Streamed Update Propagation in a Peer Data Management System

Xin Dong, Lin Liao and Zizhen Yao
Department of Computer Science and Engineering
University of Washington

Abstract

Peer to peer distributed system has been widely studied in recent years because of its appealing properties: scalability, robustness and no need for central control. Peer data management system (PDMS) rises up above P2P system to exploit the power of available data management technology. The goal of PDMS is to facilitate wide-scale data sharing within and across communities. This paper focuses on one particular problem regarding to data sharing, the update propagation problem. The problem is defined as follows: given multiple data sources and materialized views based on these base data, all of which are distributed on different hosts across Internet, how to synchronize the views when the data source has been updated?

In this paper, we proposes eager join, lazy merge policy that preserves resources without comprising the performance, and improves performance without introducing great overhead compared to traditional push, pull or hybrid schemes. In addition, we analyze the inherent tradeoff between response time and resource utilization in the system, and proposed an adaptive strategy, which exploits the available resource to minimize response time when resources are bountiful, and degrades gracefully when resource shortage occurs.

The experimental results show that when resources are abundant, the performance of our approaches approximates that of push with less resource consumption; when resources are insufficient, our approaches uses similar amount of resource as pull, while still has performance gain.

1 Introduction

Peer data management systems (PDMSs) aims at facilitating users to share data across local or wide-area networks in an ad-hoc, highly dynamic distributed environment [1]. In order to achieve high performance and availability, data are replicated and cached in strategic locations in the form of materialized views. A key challenge in this mechanism is to efficiently manage the propagation of updates from original sources to materialized views. It plays a critical role in providing efficient and up-to-date answers to queries posed by peers.

Due to the characteristics of peer-to-peer systems, such as decentralized control, symmetric communication, unmanaged update and query patterns, and resources restriction, the view maintenance on updating is much more difficult in PDMSs than in traditional distributed database systems. Furthermore, due to the complex structure of database relational model, the view maintenance problem in PDMS is more difficult than the consistency problem in web caching or in other peer-to-peer file sharing systems like Napster [2], Gnutella [3].

In order to avoid transmitting the whole base table, which causes huge traffic, [4] suggests publishing updates in objects called updategrams. In addition, if the view has a join on the updated source, denoted as \( R \), and some other sources, denoted as \( S \), \( S \) should generate and publish boosters, which are subsets of the tuples in \( S \) that may join with tuples in corresponding updategrams. Hence, the update to \( R \bot S \), called delta view, can be computed by joining the updategram and the booster. A great amount of transportation and computation could be saved in this way.

In many applications, such as stock market databases and sensor network databases, source data update much more frequently than users query the view. These queries require up-to-date data. Different policies can be used in such applications to decide when to publish updategrams and boosters: push, pull and hybrid. In the push mode, updategrams and boosters are eagerly propagated as soon as they are produced. This method maintains strong consistency and provides fast response. However, it leads to huge overhead because for each individual update, boosters need to be computed and transmitted, and the view needs to be updated correspondingly. The former leads to a high transmission cost, as the overlap of boosters of different versions cannot be removed and thus transmitted repeatedly. The latter leads to high computation cost and I/O cost. When the network bandwidth, I/O or CPU resources are
limited, the overhead of push can be overwhelming and jeopardize the performance of the whole system.

In the pull mode, the view side asks for updategrams upon arrival of query, and the corresponding boosters are generated and transmitted on the fly. When updates are far more frequent than queries, this scheme reduces overhead and saves resources. However, the response time increases drastically due to the network transmission involved to obtain the updategram and the booster. In addition, it may create huge burst traffic that imperils the system.

There are also some hybrid mechanisms. For example, the data sources can push the updategrams periodically. Also, the view side can periodically poll the data sources. However, neither guarantees that the views are up-to-date. Thus they still need to refresh data when queries come. It is hard to make the trade-off between resources usage and data staleness when deciding the intervals between pollings or pushes. Finally, the fixed “schedule” cannot adapt to various amounts of available resources.

The target of our update propagation management is to allow high performance in presenting up-to-date answers to queries and graceful degradation as the resource utilization approaches its capacity. When there are plenty of resources, our system makes the full use of available resources and attains a high performance. When the resources are inadequate, it decelerates updates and reduces resource consumption, while maintaining acceptable reply latency.

We achieve this goal by using the policy of eager join, lazy update. As network is the bottleneck in response, updategrams and boosters are generated and pushed eagerly so that when queries come, usually little time or even no time is spent on waiting for network transmission. This greatly reduces latency. However, unlike in eager mode, the propagation of updategrams and boosters does not trigger the updates of the original view. Because I/O operation is unavoidable at response time, we delay view updates as long as possible to reduce computation and I/O burden.

The experimental results show that when resources are abundant, the performance of our approach approximates that of push with less resource consumption; when resources are insufficient, our approach uses resources close to pull, while still has performance gain.

