Cost-aware view materialization for highly distributed datasets

Justin Cappos
University of Arizona
Tucson, Arizona
justin@cs.arizona.edu

Austin Donnelly, Richard Mortier,
Dushyanth Narayanan, Antony Rowstron
Microsoft Research
Cambridge, United Kingdom
{dnarayan,austind,mort,antr}@microsoft.com

ABSTRACT

Querying large datasets distributed over thousands of endsystems is a challenge for existing distributed querying infrastructures. High data availability requires either replicating or centralizing the dataset but both require infeasibly high network bandwidth. In-situ querying provides low bandwidth overheads but requires users to tolerate low data availability.

This paper advocates partial data replication, increasing the availability of a subset of the data through centralization and/or in-network (peer-to-peer) replication. This is analogous to materializing views in centralized databases, but where materialized views in centralized databases trade view update overheads for query overheads, in the distributed case they trade bandwidth usage for availability.

Given an example workload, state-of-the-art tools for centralized databases are able to determine a set of materialized views that will improve performance. Key to this is the ability to estimate view maintenance costs with different hypothetical materialized views. This paper describes estimation of view maintenance costs in a highly distributed database. We present metrics that capture the cost of different materializations, and show that we can estimate these metrics accurately, efficiently, and scalably on a real distributed dataset.

1. INTRODUCTION

Querying highly distributed datasets suffers from data unavailability due to endsystem and network outages [5, 16]. To achieve high availability the entire dataset would ideally be *centralized* by migrating it to a single site where it can be located within a single database and maintained with high-availability using standard techniques. Unfortunately, for large-scale distributed datasets this can result in prohibitively large incoming data rates. For example, a network of 300,000 endsystems each generating a modest 10 Kbps results in an incoming data stream of 3 Gbps.

At the other extreme is *in-situ querying* [16] where the dataset is not replicated but queries are executed through a scalable distributed protocol. The bandwidth overhead of in-situ querying is significantly lower than that of data centralization. However, in-situ querying suffers from low data availability: when endsystems are offline, the data stored on those endsystems cannot be queried.

On the spectrum between centralization and in-situ querying is *in-network (peer-to-peer) replication.* This provides high data availability through data replication to a few selected peers but requires more maintenance bandwidth per endsystem than centralization. Its benefit over centralization is that bandwidth usage is distributed over the entire network rather than concentrated into a single location.

With large datasets, it is infeasible to use either centralization or in-network replication for the entire dataset. However, it may be possible to centralize or in-network replicate selected *views* on the data. The infrastructure would then support three materialization options for each view: centralized, in-network replicated, or not materialized at all. Depending on bandwidth constraints, some views would be centralized, some in-network replicated, and the remaining data left on the endsystems where they were generated.

Correspondingly there would be three modes of query execution: centralized, in-network, and in-situ. Queries against the centralized views are executed against the centralized database, with high data availability and low response times. Queries against in-network replicated views require a query distribution protocol and hence higher response time, but data availability is still high. For queries against data that are neither centralized nor in-network replicated, the system uses in-situ querying, which also requires a query distribution protocol and further may have low data availability.

A major challenge is to select the views to be centralized or in-network replicated. We must ensure that the bandwidth costs of view maintenance are acceptable: a badly chosen configuration can easily overwhelm the network. Thus the key challenge, and the focus of this paper, is answering *what-if* questions about the cost of hypothetical materialized view configurations. Given this ability, a DBA or automated tuning tool can explore multiple configurations and choose the best one.

Answering *what-if* questions about hypothetical materialized views is well understood for centralized databases [7]. Automatic tuning tools such as AutoAdmin [1, 2] leverage this to determine appropriate view materializations for a given dataset and workload. They trade the increased cost of updating materialized views for the reduced cost of querying against those views. However, the analogous problem in the distributed case presents a fundamentally different set of challenges. Materialization in this case consists of migrating tuples across a network rather than writing them to disk. Thus the cost tradeoff is between the network bandwidth required for view maintenance and the increased data availability for queries against views.

Estimating distributed view materialization costs accurately and efficiently is non-trivial. The costs depend on the local data update rates on each endsystem, which means the cost information is distributed across the entire network. Further, such large networks exhibit endsystem churn, i.e., the availability of endsystems in the network changes dynamically. Hence the challenge is to provide cost estimation mechanisms that are scalable and fault-tolerant yet accurate.

Since the update rate of many datasets varies over time, the bandwidth required by our system to maintain a materialized view will also vary over time. For many applications, such as endsystem-based network monitoring, the update rate will have hourly, daily or weekly patterns. It is important to capture these, especially when aggregating costs across multiple endsystems with correlated patterns. Our cost estimation mechanisms explicitly capture these patterns by representing costs as timeseries rather than single time-invariant quantities.

1.1 Contributions

The main contribution of this paper is to describe how to support cost-aware materialized views for a highly distributed dataset. We present and evaluate the following:

- Two complementary schemes for materializing views on a highly distributed dataset,
- Cost metrics for comparing different hypothetical configurations of materialized views, and
- Efficient, scalable and fault-tolerant methods for estimating these cost metrics.

1.2 Road map

Section 2 describes our system model and view materialization mechanism, and gives a high-level overview of our cost estimation techniques. Section 3 provides an overview of Seaweed, upon which we build our system. Section 4 describes our motivating application, endsystem-based network monitoring. Section 5 details the design and implementation of our cost metrics and cost estimation mechanisms. Section 6 presents experimental evaluation of our materialization mechanisms and cost estimation techniques. Section 7 discusses integration of our work with automated tuning techniques. Finally, Sections 8 and 9 discuss related work and conclude.

