figure 4: Effect of Plan Length—This figure shows the estimated cost of plan execution as a function of the number of versions through which the plan transitions. Plans with fewer transitions are cheaper (even if the transitions are large).

(x, y), we assume that the queries are consistent with $\mathsf{LineItem}^x$ and we wish to refresh them to reflect the contents of $\mathsf{LineItem}^y$.

We report the average optimization times for the query, and the ratio of the estimated cost of execution to the estimated cost of re-materializing the query from scratch. In our cost model, we assume that all data sources, updategrams and boosters are remote. Note that the sizes of the base relations, updategrams and boosters are available to the optimizer.

5.3.2 Experimental results

The first two graphs illustrate the challenge our heuristics are facing. Figure 4 shows that the estimated cost of plan execution increases as a function of the number of versions through which the plan transitions, i.e., the plan length. These results are reported as a fraction of the worst case, which is recomputing the view from scratch. Figure 5 shows that as the number of updategrams increases, the optimizer is able to find shorter plans.

Hence, as the number of updategrams available grows, the opportunities for finding shorter plans increase, but searching the entire space of plans may be prohibitively expensive. The results of employing our heuristics are shown in Figures 6 and 7. Figure 6 plots the expected cost of the plans produced by our optimizer as a function of the number of updategrams available. The "Base" line in the figure shows the performance for the algorithm described in Section 5.2. The graph shows that the expected execution time drops off dramatically with the number of updategrams available. For example, with 2000 random updategrams available, the cost of execution is a factor of ten less than the case with only the single-step updategrams available.

The "Coalesce" line shows the performance for our first heuristic. The expected cost of

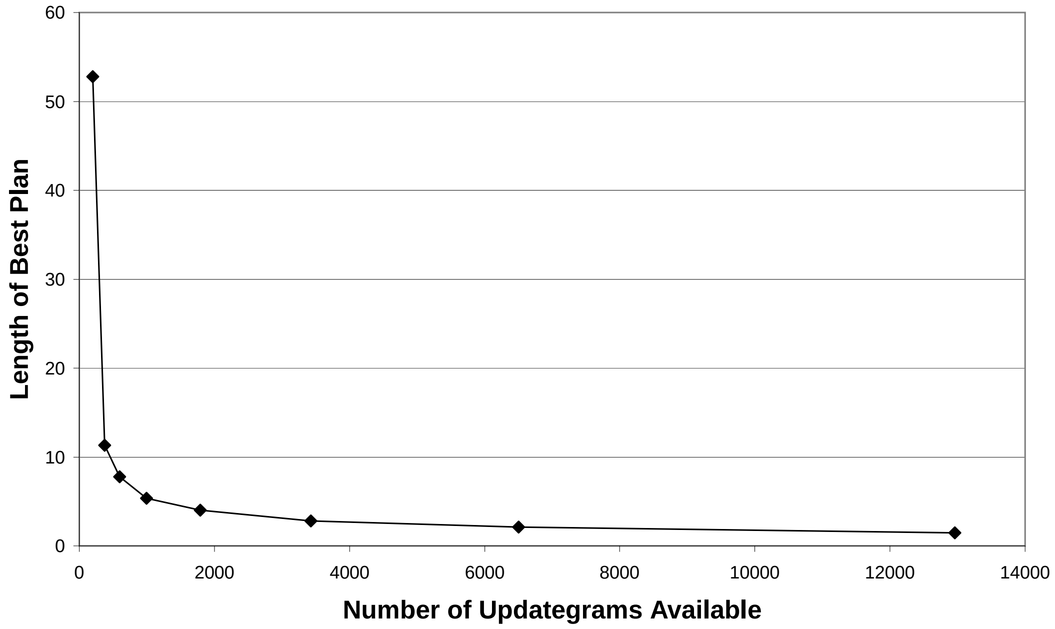

Figure 5: Average plan length as a function of the number of updategrams available.