1.1 Contributions

In this paper, we try to tackle the management of updates propagation. The contributions of this paper are summarized as follows:

1. We design a new mechanism of updates propagation. Unlike the hybrid policies, which are essentially the intermediate approaches of push and pull modes, our mechanism applies different ideas of Push and Pull at different phrases. By using push in propagation, we remove the high-cost away from the critical path of answering queries, introduce the possibility of streamed operations, and present a relatively global view of the data to the view side to enhance the chance of optimization. By using pull in update, we reduce the computation and I/O cost.
2. We develop the streamed view update algorithm based on hash table. By maintaining hash pools, we remove the overlap of updategrams while maintaining its semantics, speed up streamed join computation, remove duplication of boosters and delta views without big overhead, and enable lazy updates at a low cost.
3. We introduce dynamic strategies to adapt to the amount of resources. The speed of sending boosters and the speed of updating views are dynamically adjusted according to the availability of bandwidth, CPU and I/O resources. As a result, when the resources are limited, the performance does not reduce dramatically but degrades moderately.

The paper is organized as follows. Section 2 discusses related work. Section 3 describes the context for this work, and then formally specifies the basic model and addresses our assumptions. Section 4 describes the system in detail and section 5 defines the cost model. Results of an experimental evaluation of our system are presented in Section 6. Future work is discussed in Section 7 and we conclude in Section 8.

2 Related Work

The formal rules governing the use, creation and combination of updategrams and boosters in updates propagation in PDMSs were proposed in [4]. It also discusses how to extend the System-R styled dynamic programming algorithm to apply updategrams and boosters in query optimization. While the paper provided evidence that this approach is promising, it left many questions unanswered and it was based on the assumption of push mode.

The problem we are trying to address in this paper is in some sense, a generalization of the view maintenance problem in database system, or the replica control problem in Peer-to-peer file system. There are three aspects of this problem: update
optimization, materialized view maintenance and cache synchronization. As to view maintenance, we can divide it into two parts based on whether all sources and views are in the same location or the sources are geographically distributed. We examine these fields respectively.

How to update materialized views efficiently has been investigated seriously since late 1980’s, especially in the area of distributed data replication and data warehousing. [8] presents an analytical performance comparison of answering queries over a single view by 1) re-computation, 2) synchronize views with data sources immediately and 3) deferred synchronization. That study shows that for different types of views (e.g. SP, SPJ, aggregate view, etc), the best strategy is different.

Since early 1990’s, quite a few algorithms of updating materialized views incrementally have been presented. [9] uses algebraic operations to derive the minimal relational expression that computes the change to SPJ views. One of the most widely used algorithms is counting algorithm in [10]. The counting algorithm tracks the number of alternative derivations for each tuple in a view and can be used for non-recursive views with negation and aggregation. For recursive views, an algorithm called DRed is also presented in [10]. A good survey of view maintenance algorithms can be found in [11].

Our work is different from these works in many ways. First, those algorithms are used in a local machine or a high-speed local network, so network bandwidth is not a big concern. While in peer-to-peer system, network bandwidth becomes a big concern. Second, the availability of resources in those systems is pretty stable. But this does not hold in peer-to-peer system.

Integration of data from multiple databases is an important problem. Except for several simple views, maintaining a view requires access to data that is not available in the view itself. However, the cost of accessing the underlying databases in a distributed database system can be expensive and thus may prevent fast response. So far there are two ways to solve this problem: using self-maintainable views and using updategrams plus boosters.

View maintenance is more complex in distributed database system because it requires access to data that is not available in the view itself. [13] defines self-maintainable views, that is, a category of views that can be updated using only the contents of the view and the database modifications, but without accessing any of the underlying databases. It shows that only a very limited number of views are self-maintainable, and suggests choose such kind of views from the beginning of the design to avoid the traffic problem in data update, which is not practical in real application. [12] gives an algorithm to make a view self-maintainable by using several auxiliary views stored in the same location. As an alternative to our approach using updategrams and boosters, the self-maintainable-view approach generates less traffic and works well when related peers are not reliable. However, maintaining a set of auxiliary views means more storage space and more cascading maintaining computation at the view side. In contrast, our approach divides workloads among the booster side and the view side, divides resource usages among network, I/O and CPU. Additionally, the main idea of eager join, lazy update in our approach can also be applied to the self-maintainable-view approach.

When data sources and views are at the different locations, views synchronization problem occurs, that is, views needs to be updated according to the updates to the base table. There are in general four mechanisms as to when and how to propagate the updates from source to views: (i) eager push (ii) pull on query (iii) divergence-based push and (iv) polling. Eager push maintains strong consistency: whenever there is an update at the source, it is propagated to all views that are dependent on this source. This mechanism maintains the strong consistency, but the overhead (bandwidth, disk access and CPU time) is big. For sources that have many dependent views and/or updates, it may become the bottleneck of the system [7]. Our work is different from push in that, our refresh rate is adaptive to available resources to avoid the system overload.

The latter three approaches maintain best-effort synchronization in that the views can be in stale state. Divergence push mechanism typically place bounds on one or more divergence parameters. For example, some systems place bounds on time (temporal transaction control) or the number of update transactions (value-based divergence control). These systems are usually based on weak consistency models. Mariposa [6] is a temporal divergence-based system that provides standard consistency guarantees using two-phase commit at the cost of greater staleness. Mariposa also proposed a dynamic refresh strategy that slows down refreshing rate by default, and speeds up on positive feedback when resources are bountiful. Similar to Mariposa, our strategy is also a dynamic approach that adapts to availability of resource. It is different from Mariposa in that, In PDMS system, strong consistency is required, thus best-effort synchronization is not enough in this case. In addition, the granularity of data is different. In Mariposa, units of data are semantics-free objects, where in PDMS, we are dealing with tuples of database with more complicated semantics.