2. OVERVIEW

Our target domain consists of large-scale datasets distributed over many networked endsystems. In our model, the data consists of a small number of tables, each of which is *horizontally partitioned* across all the endsystems. Thus each endsystem locally generates and is responsible for managing a small subset of each table. We assume that unavailable endsystems do not generate new tuples.

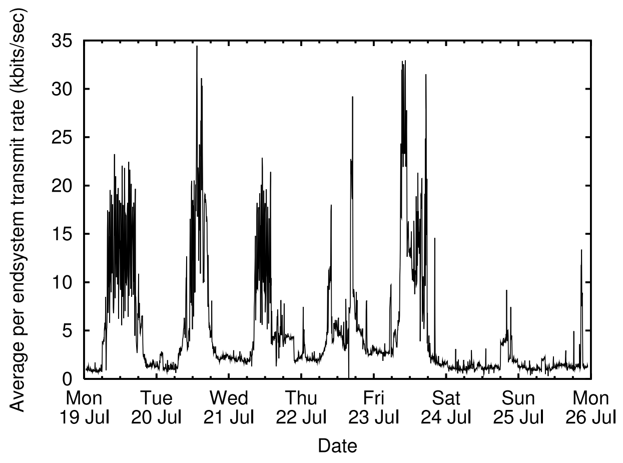


Figure 1: Average per-endsystem bandwidth required if all tuples were centralized.

	Availability	*Maintenance bandwidth*	
		Max/endsystem	*Aggregate*
Centralized	Very high	Very high	Medium
In-network	High	Medium	Very high
In-situ	Low	None	None

Table 1: View materialization tradeoffs

An example application is an endsystem-based network management system in an enterprise network containing thousands to several hundred thousand endsystems [9]. In such an environment it is usually not feasible simply to centralize all the data. Figure 1 shows, over a week, the average bandwidth per endsystem to migrate *all* tuples to a centralized DBMS. The per-endsystem average and peak bandwidth are 5.0 Kbps and 34.5 Kbps respectively. A typical large enterprise network might have $\sim 300,000$ endsystems. At this size the average inbound bandwidth to the centralized DBMS would be 1.5 Gbps and the peak bandwidth would exceed 10 Gbps, which is infeasibly high.

Since complete data migration is infeasible, we provide mechanisms to centralize or in-network replicate one or more views on the data. For any hypothetical configuration, a DBA or auto-tuning tool needs to know the cost entailed. This *what-if* functionality [7] is supported through *cost estimation queries*.

The remainder of this section describes our system model and our approach to cost estimation. Section 5 describes in detail the implementation of view materialization and the cost estimation algorithms.

2.1 System model

Although tables are horizontally partitioned across a large number of endsystems, this partitioning is abstracted away by the system. Users issue queries against entire tables. The query might then be executed against tuples stored in a centralized database, or distributed across endsystems in the network. In the latter case, each endsystem computes query results for its locally stored tuples, and the results are aggregated back in the network.

Queries are performed against views that are either maintained in a centralized DBMS, in-network or in-situ. Table 1 shows the tradeoffs between these three options. Centralized and in-network querying requires that the view tuples are replicated. In the centralized case the tuples are replicated on a centralized DBMS, while in-network requires that tuples are replicated on other endsystems. We assume that a view maintained on a centralized DBMS has high availability. The centralized DBMS is hosted in a well-managed data center with redundant hardware, and is therefore highly available and resilient to failures.

If a view is replicated in-network, the tuples are replicated on arbitrary endsystems with availability orders of magnitude lower than that of the centralized DBMS; in a large enterprise network 20% of endsystems are unavailable on average [5]. Hence, multiple replicas of each tuple must be maintained; then with high probability at least one of the replicas should be available at all times, and thus the view has high availability. Table 1 summarizes this, showing that creating a centralized view provides higher data availability in general than in-network, but the trade-off is in the maintenance bandwidth. With in-network view materialization, each tuple has to be replicated multiple times which requires a high per-endsystem outbound bandwidth and high aggregate bandwidth across the network when compared to centralized view materialization. However, the inbound bandwidth requirement for centralized view materialization at the centralized DBMS is orders of magnitude higher than that required at any individual endsystem to materialize the view in-network.

The final option is to query the view in-situ, which requires no replication of the view tuples, as advocated in Seaweed [16]. Tuples are simply stored on the endsystems that generated them, and queries are performed at each of these endsystems. However, at any point in time only a subset of a view's tuples will be accessible as tuples stored on endsystems that are unavailable will obviously not be accessible. Seaweed addressed this issue by estimating for each query the number of tuples that are currently unavailable and when these tuples are likely to become available. Therefore, unlike centralized and in-network materialized views, querying a view in-situ provides much lower data availability. In-situ querying persists the query, and distributes it to endsystems as they come online, thus increasing the data availability over time. However, it may take many hours to reach a sufficiently high level of coverage, say 99% of all tuples. As Table 1 show the benefit of in-situ querying is that no bandwidth is required to maintain the view, as the view tuples are stored on the endsystem which generates them.