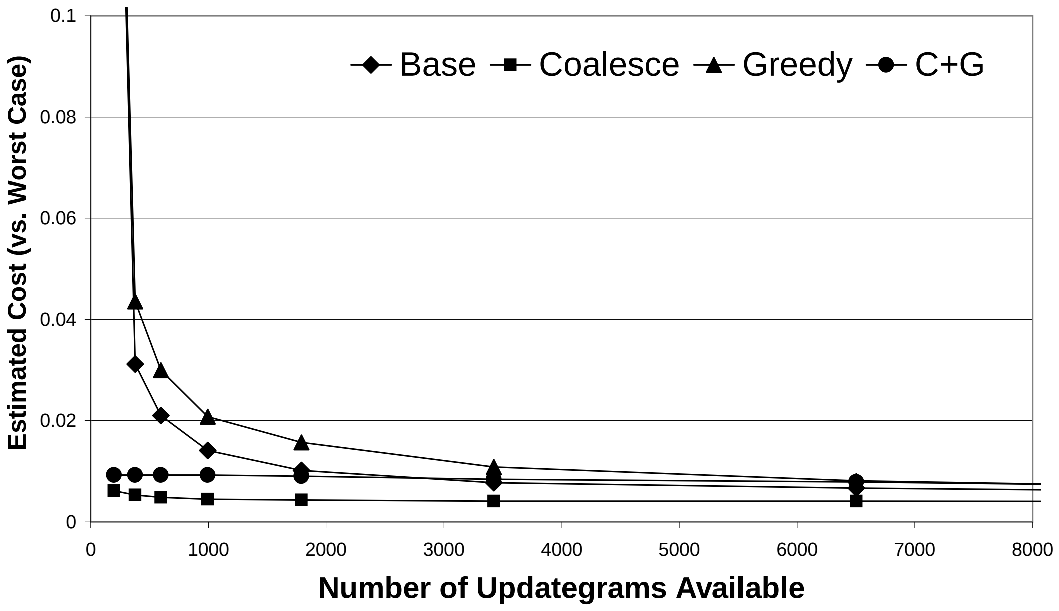

Figure 6: Estimated cost of execution as a function of the number of updategrams available. Coalescing produces more efficient plans; the greedy heuristic identifies only slightly less efficient plans.

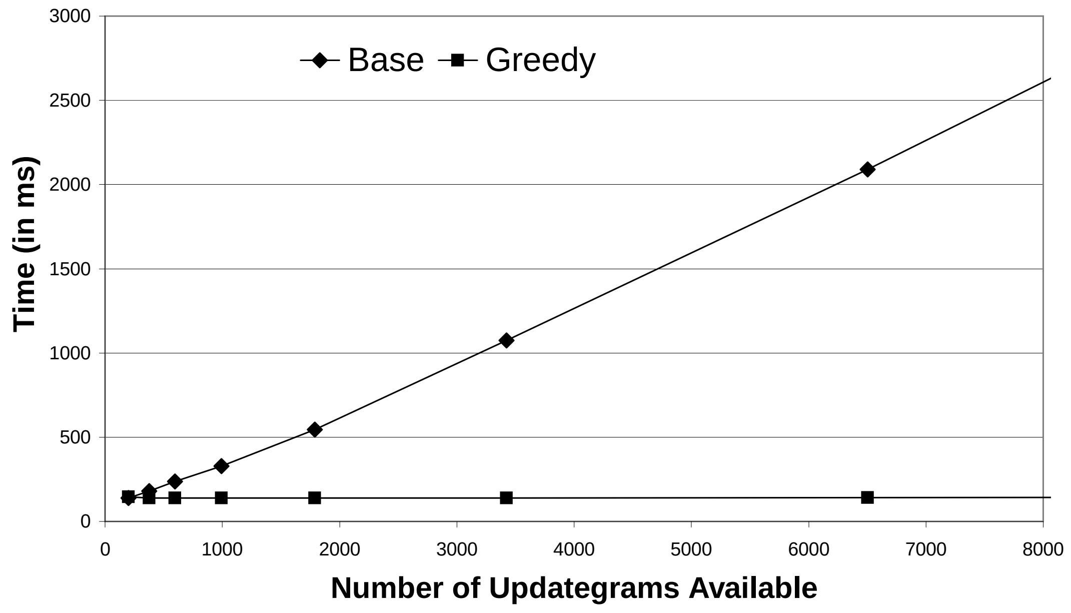

Figure 7: Actual cost of optimization as a function of the number of updategrams available. The greedy algorithm does not depend on the number of updategrams available.