In Pull on query mechanism, views check the version of the source when queries arrive. If the view
is based on is an up-to-date version of the source, it executes query directly, otherwise, view update is conducted before query execution. This approach incurs greater delay for the query, but avoid the waste of update propagations that are not needed. By contrast, our approach avoids such delay when resources are bountiful by refreshing ahead of time, and it incurs about the same amount of overhead with pull when resources are limited.

In polling mechanism, a view periodically polls the sources that it depends on, if the source is updated, and it pulls the updates, and refresh itself accordingly. Instead of polling, we rely on updategram-side, and booster-side pushing to increase freshness of data before user query to avoid unnecessary polling requests.

3 Problem Definition

The problem we are trying to solve is defined as follows: Given two base tables R and S, and the view V joins both R and S, all of which are located on different hosts across Internet, if one of the table R is updated, how to propagate the update to V efficiently? By efficiency, we mean minimal response time with available resources. More specifically, when resources are sufficient, the primary goal is to reduce response time using whatever resources available. When resources are in shortage, the emphasis is to preserve resources so as to avoid system overload and deliver best performance.

We base our solution on the framework addressed in [4]. When R is updated, the updategram side packs updated tuples into the updategram, and sends it to the booster side and the view side simultaneously. The booster side semi-joins S with the updategram to produce the booster and then delivers it to the view side. The view side joins the updategram and the booster to produce its update, ∆V. Finally V is updated according to ∆V. The whole process is indicated in Figure1.

![Figure 1. Problem Definition](image)

We make the following assumptions for the rest of the paper:

- Both individual updategram and booster are small enough to stay in memory. The join of the updategram and the booster, i.e. the delta view, can also stay in memory.
- Base tables of R, S and the materialized view V are too big to fit in the memory. Accessing them need I/O operations.
- The base table of boosters, S, has index on the join field(s). There is no index on the view. Besides, there is no index on the transmitted updategrams and boosters.
- Although there are cases that using base tables instead of updategrams and boosters can perform better, we only consider the common case of using updategrams and boosters.

Some related issues including materialized view placement, schema heterogeneity, underlying network support, are out of scope of this paper.

4 System Description

The fundamental idea in our streamed update propagation mechanism is to make full use of available resources to improve performance, and at the same time, have the flexibility of adapting to limited resources to avoid dramatic performance degradation.

The above design principle has two correlated components. The first is to reduce response time. This is attained by the “eager join, lazy update” policy, which removes the bottleneck away from the critical path, described in Section 4.1. The second issue is to reduce resource usages. We achieve this by using the dynamic strategy stressed in Section 4.4. The streamed view update algorithms explored in Section 4.2 and 4.3 facilitate both of them.

4.1 Propagation Mechanism Overview

In order to reduce response time and at the same time save resources, we need to distinguish the workloads according to whether it forms the bottleneck in response time. Those that do should be removed from the critical path and done eagerly provided the resources are enough. Those that do not should be done lazily so as to reduce overhead without impacting performance too much.

Due to the low speed of network transmission relative to I/O operation and CPU computation, pull mode has a much longer latency because it has to perform network transmission during query response. Even worse, the pull mode creates high burst traffic. This can increase response time even more significantly and may cause network congestion. To
avoid this, we decide to propagate updategrams and boosters in an eager way. Whenever an updategram is produced and the network can afford to transmit it, it is pushed to the booster side and the view side. Similarly, whenever the booster side receives the updategrams, it eagerly produces the corresponding boosters and pushes them to the view side as far as the network resources can sustain it. Here, we assume that the lower level of the network can send those data in a smart way that large amounts of small data can be transmitted without causing too much overhead.

We can go a little further to join the updategrams and boosters eagerly, that is, whenever a booster arrives at the view side, it is joined with the corresponding updategram right away. The motivation for this strategy is, joining in memory has little overhead and there is little room for improvement by doing lazy join, even if it can accumulate boosters to exploit their overlap. Furthermore, if the boosters are accumulated to be too large to fit in memory, we even have to pay for I/O cost.

I/O operation can also be expensive because of large table scan. Therefore, if the materialized view is large, I/O may be the dominant factor in response time. However, we observe that such cost is inevitable no matter what strategy is used, since we have to scan the view at least once for query processing. In addition, delta views are not large and can reside in the memory. This fact helps us design the algorithm to update view while processing the query with only one table scan. The extra workload comparing to query processing itself is neglectable. Based on this, we merge delta view into original views as lazy as possible. In the ideal situation when the memory is big enough, the CPU is fast enough, and the queries are not too sparse, view update is done at the time of query.

Eager join and lazy merge introduce longer intervals between the time when the data arrive and the time when the data are handled, which makes it possible to do some further optimization at the view side. Firstly, when updategrams and boosters are buffered at the view side, the server can stream the join operation and perform it as a back-end process when the CPU is idle. Secondly, the long intervals present a relatively global view of data and thus better chance for optimization. The detailed algorithm is described in section 4.3.

4.2 Booster Generation

According to the definition, a booster is generated by semi-joining the base table with the updategram at the booster side. Either using hash join or sort merge join, the cost of semi-join is at least the cost to scan the base table. The I/O cost can be prohibitive especially when updategrams arrive frequently. A better solution is to create an index on the join field(s). Therefore, whenever an updategram arrives, we only locate the matching tuples and ignore others. This can eliminate unnecessary I/O and computation. The low cost presents the possibility to produce boosters at background when CPU and I/O are idle. Furthermore, in this scheme, the cost to generate one booster for several updategrams and the cost to generate boosters separately for individual updategrams is roughly the same. This allows us to generate boosters eagerly at no higher cost than doing it lazily.