In general, we assume that query results are typically small and can be efficiently aggregated in-network. Thus the network costs are dominated by the cost of view maintenance. Centralized views impose a high load on the incoming link to the centralized DBMS; in-network replicated views cause a high aggregate load distributed over the network; and in-situ querying with no materialized views incurs no view maintenance costs, but provides much lower data availability.

In our query model we assume single-table select-project-aggregate queries against horizontally partitioned tables. We only support joins against centralized views: distributed joins are very difficult to perform efficiently over hundreds of thousands of networked endsystems, and we chose not to provide functionality that could overwhelm the network if used carelessly.

2.2 View materialization

We assume there is a highly available centralized DBMS with a limit on its inbound bandwidth. The bandwidth usage of centralization must be kept under this limit: even a well-connected DBMS could be overwhelmed by the combined data rates of a large number of endsystems.

Each endsystem e in the network has a limit L_e on the outbound bandwidth that it can use for centralization or in-network replication. In a typical enterprise network scenario such as we discuss in Section 4, view maintenance will run as a low-priority, background service. Thus L_e will be constrained to be a small fraction of the endsystem's network link, dependent on local connectivity and policy.

A view configuration is a pair $< S_C, S_P >$. S_C is the set of views to be centralized. Endsystems transmit tuples in these views to the centralized DBMS. Since the limiting factor when centralizing data is the inbound bandwidth at the centralized DBMS, individual endsystems may not be using all their available capacity L_e to maintain S_C. Thus each endsystem can use any remaining bandwidth to *in-network replicate* an additional set of views S_P to a small number of other endsystems.

Aggregation views can be materialized but require different cost estimation metrics and mechanisms since they use an in-network aggregation tree. Typically, they require very little bandwidth to maintain and hence network costs are dominated by select-project views. We do not present cost estimation mechanisms for aggregation views in this paper but we are confident that our techniques can be extended to estimate their costs.

2.3 Partial materialization

A view is materialized by broadcasting a view materialization command to all endsystems, containing the view query and the materialization mode (centralized or in-network replicated). Endsystems then begin to transmit tuples matching these views either to the centralized DBMS or to other selected endsystems in the network.

In a distributed system some endsystems will be offline when the materialization command is inserted. We cannot centralize tuples held by those endsystems until they come online which may take hours or days. However, we would still like to allow queries on the centralized DBMS against some clearly defined subset of the view tuples. We achieve this by introducing the notion of a *partially materialized view*.

A partially materialized view has a "start time" T_s, typically the time that the materialization command was inserted into the system. Tuples generated by available endsystems after time T_s will be proactively pushed to the centralized DBMS. Thus queries on the view that relate only to tuples after time T_s can be executed immediately on the centralized DBMS without the need for distributed querying. If the query references tuples generated before T_s, then in-network or in-situ querying must be used.

Most queries in our motivating application — endsystem-based network management — require high availability of the most recent data; thus when materializing a view the natural assumption would be that the tuples produced after the view creation are required to be highly available. This motivates our decision to prioritize these tuples over historical tuples for transmission over the network.

Note that since the tuples originate from a large num-

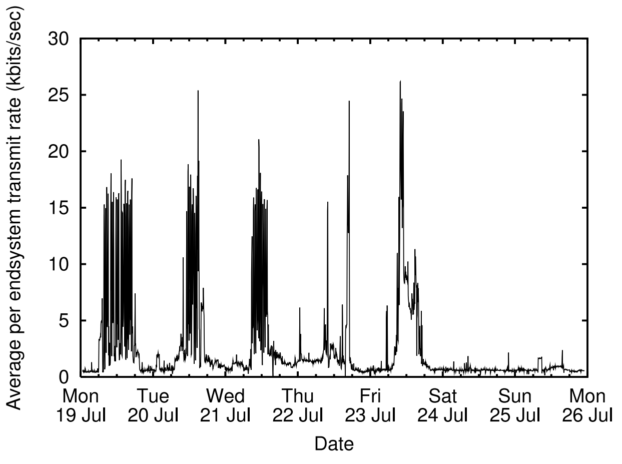


Figure 2: Average per-endsystem bandwidth required if only tuples relating to SMB were centralized.

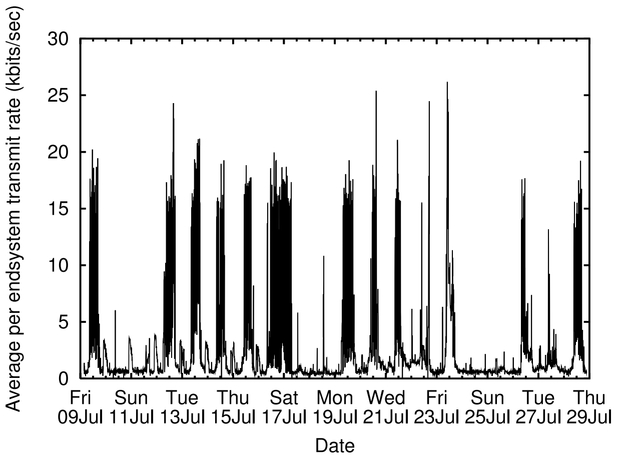


Figure 3: The data in Figure 2 shown over an extended 3-week period. Daily (day/night) and weekly (weekday/weekend) periods are clearly visible.

ber of sources in a wide-area network, there may still be some network delay before they reach the centralized DBMS, and there is no ordering across tuples arriving from different endsystems. Thus the partially materialized view corresponds to a (partial) *dilated reachable snapshot* [13] of the view tuples in the distributed dataset. Note that in our model unavailable endsystems do not generate new tuples, or equivalently, that any such tuples are only considered part of the dataset when the endsystem next becomes available.