"Coalesce" is ≈28% of the expected cost for "Base".

The "Greedy" line shows that the greedy heuristic quickly identifies plans of minimal length. This allows the algorithm to find reasonable plans, but not necessarily optimal plans. The figure shows that the plans produced by "Greedy" are ≈30% more expensive than those produced by "Base". Figure 7 illustrates the advantage of "Greedy." The optimization cost depends only on the length of the interval; it is constant in the number of available updategrams (since only one plan is considered for each landmark).

Finally, we include the combination of both heuristics. These results are shown as "C+G" in Figure 6. The same trends hold when comparing "C+G" to "Coalesce" or "Greedy": "C+G" exhibits better execution times than "Greedy" and better optimization costs than "Coalesce" (in Figure 7 this curve coincides with the curve for "Greedy" since only one additional plan is considered).

In summary, while our experiments are not comprehensive, they suggest that with our two heuristics it is possible to efficiently obtain short, low-cost plans even in the presence of a large number of updategrams and boosters.

6 Related Work

Our work builds on the rich literature on view maintenance and updates. The key novelties in our work arise from the fact that we are interested in open architectures in which updategrams and boosters are published and updates can be propagated using a variety of strategies. Hence, we focus on the questions of creating and reusing updategrams and boosters and on their use in optimization.

Surveys of view maintenance are given in [37, 15]. Mohan [27] surveys the related area of using data replication to increase availability. Several techniques for efficiently computing updates are described (e.g., [16, 5, 1]). In the context of data integration or warehousing, the focus has been on methods that would enable a view to be updated *without* having to access the base relations, i.e., to guarantee that views are *self-maintainable*. In general, [14] shows that without knowledge of keys, views that join two or more relations cannot be updated without some external knowledge (i.e., the contents of the other relations). In [32] it was shown how to slightly modify a view definition to make it self-maintainable, and in [23], it was shown which other auxiliary views should be maintained in a warehouse in order to speed view maintenance.

In the Heraclitus [12] System, updates are also treated as first-class citizens, but there it was for the purpose of accommodating *hypothetical* queries over possible different states of the data. In the Propagation Manager [7] component of SIES, updates are transformed and propagated using scripts whereas we consider declarative views.

The issue of developing policies for guaranteeing fresh data has also received attention. For example, Vista [37] identifies two basic strategies that guarantee fresh data: immediate and deferred updates. An alternative that does not guarantee freshness is periodic updates. Our work is agnostic towards these distinctions, allowing data providers to choose the frequency with which materialized views are refreshed and updates published. A analysis of performance implications can be found in [18].

There is a large body of work that uses data replication to increase availability (see [27] for a recent tutorial). All such systems face the problem of keeping replicas consistent. Strategies for doing so include primary-copy approaches [2, 30], in which clients write to a primary master, only interacting with slave nodes for reads or on failure of the primary, and lazy strategies (e.g., [24]), in which operations are loosely ordered (i.e., consistency is not strictly guaranteed). In [36] any replica can be updated; conflicts are resolved using anti-entropy, during which replicas progress towards reaching eventual consistency. Some systems keep all replicas synchronously up-to-date; the performance implications of doing so are discussed in [13]. In our work, we assume updates to base data only originate in one place, and therefore, there is no need for complex consistency protocols.

A common technique for ensuring data consistency is to use triggers. An algorithm for automatically constructing triggers for updating views can be found in [6]. This assumes that the base relations are aware of the views that will require maintenance. In our context, however, it is not possible in general to make that assumption.

Finally, two related areas that we have not touched upon include choosing which views to materialize at a particular node, and the treatment of *streaming data*. Streams can be viewed as an extreme case of update propagation in which there are a stream of updategrams. The value of boosters in such a context is hinted at in [3]. However, the application of our techniques to streams is future work.