4.3 View Update Optimization

The operation on the view side can be defined as:

\[
\Delta \text{View (Delta View)} = \text{updategram} \triangleright \text{booster} \\
\text{View'} = \text{View} \oplus \Delta \text{View}
\]

There are several aspects we can explore to improve performance while reducing resource usage.

- Several updategrams may operate on the same tuple. Removing these tuples can reduce the computation of joining updategrams and boosters. However, we need to consider the order of those updategrams and maintain the semantics of those updates.
- There may also be duplications in delta views. Removing it reduces not only response time but also computation resources and memory requirements.
- By choosing proper joining method and maintaining appropriate indexes, joining updategrams with boosters and updating original views according to delta views can be done efficiently.

To explore all of these optimization spaces, we design a streamed view update algorithm based on hash pool. The data structure at the view side is shown in Figure2.

![Figure2: Data Structure at the view side](image-url)
4.3.1 Removing Overlaps on Updategrams

A challenge in removing overlaps of updategrams is to remove duplications while maintaining the semantics. To achieve this goal, we use 2 hash pools, updategram insert pool and updategram delete pool. Each pool is maintained as a hash table on the joining columns. Due to the eager join of updategrams and boosters, usually updategrams come and leave frequently and thus the two pools do not hold a lot of updategram items. Therefore, we can set the pool size to be relatively small and hold it in memory.

When an insertion updategram item comes, it is put into the updategram insert pool with the version information according to its hash value, provided it is not duplication. When a deletion item comes, before putting it into the delete pool, we first check whether the same tuple appears in the insert pool. If so, the tuple is removed from the pool and the deletion item is discarded. In this way, overlaps of updategrams are removed as early as possible. To further optimize, if there are select predicates related only to updategrams, we can apply selections before putting them into the pools.

The complete algorithm is described in Algorithm_1. In this algorithm, we define the function of poolEntry to insert a tuple in the delete or insert pool of specified pool set. We use hash functions of hashInsert, hashDelete, hashLookup and hashGet to insert an item, delete an item, judge whether a tuple is in the hash table, and get the whole tuple with the given key respectively.

Algorithm_1

```java
poolEntry(tuple, type, hashPool)
{
    if (type == insert)
        hashInsert(tuple, insert, hashPool);
    else if (type == delete)
        if (hashLookup(tuple, insert, hashPool))
            hashDelete(tuple, insert, hashPool);
        else hashInsert(tuple, delete, hashPool);
}
select_on_updategram;
poolEntry(ugram, ugram.getType(), ugramPool);
```

The following two theorems show the correctness of the algorithm. It helps to removes the overlap of updategrams as early as possible.

**Theorem.** Let $UGRAM'$ be a new updategram attained after Algorithm_1 by listing all the items in the delete pool in any order and then all the items in the insert pool in any order. $UGRAM'$ has the same semantics as original updategram series.

4.3.2 Joining Updategrams and Boosters

When the booster comes, it is joined eagerly with corresponding updategrams as far as the CPU resources permit. To maintain the semantics, a booster is first joined with corresponding updategram items in the delete pool and then joined with that in the insert pool. As the updategram pool is hashed on the joining columns, the join can be done efficiently. The result is put into the delta view insert pool, and the delta view delete pool in the same way as for updategrams. We handle with boosters in order. After processing the last booster item with a certain version, all updategram items with the same version are deleted. Again, we can apply related selections on boosters before joining and on the results before putting it into hash pools. The whole algorithm is described in Algorithm_2.