Endsystems also propagate older view tuples (i.e., those generated before time T_s) lazily to the centralized DBMS. When all such historical tuples have been received, the view is marked as *fully materialized.* At this point, all queries on the view, irrespective of time, can be redirected to the centralized DBMS. For applications such as network management, users are often more interested in recent data. Hence there is great value in immediately materializing views on the recent data, and lazily centralizing the historical tuples, rather than waiting for all tuples to be centralized before allowing use of the view.

For in-network replication, it is difficult to determine the point at which a view becomes fully materialized, due to the dynamic and distributed nature of the replication. Hence we regard all in-network replicated views as partially materialized. In-network replicated data generated after T_s will be fully materialized and queries on this data will see high data availability. Queries on older data will see low data availability initially but over time, as older tuples are lazily replicated, their data availability will improve.

2.4 View cost estimation

Distributed view cost estimation is very different from that in a centralized database. The main cost of distributed view materialization is network bandwidth and the main benefit is higher data availability. This differs from the traditional cost metrics used in a centralized DBMS. Further, bandwidth and data availability depend on the dynamic availability of endsystems as well as the data update rates on individual endsystems. Hence any estimation mechanism has to collect this distributed information scalably and in a fault-tolerant manner, and also to generate accurate estimates despite endsystem unavailability.

Both bandwidth usage and data availability vary over time and between endsystems. For example, in the context of a network management tool, a view that captures tuples relating to remote filesystem activity will have higher update rates during working hours (Figure 2). This time-dependence need not be the same for all endsystems: large enterprise networks contain endsystems in different time zones. Thus, although we are interested in statistics such as the average and 95th percentile of bandwidth usage at the centralized DBMS, we cannot simply derive them from the corresponding statistics on individual endsystems; we need cost metrics that preserve the time correlations.

Thus we use timeseries representations of expected bandwidth usage and endsystem availability. It is important that the timeseries representation captures any cyclic patterns observed in the dataset. In our network management dataset we observe hourly, daily, and weekly patterns as shown in Figure 3, and so we use a timeseries that spans a week. In our implementation each of these timeseries is represented by a histogram with 2016 bins, each bin representing a 5 min interval, the typical timescale of interest for network operators [14]. This gives us a good compromise between fine granularity and estimation overhead: unnecessarily fine granularity will increase the size of the timeseries and hence the network overhead of cost estimation queries.

A cost estimation query specifies a proposed hypothetical configuration $< S_C, S_P >$ and is efficiently broadcast through the system. A per-endsystem cost estimate in timeseries form is computed for every endsystem in the system. Each online endsystem locally generates its own cost metric; for each offline endsystem a single online endsystem assumes responsibility for estimating the cost on its behalf. These per-endsystem timeseries are then efficiently aggregated in the network to give the overall cost estimate. To ensure that aggregation is efficient it is essential that the cost metrics be *additive*, i.e., that the global cost estimate can be computed by applying a commutative, associative operator to the endsystems' local cost metrics. With timeseries we can maintain this additive property without losing the time correlations across endsystems.

The local cost estimate timeseries at each endsystem is based on past history. In other words, to answer the question “what will be the future cost of centralizing or in-network replicating these views”, we answer the question “what would have been the cost if I had started centralizing or in-network replicating these views in the past”. In our evaluation we use 2 weeks of history and maintain it as a 1-week timeseries, e.g., a timeseries of update rates for each base table on each endsystem.

The response to a cost estimation query for a configuration $< S_C, S_P >$ is a 4-tuple of timeseries:

$$< B_C(t), B_P(t), A_C(t), A_P(t) >$$

$B_C(t)$ and $B_P(t)$ are the aggregate bandwidth cost estimates of maintaining the centralized and in-network views respectively. The aggregate centralized bandwidth $B_C(t)$ is the same as the inbound bandwidth to the centralized DBMS. Any feasible configuration must ensure that this bandwidth $B_C(t)$ is always within the capacity of the centralized DBMS. Since the centralized DBMS sinks data from a very large number of endsystems, we expect $B_C(t)$ to determine the feasibility of S_C, independent of the per-endsystem transmit caps.

$A_C(t)$ and $A_P(t)$ are the availability of the data in the centralized and in-network views respectively. We define (data) availability as the fraction of the tuples in a view that are available for querying at any given time relative to the total number of tuples in that view. Endsystems preferentially centralize/replicate tuples from views with low availability. The effect is to equalize availability across views, hence we report a single mean availability for all views in S_C and similarly for S_P. In other words, we aggregate over views at each local endsystem and over endsystems in the network but we do not aggregate over time, maintaining availability as a timeseries. The aggregate data availability across endsystems is the the mean availability weighted by each endsystem’s tuple count.

Since the bottleneck for centralization will be the centralized DBMS’s incoming link, and we avoid configurations that overwhelm this link, most tuples will be centralized soon (i.e., to within network latency) after they are produced on the local endsystem. Hence $A_C(t)$ will be very close to 1, only dropping occasionally when the centralized DBMS’s incoming link is temporarily overloaded causing a backlog. Since we expect such backlogging to be rare, the main metric of interest for centralized views is the bandwidth $B_C(t)$.