7 Conclusions and Future Work

The efficient management and propagation of updates is a crucial component of large-scale data sharing. This paper lays the foundation for update management in a variety of system scenarios, by proposing a framework in which updategrams and boosters are published as first-class objects. We described a set of rules that guides the use of these constructs and covers both GAV and LAV mediation formalisms, as well as views with aggregation. We then described an optimization algorithm that is able to use updategrams and boosters to produce efficient view update plans, and described practical heuristics for pruning the search space of query plans, while producing near optimal plans.

There are several directions we plan to pursue next. First, we will study the question of *which* views to store on nodes in the network in order to increase data availability and improve performance. Second, we plan to apply our framework and methods to a specific data sharing architecture.

8 Acknowledgements

We would like to thank Rachel Pottinger for her comments on earlier drafts of this paper.

References

[1] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, 1998.

[2] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *ICSE*, 1976.

[3] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[4] P. A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Database Management for Peer-to-Peer Computing: A Vision. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2002.

[5] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, pages 61–71, 1986.

[6] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.

[7] C. Constantinescu, U. Heinkel, R. Rantzau, and B. Mitschang. A system for data change propagation in heterogeneous information systems. In *Int. Conf. on Enterprise Information Systems*, 2002.

[8] O. Duschka, M. Genesereth, and A. Levy. Recursive query plans for data integration. *Journal of Logic Programming, special issue on Logic Based Heterogeneous Information Systems*, 43(1):49–73, 2000.

[9] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *Proc. of PODS*, pages 109–116, Tucson, Arizona., 1997.

[10] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, 2003.

[11] M. Friedman, A. Levy, and T. Millstein. Navigational plans for data integration. In *Proceedings of AAAI*, 1999.

[12] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM TODS*, 21(3):370–426, 1996.

[13] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, pages 173–182, 1996.

[14] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *EDBT*, pages 140–144, 1996.

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

[16] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally (extended abstract). In *SIGMOD*, pages 157–166, 1993.

[17] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *Proc. of ICDE*, 2003.

[18] E. N. Hanson. A performance analysis of view materialization strategies. In *SIGMOD*, 1987.

[19] R. Hull. Managing semantic heterogeneity in databases: A theoretical perspective. In *Proc. of PODS*, pages 51–61, Tucson, Arizona, 1997.

[20] P. Kalnis, W. Ng, B. Ooi, D. Papadias, and K. Tan. An adaptive peer-to-peer network for distributed caching of OLAP results. In *SIGMOD*, 2002.

[21] Kazaa. Homepage: http://www.kazaa.com, November 2003.

[22] A. Kementsietsidis, M. Arenas, and R. J. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *Proc. of SIGMOD*, 2003.

[23] W. Labio, D. Quass, and B. Adelberg. Physical database design for data warehouses. In *ICDE*, pages 277–288, 1997.

[24] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4), 1992.

[25] M. Lenzerini. Data integration: A theoretical perspective. In *Proc. of PODS*, 2002.

[26] A. Y. Levy and Y. Sagiv. Queries independent of updates. In *VLDB*, pages 171–181, 1993.

[27] C. Mohan. Caching technologies for web applications – a tutorial. In *Proc. of VLDB*, 2001.

[28] M. K. Mohania and Y. Kambayashi. *Data and Knowledge Engineering*, 32(1):87–109, 2000.

[29] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. Peerdb: A p2p-based system for distributed data sharing. In *ICDE*, Bangalore, India, 2003.

[30] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB Journal*, 8(3–4), 2000.

[31] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating web data. In *VLDB*, 2002.

[32] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *PDIS*, pages 158–169, 1996.

[33] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[34] P. G. Selinger, M. M. Astrahan, D. D. Chamberlain, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[35] A. P. Sheth and J. A. Larson. Federated database systems for managing, distributed, heterogenous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[36] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, pages 172–183, 1995.

[37] D. Vista. *Optimizing Incremental View Maintenance Expressions in Relational Databases.* PhD thesis, University of Toronto, 1997.

[38] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, 1992.

[39] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *ICDE*, 2003.