Algorithm_2

```java
select_on_booster;
if (ugram=hashGet(booster getKey(), delete, ugramPool))
    if (ugram getVersion()=booster getVersion())
        ∆view = ugram ∆ view
        select on ∆ view;
        poolEntry(∆ view, delete, ∆ view Pool);
        if (is the last item of current booster)
            for (all items u in Updategram Delete Pool)
                if (u getVersion()=booster getVersion())
                    hashDelete(ugram, delete, ugramPool)
。

if (ugram=hashGet(booster getKey(), insert, ugramPool))
    if (ugram getVersion()=booster getVersion())
        ∆view = ugram ∆ view
        select on ∆ view;
        poolEntry(∆ view, insert, ∆ view Pool);
        if (is the last item of current booster)
            for (all items u in Updategram Insert Pool)
                if (u getVersion()=booster getVersion())
                    hashDelete(ugram, insert, ugramPool)
。

The eager join policy usually disallows removing the overlap of two updategrams over a large version gap. On the other hand, the lazy update policy makes it possible to accumulate a large amount of duplications in delta view. By maintaining delta view insert pool and delta view delete pool, which should be much larger than the pools for updategrams, we further remove this part of duplication. Essentially, ∆view can be regarded as an updategram on the view. We use the hash pool as a uniform implementation of an updategram. This both helps to remove duplication and constructs a hash table for probing.
4.3.3 Merging Original View and Delta View
The original view is updated lazily when answering queries or when the delta view pool is 80 percent full. When merging the delta view back to the original view, we take the advantage of the hash table on delta view and do hash join. Unlike updategram pools, delta view pools are hashed on the primary key of the view. If there is no need to remove duplication, we can firstly merge the original view with the delta view in the delete pool using the index, and then simply concatenate the delta view in the insert pool. Or else we can merge the two delta views with the original views at the same time, so long as we ensure that deletions are merged earlier than insertions. After this, the delta view pools are purged and the view is up-to-date.

4.4 Adaptive Strategy
The algorithms stated above work well when there are bountiful resources. It makes full use of available resources to delivery best performance. However, in the situation of limited resources, the same policy may consume too much resource and lead to overload. Our system adapts to this situation by reducing booster push speed and slowing down updates. In P2P system, bandwidth or I/O task variations can arise because of fluctuating workloads. The adaptation can be valuable in coping with such variations.

4.4.1 Adapting Booster Push Rate to Available Bandwidth
As the number of updategrams becomes larger, the possibility that two updategrams share the same booster tuple increases. The booster size grows sublinearly. Delivery of a booster for several updategrams is more efficient than delivering separate boosters for each updategram. When bandwidth is not enough, by merging several updategrams before generating and transmitting boosters, the total amount of traffic between the booster side and the view side can be reduced.

Assume the values on the join field distribute uniformly both in the base table of booster side, and in the updategram. Let $S$ be the size of the base table at the booster side, $V$ be the number of distinct values on the join field in the base table, and $T(n)$ be the expected number of distinct values on the join field for $n$ items in the updategram. We have the following estimation:

$$ T(n) = T(n-1) + (1 - \frac{T(n-1)}{V}) $$

$T(n-1)/V$ is the probability that the value of the joining field in the $n$th updategram tuple is the same as some former one. $T(n)$ can be computed by solving the recursive relation:

$$ T(n) = V(1 - (1 - \frac{1}{V})^n) $$

The booster size can be estimated as

$$ S(n) = T(n) \frac{S}{V} = S(1 - (1 - \frac{1}{V})^n) $$

From this formula, we notice that $S(n)$ increases much slower than $n$, and finally converges to $S$. This suggests that combining boosters of individual updategrams can achieve great saving.

Let $B_0$ be the bandwidth capacity that is required when we send each booster eagerly, $B$ is the available bandwidth. If $B < B_0$, then the system can only sustain generating and sending no less than $n$ boosters at a time:

$$ S(n) = \frac{B}{nS(1)} $n\leq B_0 $$

To simplify the computation, we get Tyler expansion of order 2 for $(1-1/V)^n$. Thus we can set the booster push rate to be:

$$ n = 2V(1 - \frac{B}{B_0}) + 1 $$

4.4.2 Adapting View Updating Rate to Available CPU and I/O Resources
In our system, we set up a certain size for the hash pools at the initial time. The delta view lying in the hash pool is not merged to the original view until the pool is 80% full, which assures the performance of hash functions. The pool size can be enlarged in the extreme condition when CPU or I/O is in shortage. In this case, merge rate slows down and I/O consumption reduces, while memory usage expands. The key point here is to use space to save time.

Since I/O is more likely to be the bottleneck of the system, we focus on reducing I/O operations. Let $I_0$ be the I/O capacity that is required for updating with the hash pool of the initial size, $I$ is current available I/O capacity. If $I < I_0$, then the update rate should reduce by $I/I_0$. Approximately, we can increase the pool size by $I_0/I$.

5 Cost Model
The major goal our project is to adapt the behavior of the system to available resources. Therefore, we need a cost model to estimate the expected usage of the resource, and then adjust the strategy so that the available resource can meet the demands.

In the following discussion, the resources we are interested in include CPU, network bandwidth and I/O
ports. The availability of these resources is measured by their capacities, defined as C for CPU, B for bandwidth, and I for IO. The precise definitions of capacities for bandwidth, CPU and IO are given below:

**Bandwidth** (or B): the number of tuples that can be delivered by network per unit time.

**CPU** (or C): the number of tuples that can be processed per unit time.

**IO** (or I): the number of pages that can be scanned per unit time.

Next, we describe our general estimation model of CPU cost, network cost, and IO cost. The cost of an operation we estimate here is proportional to the real time taken to complete the operation. The response time is estimated by adding up the CPU cost, network cost and IO cost involved in answering a query. Note that not all costs are reflected in response time, since some operations are done offline, and hidden from users. The cost is computed by dividing the complexity of the operation involved by the capacity. We are interested in comparing the cost metrics for three strategies:

- **Push**: At each update, the updategram side delivers updategram to both booster side and view side, the booster side computes booster, and the view side computes the delta view from updategram and booster, and update the view.
- **Hybrid**: deliver updategram, compute booster, compute the delta view, and update the view when the number of updates reach some threshold. **PULL** can be regarded as one extreme of this mechanism, where the numbers of updates accumulated before delivery of updategram and booster is equal to the number of updates occur between two adjacent queries.
- **Adaptive**: Our strategies described in section 4.

### 6.1 CPU Cost

The CPU cost to compute the view from updategram and booster can be estimated as the sum of two costs: the cost to join the updategram and booster, and the cost to merge the resulting delta view with the original view:

\[
\frac{(\text{updategram size} + \text{booster size})}{C} + \frac{(\Delta \text{view size} + \text{view size})}{C}
\]

Since both updategram and boosters are in general, not big and can be held in memory, we can build a hash table for one of them on the join field, and hash join is linear time regarding to updategram size and booster size. Similarly, Δview is usually not big, so we can build a hash table for it. The cost of merge Δview and view is therefore the sum of the cost to maintain the hash table, and the cost to scan the view.

Three strategies have different computation cost because the joins and merges are done at the different rate.

- **Push**: Join and merge are done at each update.
- **Hybrid**: Join and merge are done at slower rate than push, while each operation is more expensive, because the tuples involved are larger.
- **Adaptive**: Join rate is determined by booster push rate, and merge rate is determined adaptively according to CPU and IO resources.

### 6.2 IO Cost

The IO cost to update the view from Δview can be estimated as the cost to read the original view and the cost to write the updated view:

\[
\left( \frac{\text{view size}}{T} + \frac{\Delta \text{view size}}{I} \right)
\]

where T is the number of tuples in a page.

To scan the original view, the number of pages read is \(\frac{\text{view size}}{T}\). Assume that view has some index built on it, then the to merge Δview, we only need to write those tuples that are updated. These updated tuples are likely to be inserted into or deleted from different pages, therefore the number pages written is close to the number of updated tuples. View update rate of different strategies have been discussed above.

### 6.3 Burst Network Cost

The cost to transmit updategram packet and booster to the view side can be estimated as:

\[
\frac{(\text{updategram size} + \text{booster size})}{B}
\]

To compute Δview, updategram and booster need to be delivered via network.

- **PUSH**: updategram and booster are sent at each update.
- **Hybrid**: updategram and booster are sent at a fixed rate that is slower than push.
- **Adaptive**: updategrams are sent at the update rate, boosters are sent at the rate that is adaptive to the network bandwidth.

### 6.4 Total Network Cost

Given the fixed number of updated tuples, the total traffic involved transmitting updategrams and boosters to the view can be estimated as:

\[
\sum_{\text{updategrams}} \frac{(\text{updategram size} + \text{booster size})}{B}
\]

### 6.5 Response Time

The time to answer the user query after its arrival can be estimated as

- **Push**: \(\frac{\text{view size}}{T} / I\)
- **Hybrid**: \(\frac{(\text{updategram size} + \text{booster size})}{C} + \frac{(\text{updategram size} + \text{booster size})}{B}\)
\[ \frac{(ugram.size + booster.size)}{C} + \frac{(\Delta view.size + view.size)}{C} + \frac{(view.size/T + \Delta view.size)}{I} \]

- **Adaptive:**
  If network resource is limited:
  \[ \frac{(ugram_{last}.size + booster_{last}.size)}{C} + \frac{booster_{last}.size}{B} + \frac{(ugram_{last}.size + booster_{last}.size)}{C} + \frac{(\Delta view.size + view.size)}{C} + \frac{(view.size/T)}{I} \]
  Otherwise:
  \[ \frac{(\Delta view.size + view.size)}{C} + \frac{(view.size/T)}{I} \]

For push, the response time simply the time to scan the view table to answer the query, because the view is always up to date.

For hybrid, the response time involves the time for booster generation (receiving the updategram and compute the booster), booster transmission, delta view computation, and the I/O cost to update view while answering the query. All these operations are on the critical path of answering the query.

For Adaptive scheme, the updategram transmission and booster generation is eager, so it is always out of critical path of answering the query. Booster transmission is eager if network resource is bountiful. In this case, it is not counted in response time. Otherwise, the transmission of last booster needs to be counted. Delta view merge is done lazily, therefore it may be on the critical path. In our strategy, writing back the result can be delayed after the query has been answered, so it is not counted in response time.

6 Experimental Results

In this section, we present our experimental results that explore the performance of our strategy on update propagation. We compare our solution with other schemes described earlier in both the situations of enough resources and limited resources.

6.1 Methodology

Since the benchmark for databases in the context of peer-to-peer systems does not exist yet, we do our experiments over a 0.1 scale TPC-H benchmark []. This benchmark has 8 base relations and 22 queries that could be used as the materialized views on peers. Since our project is focused on the update propagation for SPJ views on 2 base relations, we use the simplified version without aggregation from [] and we select 7 queries from them. For all the experiments, our results are the average of the results collected for the 7 queries. For each query, we create an updategram collection consisting 1000 single record insertion, deletion or update.

In each experiment, we run the queries on real database (IBM DB2) and collect statistic information, which serve as the input to our cost models. We compute the performance metrics and resource cost based on our cost model. Since our cost model approximates the real cost and the input is real data, we believe our experiments are reasonable.

6.2 Comparison Under Enough Resources