On the other hand for in-network replication the aggregate bandwidth usage $B_P(t)$ is of secondary importance, since each endsystem locally ensures that its bandwidth constraints are met. However, the resulting data availability will vary depending on the amount of churn in the system and the degree of replication that each endsystem e can achieve given its bandwidth cap L_e. Thus the main metric of interest for the in-network replicated views is $A_P(t)$.

One of the main challenges in cost estimation is in generating estimates for endsystems that are currently unavailable, but will become available in the future and begin to centralize or replicate data. Our general approach is to have online endsystems estimate costs on behalf of offline endsystems using the underlying protocols to ensure that exactly one estimate is submitted for every endsystem. It is key that these estimation algorithms use no global knowledge, but rely only on small amounts of meta-data which are replicated periodically by each endsystem to a small number of other endsystems. The details of the estimation techniques used depend on whether the view is centralized or in-network replicated, and are described in Section 5.

3. BASE SYSTEM

We build our view materialization and cost estimation mechanisms on top of Seaweed [16], an infrastructure for in-situ querying of highly distributed data. Seaweed does not centralize or replicate data, and thus has low maintenance overheads but also suffers low availability when endsystems are offline. Seaweed is implemented as a decentralized overlay application built on top of the Pastry [17] distributed hash table, and implements a number of overlay-related services through scalable, fault-tolerant distributed protocols. Here we briefly describe how we utilize these underlying Seaweed services; a detailed description of Seaweed can be found in [16].

Seaweed provides support for *in-situ querying*, and we use this unmodified for queries against data that is neither centralized nor in-network replicated. This support includes disseminating each query to every online endsystem; storing each query in the network and ensuring it is delivered to every offline endsystem that subsequently comes online; and building, per-query, a persistent aggregation tree that includes each endsystem’s contribution without duplication, incrementally refining the aggregation result as offline endsystems come online.

Seaweed also supports in-network *metadata replication*: each endsystem can replicate small objects to a small number of replicas. Seaweed implements *replica-set maintenance*, ensuring that all replicas have a consistent view of the set and that objects are correctly re-replicated when there is churn in the replica set. Seaweed also supports *estimation queries* against these replicated objects. These are used to estimate data availability over time. Estimation queries are executed locally by each online endsystem, and exactly one of each offline endsystem’s replicas will respond on its behalf. Since estimation queries do not wait for endsystems to come online, their results are aggregated using a *lightweight aggregation tree* rather than the heavyweight persistent tree used by in-situ querying.

We use Seaweed’s metadata replication and estimation query mechanisms to support view cost estimation queries by extending the replicated metadata to support more complex estimation queries. This lets us estimate view maintenance costs for both online and offline endsystems. To do so, we replicate a small amount of additional metadata about data update rates.

We extend Seaweed’s mechanisms to support in-network view replication. Here the “objects” to be replicated are the local tuple-sets corresponding to each view. These can be much larger than the small objects supported by Seaweed’s replication mechanism. Thus, we rely on Seaweed’s replica-set maintenance but implement our own mechanisms for object replication. In particular, we implement mechanisms for partial replication of tuple-sets, and prioritization across different tuple-sets.

We also extend the Seaweed query mechanisms to implement queries against in-network replicated views. Seaweed guarantees that only one replica of each offline system will respond on its behalf. We further ensure that queries against an in-network replicated view are always answered by the replica holding the most tuples for that view, thus maximizing data availability.

4. MOTIVATING APPLICATION

Our motivating application is endsystem-based monitoring of enterprise networks [9]. Each endsystem locally captures its own network activity into a table, `Packet`. A record is generated in the `Packet` table per transmitted or received packet. Each record contains a timestamp, local and remote IP addresses and ports, application, protocol, direction of the packet (receive or transmit), and the packet size in bytes.

Network administrators perform queries over the dataset. For example, a typical query could be to determine the number of bytes of web traffic:

```
SELECT SUM(Bytes) FROM Packet WHERE LocalPort=80
```

or, the traffic on privileged port numbers (those below 1024), which would capture server traffic:

```
SELECT SUM(Bytes), LocalPort FROM Packet
WHERE LocalPort < 1024 GROUP BY LocalPort
ORDER BY SUM(Bytes).
```

Another example would be to discover the busiest web servers:

```
SELECT TOP (10) LocalIP, SUM(Bytes) FROM Packet
WHERE LocalPort=80 GROUP BY LocalIP
ORDER BY SUM(Bytes)
```

perhaps followed by a query pertaining to a specific web server that has been identified as a "heavy hitter":

```
SELECT TimeStamp, Bytes FROM Packet
WHERE LocalPort=80 And LocalIP=IPLookup('myserver')
```

As these are common queries, it would be useful if a view could be centralized that would allow all these queries to be answered with high availability and low response time. The single view:

```
SELECT Timestamp, Bytes, LocalPort, LocalIP
FROM Packet WHERE LocalPort < 1024
```

would achieve this for the above queries. However, users often generate new query variants dynamically, e.g., to diagnose an unexpected performance issue. Therefore, only a sample of the queries to be performed is known ahead of time. Thus, it may be better to centralize a more general view even though it will incur a higher bandwidth maintenance cost:

```
SELECT * FROM Packet WHERE LocalPort < 1024
```

or even:

```
SELECT * FROM Packet
```

Unfortunately the administrator has no way of knowing the cost of centralizing these views. The last proposed view is typically too large to centralize. Depending on available bandwidth, data update rate, and the bandwidth usage of other materialized views, even the first proposed view might have been too large to materialize centrally. If the administrator knew these costs, she might choose to centralize the views:

```
SELECT Timestamp, Bytes, LocalPort, LocalIP
FROM Packet WHERE LocalPort < 1024
```

and:

```
SELECT * FROM Packet WHERE LocalPort = 80
```

and additionally in-network replicate the view:

```
SELECT * FROM Packet WHERE LocalPort < 1024
```

rather than centrally. Thus queries on server traffic that could not be satisfied by the centralized view could still use the in-network replicated view, giving high availability with response times higher than centralized querying but lower than in-situ querying. Any queries not against materialized views would fall back on in-situ querying.

5. DESIGN

Given a configuration $< S_C, S_P >$ each endsystem monitors its locally generated tuples to detect updates to each of the materialized views. The tuples are locally queued for transmission and are then transmitted whenever possible, subject to the local bandwidth constraint L_e. When there is insufficient bandwidth, higher-priority tuples are sent first and lower-priority ones are queued until there is available bandwidth. Centralized views are always prioritized ahead of those for in-network replicated views. Historical tuples, those generated before the view creation time T_s, receive the lowest priority of all. Within a view, tuples are prioritized by timestamp, earlier tuples receiving higher priority subject to the constraint that historical tuples are always lower priority than tuples generated after the view was materialized. Within each view set (S_C or S_P), we prioritize views by availability as described below.

In this section we describe how tuples are prioritized for transmission, and the estimation algorithms that predict the network costs of this transmission and the resulting availability, first for centralized views and then for in-network replicated views.

5.1 Centralized views

Each endsystem e has a set of objects $\{o_i^e\}$. Each o_i^e contains the tuples *owned* by e (i.e., locally generated by it) and contained in some view v_i where $v_i \in S_C$. We define the availability at any time of each o_i^e as the fraction of its tuples that are stored on the central server C:

$$AvailCentral(o_i^e) = \frac{TupleCount(C, o_i^e)}{TupleCount(e, o_i^e)}$$

Each endsystem e always prioritizes the object with the lowest current availability and transmits its tuples to C.

5.1.1 Cost estimation

Given a cost estimation query containing a set of views S_C, each endsystem computes its local cost estimate $< B_C(t), A_C(t) >$ for centralization. This pertains only to tuples generated after time T_s: historical tuples do not have any periodic long-term costs, and they are transmitted in the background until they have all been centralized.

Each endsystem e first computes the update rate time-series $U_v^e(t)$ for each view $v \in S_C$. $U_v^e(t)$ is generated by observing when tuples in the recent past, e.g., the last week, would have been generated for each view v. New endsystems with less than 1 week of history are ignored for the purposes of estimation; typically there are a very small number of such endsystems at any given time.

Locally at each endsystem the update rate timeseries for the data to be centralized is:

$$U^e_C(t) = \sum_{v \in S_C} U^e_v(t)$$

The mean utilization, i.e., the average fraction of available bandwidth used, is:

$$M^e_C = \frac{\sum_t U^e_C(t)}{N_t L_e}$$

where, N_t is the number of timeseries intervals (2016 in the implementation and experiments described in this paper) and L_e is the local transmit bandwidth cap, represented as the maximum amount of data that can be transmitted in a single timeseries interval. If $M^e_C > 1$, then centralization of this set of views is infeasible for this endsystem, since there will be insufficient long-term bandwidth to transmit all the view tuples.

Each endsystem e also imposes its limit L_e on transmissions within each interval. Hence, the actual amount of transmission bandwidth used in each interval $[t-1, t]$ is

$$B^e_C(t) = \min(L_e, Q^e_C(t-1) + U^e_C(t))$$

where $Q^e_C(t-1)$ is the backlog at time $t-1$, i.e., the number of untransmitted tuples from previous intervals.

$$Q^e_C(t) = \begin{cases} 0 & \text{if} \quad t = 0 \\ \sum_{\tau=0}^{t}(U^e_C(\tau) - B^e_C(\tau)) & \text{if} \quad t > 0 \end{cases}$$

Given $Q^e_C(t)$, it is also possible to compute the availability over time:

$$A^e_C(t) = \begin{cases} 1 - \frac{Q^e_C(t)}{\sum_{\tau=1}^{t} U^e_C(\tau)} & \text{if} \quad M^e_C \le 1 \\ 0 & \text{if} \quad M^e_C > 1 \end{cases}$$

In other words, if the average bandwidth is sufficient to replicate all tuples, then availability will occasionally fall below 1 as tuples get temporarily backlogged. However, if the average bandwidth is not sufficient, then the backlog will grow indefinitely, and availability will stay consistently below 1, i.e., some of e's tuples will never become available at the centralized DBMS. The long-term "availability" of e's tuples will then be the ratio of available bandwidth to required bandwidth. However, since our aim is only to centralize views which can be maintained with a long-term availability of 1, we conservatively estimate e's availability to be 0.

5.1.2 *Estimating for offline endsystems*

In addition to online endsystems, we must also generate estimates for offline endsystems. In order to achieve this we replicate a small amount of *metadata* per endsystem to a replica set. When an endsystem e is offline a single member M of its replica set is selected as its manager and assumes responsibility for generating e's cost estimate.