We first compare our solution with other candidates when the resources are enough. Here the “enough” means all the solutions can refresh the materialized views up to date without overloading the system. Our candidates include push, pull and two hybrid strategies push-10 and push-100. Here push-10 and push-100 mean publish the updategrams when the size of updategrams is above 10 and 100 respectively. Because the size of our updategram collection is 1000, pull is equivalent to push-1000.

We compare all these solutions on two aspects. The first aspect is the performance. The performance metrics used here is response time. The second aspect is resource usage, which includes metrics of CPU cost, I/O cost, burst network cost and total network cost.

The performance comparison is shown in Figure 2. The vertical axis indicates the response time relative to the response time of our solution. We see that the response time of push is the smallest, followed by our strategy, push-10 and push-100 with only slightly difference from push. And then there is a significant jump on pull.
The reason why there is no significant difference among push, push-10, push-100 and our strategy is that the response time includes the time to scan the view in order to answer the query, which is the same for all the strategies. This time is the dominant for the above four strategies, thus, their response time is very close. For pull, the time spent on computing and transmitting the booster is comparable with the time scanning the view, so the response time increases significantly.

Figure 3 shows the results of resource usages. For CPU and I/O costs, with the increment of updategram size, the costs drop dramatically. The reason is that the costs of CPU and I/O are dominated by the times of scanning the view, because we suppose boosters could be produced using index but there is no index on the view side. The total number of view scans is inversely proportional to the size of updategrams. Our strategy consumes roughly the same amount of CPU and I/O resources with push-100. That’s because we set the buffer size equal to 100, and view is updated for every 100 updategrams in our experiments.

Burst network cost is determined by the booster size and updategram size. Figure 2 show that the burst network costs increases a little slower than linear with the updategram size, because of the overlap exists in booster. The burst network cost of our solution is as low as push. That’s because we eagerly send the boosters and thus reduce the burst cost significantly.

The difference of total network cost is not dramatic. With the updategram size increases, this cost will decrease gradually because more overlap could be explored when booster size is bigger.

### 6.3 Comparison Under Limited Resources

In a peer-to-peer system, the availability of resources is not guaranteed. Since all the peers are connected through wide-area network and peers are not dedicated to a single task, the available amount of network bandwidth, CPU and I/O may fluctuate significantly. With the decrease of available resources, the time needed to keep views up to data becomes longer. When resource decrease to some extend, the response time may become unacceptable and in that case it is not feasible to maintain the view up to data. We call the system is overloaded at that time.

In this experiment, we evaluate the throughput and response time (i.e. latency) of different strategies when resources are scarce. Here throughput is defined as the maximum update rate that the system can tolerate and make views up-to-date. When the update rate becomes greater than the throughput, the system become overloaded. If we suppose the data query rate is a constant, intuitively, with the shrink of the available resources, the throughput, i.e. the allowable update rate, will decrease. Since we want to compare different strategies, we do not care the absolute values of throughput. Instead, we are more interested in the trend of throughput change under different strategies.

Each resource, network, I/O or CPU, could become bottleneck and affect the throughput and response time. Here we only show the experimental results when network bandwidth becomes the bottleneck. We could get similar result when I/O or CPU resources are limited.

Since we cannot get the throughput value directly from the experiments, we derive it from the actual booster sizes. Let $B_0$ be the bandwidth capacity in normal situation, i.e. the number of tuples that can be delivered by network per unit time. Let $\varepsilon$ be the proportion of available network resource ($0 \leq \varepsilon \leq 1$), and let $U(i)$ be the size of an updategram including $i$ tuples and $B(i)$ be the size of the booster corresponding to the $i$ tuples.

For PUSH, the possible update rate without overloading the system should satisfy the following condition:

$$(U(1) + B(1)) \times n \leq \varepsilon B_0$$

then we get:

$$\text{throughput} = n_{\text{max}} = \frac{\varepsilon B_0}{(U(1) + B(1))}$$

For PULL, the condition is:

$$(U(n) + B(n)) \leq \varepsilon B_0$$
From our experiments, we can get a sequence of \( U(n) \) and \( B(n) \) when \( n \) changes. Thus we can also compute the throughput for different \( \varepsilon \).

For our strategy, when network resource is scarce, it tends to buffer more boosters to save network bandwidth. Thus it performs similarly with PULL. But since the buffer size is limited, it cannot buffer all the boosters when \( n \) becomes bigger and bigger.

Figure 5 shows the change of throughput with the change of available resource in the three strategies. Because the PULL can explore the overlap of boosters and thus reduce the network cost, in our experiments the throughput is roughly as double as that of PUSH. For our strategy, when the throughput is not big, our strategy has the same throughput with PULL, because the buffer is not overfilled yet. When the throughput becomes bigger, our strategy degrades slightly relative to PULL because of the limit of buffer size. In our experiments, the maximum updategram size is 1000 tuples and our buffer size is only 100 tuples. Our experiments show that we can almost fully achieve the advantage of booster overlap by using a not big buffer.

![Figure 5: Throughput vs. Available Resource](image)

Besides throughput, another concern is response time. Figure 6 shows the response time of three strategies when network bandwidth is not enough. As we can see from the figure, when available bandwidth ratio is high, the response time of our strategy is roughly as low as push and much lower than pull. As the network bandwidth decreases, the performance of all the three strategies will degrade. Because for PULL the bottleneck is the network, the response time for PULL increases more obviously. In our experiment, when the ratio decreases to around 0.5, the push strategy overloads the system. In the figure, the response time jumps to a huge value. (The exact value of the huge number has no real meaning; it’s just to indicate overloading in our experiments). Because our strategy can detect the network congestion and buffer more boosters, it can avoid the sharp overloading. Instead, it degrades gracefully and works more like PULL. Then when the ratio continues to decrease to around 0.3, both PULL and our strategy overload.