The metadata does not directly contain the per-view update rate as it is infeasible to replicate the update rates for all possible views. The metadata contains:

- An availability profile $P_e(t)$ which contains e's recent availability, i.e., the times at which it was online and offline, and
- An update rate timeseries $U^e_T(t)$ for each base table in the dataset, and
- The transmission bandwidth limit L_e.

The manager uses the metadata to generate cost estimates for e. To do this, it computes e's update rate for any view v on the base table T as:

$$U^e_v(t) = f^e_v(t) U^e_T(t)$$

where $f^e_v(t)$ is the amount of data reduction due to selection and projection. This is estimated from M's local data:

$$f^e_v(t) = f^M_v(t) = \frac{U^M_v(t)}{U^M_T(t)}$$

Using this, M is able to estimate $B^e_C(t)$ and $A^e_C(t)$ for the offline endsystem e. This is an approximation based on the assumption that the selectivity of the view v will be similar for M's data and e's data.

5.2 In-network replicated views

For in-network replication, endsystems use the replica set maintained by Seaweed. Each endsystem e is provided with a replica set containing n live endsystems:

$$R = \{r_1, r_2, \ldots, r_n\}$$

The replicas are ranked such that when e goes offline, Seaweed will choose r_1 to be its manager, or if r_1 is offline at that point, it will chose r_2, etc.

For each object o^e_i where $v_i \in S_P$, we define its availability on each replica r_j similarly to the centralized case:

$$Avail(r_j, o^e_i) = \frac{TupleCount(r_j, o^e_i)}{TupleCount(e, o^e_i)}$$

We define the overall availability of an object o^e_i as a vector of per-replica availabilities sorted in decreasing order:

$$AvailAll(o^e_i) = \{a_1, a_2, \ldots, a_n\} =$$

$$Decreasing(\{Avail(r_1, o^e_i), Avail(r_2, o^e_i), \ldots, Avail(r_n, o^e_i)\})$$

This availability vector defines an ordering on the objects:

$$o^e_i < o^e_j \iff \exists m, (a^i_1 = a^j_1) \wedge \ldots \wedge (a^i_m = a^k_m) \wedge (a^i_{m+1} < a^j_{m+1})$$

The object prioritized for replication is always the one with the least vector in this ordering. In other words, we prioritize the object with the least availability on its best replica by first comparing the best replicas (highest availability) across all objects, breaking ties by comparing the second-best replicas, and so forth.

The endsystem e then chooses a target replica for this high-priority object o^e_i. This is chosen to be the highest-ranked replica in the Seaweed ranking that does not already have an availability of 1. It then transmits missing tuples to the target replica in timestamp order.

5.2.1 *Replicating for offline endsystems*

View materialization works as described above for online endsystems. Offline endsystems are assumed not to generate tuples, but for each offline endsystem e, it is necessary to maintain the availability of its in-network replicated objects $\{o^e_i\}$. This is achieved by re-replicating the tuples belonging to $\{o^e_i\}$ as replica set membership changes due to endsystem failure or recovery.

When e is online, it manages all in-network replication of its tuples. When offline the underlying Seaweed infrastructure assigns to it a *manager* M. The manager takes responsibility for ensuring availability of the tuples, re-replicating

v1	`SELECT * FROM Packet`
v2	`SELECT * FROM Packet WHERE LocalPort < 1024`
v3	`SELECT * FROM Packet WHERE App='SMB'`

Table 2: Example views used in experiments.

the objects o_i^e as necessary. Over time the manager of an endsystem changes as other endsystems become available or the manager fails.

The manager uses the same algorithm as described above to prioritize views for replication. It will be responsible for replicating objects owned by it as well as those of other endsystems for which it is the manager. The availability criterion for prioritization is applied across all these objects irrespective of their owner. However availability is now computed with respect to the number of tuples of each object that are stored on M, which may be less than that on the original owner e. The target replica is chosen from M's ranked replica set. Typically M will have more tuples than any other replica and will push tuples to the target replica; in rare cases where the target replica has more tuples, M pulls missing tuples from it.

5.2.2 *Cost estimation*

Unlike centralized views, the bandwidth and availability timeseries for in-network replication cannot be computed easily through a formula, since they depend on the interaction between tuple update rates and endsystem availability. To compute these timeseries, each endsystem e simulates the effect of in-network replication by stepping forward through each interval in a week-long timeseries.

The input to the simulation is the view update rate and availability timeseries for the endsystem as well as its replicas: these are computed based on local history. Another input is the bandwidth timeseries representing the bandwidth left over after centralization. The simulator then steps through the timeseries, at each step computing the number of tuples replicated from each object. In other words, we estimate our future behavior based on simulating our hypothetical past behavior, i.e., by answering the question "what if I had materialized these views 1 week ago?"

The estimation does not capture the impact on bandwidth and availability of the endsystem e re-replicating objects for which it is the manager but not the owner. Based on local information alone, e cannot predict how many objects it would be required to manage, since that depends on changes in the replica sets of other endsystems, and this is not included in the metadata replicated to e. In Section 6 we show that, despite ignoring re-replication costs, our estimator has good accuracy for enterprise networks where churn is moderate and re-replication is relatively infrequent.

Similarly to the centralized case, estimates for each offline endsystem are generated by its manager, and in this case the view update rates are estimated based on the manager's data reduction factor $f_v^M(t)$.