![Figure 6: Response Time vs. Available Resource](image)

6.4 Under Different Workloads

All the experimental results above are the average results of 9 queries and do not distinguish different workloads. Here we investigate the effect of different workloads. We divide the queries into three groups. The first group is \( U(n) > B(n) \), i.e. the size of updategram is significantly greater than the size of boosters. This is common when the updated table holds a foreign key referencing the booster table. The second group is \( U(n) = B(n) \), i.e. the size of updategram is roughly the same with the size of boosters. And the third group is \( U(n) < B(n) \).

Figure 7 shows the response time of the three strategies in each group. As we can see, our conclusion got from the experiment in 6.2 holds in each group. With the size of boosters becomes bigger
and bigger, the respond time for PULL become longer and longer.

Figure 7: Response Time Under Different Workloads

Figure 8 and 9 show the burst network cost and I/O cost in each group, respectively. Basically, there is no significant difference in each group.

Figure 8: Burst Network Cost Under Different Workloads

Figure 9: I/O Cost Under Different Workloads

7 Extensions and Future Work

So far we have considered the case when only two base tables attend in the join operation and only one of them are updated. The main idea of this policy and the algorithm can be generalized to manage more complex situations. In the remainder of this section we describe several extensions of this basic problem, and identify our future work.

7.1 Updates to Both Relations

If both base tables are updated simultaneously, they send updategrams of theirs own and boosters produced for the updategrams from the other side. According to Propagation Rules in [5], the computation on the view side can be defined as:

\[ V' = (R \oplus \mu_R) \bowtie (S \oplus \mu_S) \]

\[ = (R \bowtie S) \oplus (\mu_R \bowtie S) \oplus (R \bowtie \mu_S) \oplus (\mu_R \bowtie \mu_S) \]

\[ = V \oplus (\mu_R \bowtie \beta_1(\mu_R, S)) \]

\[ \oplus (\beta_1(\mu_R, S) \bowtie \mu_S) \oplus (\mu_R \bowtie \mu_S) \]

Thus we need to hold two sets of separated updategram pools for the two base tables. They are maintained and joined with boosters in the same way as described in Section 4.3. In addition, the updategram items in the two pools are joined with each other using double pipelined hash join [6]. The results of all the operations mentioned above are put into the delta view pools.

To guarantee the semantics of the updates, we need to synchronize update items from both base tables. A key point here is how to add timestamps that can help the view side tell the exact order of those updategrams from both sides.

7.2 Join Views on Multiple Relations

Another extension is that when the view joins several base tables together but only one of them are updated. There are two ways to address this case. One is called cascading method. It is based on the following formulas:

\[ V' = (R \oplus \mu_R) \bowtie S_1 \bowtie S_2 \bowtie \ldots \bowtie S_n \]

\[ = (R \bowtie S_1 \bowtie S_2 \bowtie \ldots \bowtie S_n) \oplus (\mu_R \bowtie S_1 \bowtie \mu_R \bowtie S_2 \bowtie \ldots \bowtie S_n \]

\[ = V \oplus (\mu_R \bowtie \beta_{V_1}(\mu_R, S_1)) \bowtie \beta_{V_2}(\mu_R, \beta_{V_1}(\mu_R, S_1)) \bowtie \ldots \bowtie \beta_{V_n}(\mu_R, S_1) \bowtie S_n \]

where

\[ V_1 = R \bowtie S_1 \]

\[ V_2 = R \bowtie S_1 \bowtie S_2 \]

\[ V_{n-1} = R \bowtie S_1 \bowtie S_2 \bowtie \ldots \bowtie S_{n-1} \]

\[ V_n = R \bowtie S_1 \bowtie S_2 \bowtie \ldots \bowtie S_n \]
To implement this formula, we need to maintain $n$ sets of delta view pools. When an item is inserted into the $i$th pool, it is sent to the $(i+1)$th booster side. After the side produces the corresponding booster and send it back to the view side, it is joined with the booster items and the result is put into the $(i+1)$th pool. The drawback of the cascading method is that the boosters need to be generated sequentially. Also, the order in which different base tables generate the boosters are fixed and this may lead to bad execution plan.

In another approach, called parallel method, the boosters are generated in parallel. The basis of this method is:

$$\beta_{V_2}(\mu_{V_1} \Join S_2) \subseteq \beta_{V_2}(\mu_{S_0}, S_2)$$

$$\beta_{V_{n-1}}(\mu_{V_{n-1}} \Join S_n) \subseteq \beta_{V_n}(\mu_{S_0}, S_n)$$

So the base table sides for $S_1$, $S_2$, … $S_n$ can produce boosters simultaneously. Then the view side joins the them together with the updategram. However, the boosters generated in this approach can be much larger than those in cascading method. Also, the streamed join algorithm needs to be modified dramatically to handle with multiple boosters.

Cascading method and parallel method are two extremes for this case. Future work can be done to explore the search space between the two extremes. Moreover, how to pursue an algorithm for the situation blending both cases mentioned in above two sections is an open issue.

### 7.3 Updates Propagation in Uncentralized System

In our system, we assume that all the participants of a view know the definition of the views and the locations of other participants. This can be done by registering and maintaining view definition tables. In this way, there is no need for a centralized controller. However, the specific mechanism to do needs to be developed.

### 8 Conclusion

A significant concern in building a peer data management system (PDMS) is the efficient management of database updates while maintaining consistency of dependent materialized views located on a remote host. The eager join, lazy merge policy we proposed in this paper try to improve the performance of the system while preserving resources when they are limited. In particular, we have demonstrated that eager join introduces little extra overhead compared to pull while improve response time significantly, while lazy merge saves huge amount of computation and disk I/O compared to push. In addition, we also proposed adaptive approaches with grace performance degradation when resources are limited. The experimental results demonstrate the effectiveness of our system.

### References