6. EVALUATION

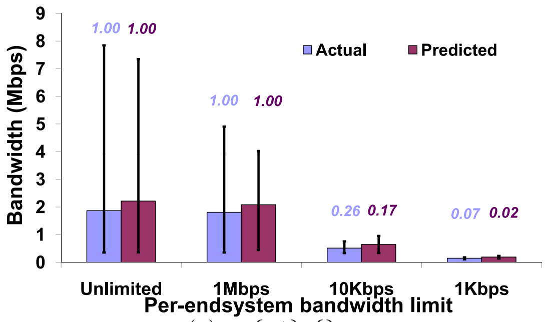
In order to evaluate the proposed system we implemented a discrete event simulator, which allowed evaluation of a large number of configurations and design choices. The simulator models each endsystem independently and incorporates events for addition of new tuples to base tables and views, transmission of tuples for centralization or in-network replication, meta-data replication, and endsystem churn (i.e., endsystems failing and recovering). The simulator models per-link bandwidth for the links connecting endsystems to the network, allowing us to examine the effect of varying the outbound link capacity L_e.

To drive the experiments we used a number of real-world traces: an endsystem availability trace from an enterprise network and an application dataset representative of an endsystem based network management application. To generate the representative application dataset we collected a packet trace of all routed network traffic in our building for the period 30 Aug 2005—20 Sep 2005. The raw packet trace was then processed to generate per-endsystem `Packet` tables for the 456 endsystems in our building, containing a total of 13 billion tuples. In the `Packet` table each tuple corresponds to a packet and contains timestamp, local and remote IP addresses and ports, application name, protocol, packet direction, and packet size in bytes. The 456 endsystems represented in the trace are significantly fewer than those targeted by our design. Scaling the dataset could introduce biases (e.g., by replicating each endsystem's data multiple times), so we present results for the trace network size.

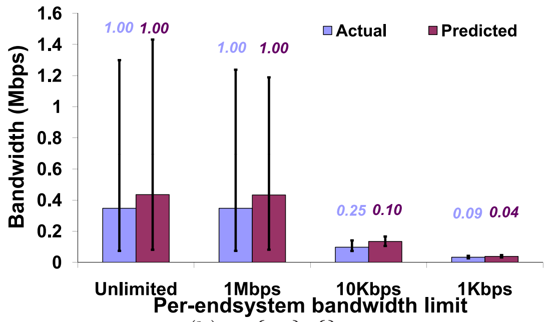
The bandwidth overhead of performing estimation and insitu querying will increase with the number of endsystems, but previous work has demonstrated that these can be implemented scalably [16]. The total bandwidth required to centralize or in-network materialize a set of views will increase linearly with the number of endsystems. In the evaluation we focus on the accuracy of cost estimation. We believe that the accuracy of cost estimation achieved on our dataset is representative of that achievable with a much larger number of endsystems.

In order to evaluate the effectiveness of estimating the materialization costs for offline endsystems we require per-endsystem availability profiles. However, the packet trace does not capture endsystem availability and inferring this from network packet traffic is unreliable. Therefore, the per-endsystem availability was generated using an availability trace gathered over approximately 4 weeks in July/August 1999 in the Microsoft corporate network [5]. This trace actively probed each of 51,663 endsystems on the corporate network hourly. For each experiment the 456 endsystems were each mapped to a single endsystem randomly selected from the set of 51,663 endsystems in the availability trace. When mapping the availability trace to the packet trace we performed the mapping such that we maintained both time of day and day of week. If the packet trace indicated that a tuple was generated during a period when the endsystem was unavailable it was discarded, since in reality endsystems would not generate or log network traffic when unavailable. We observed approximately 20% of the tuples being discarded in the experiments, and this is an unavoidable artifact of using independent packet and availability traces.

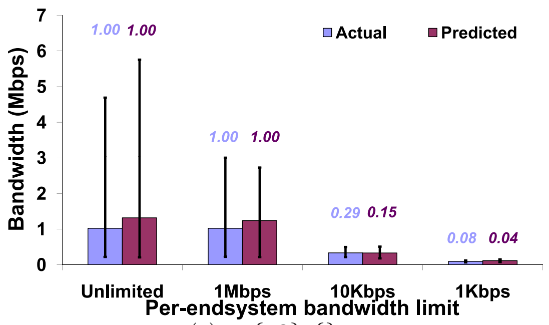
Each experiment covers three weeks of simulated time. During the first two weeks tuples are appended to local tables but not centralized or in-network replicated. Also, each endsystem accumulates statistics on base table update rates, as well as availability patterns of itself and its replica set members. At the end of the two weeks, a cost estimation query for a specific view configuration $< S_C, S_P >$ is



(a) $< \{v1\}, \{\} >$



(b) $< \{v2\}, \{\} >$



(c) $< \{v3\}, \{\} >$

Figure 4: Bandwidth annotated with availability, predicted and measured for each centralized view materialization configuration. Histograms give the mean bandwidth, error bars give the 5th and 95th percentiles.

inserted, the local endsystem cost estimates are computed and aggregated to generate the global estimates.

We then inject the materialization command for the configuration $< S_C, S_P >$. During the third week of simulation, the configuration is maintained through centralization or in-network replication of tuples. The simulator also gathers statistics of bandwidth usage and view availability. Thus at the end we are able to compare the cost estimates at the beginning of the third week with the measured costs during the third week.

In our experiments we used a number of different views, which varied update rates, update patterns and so forth. Due to space constraints we limit the results shown to cover the three views specified in Table 2.

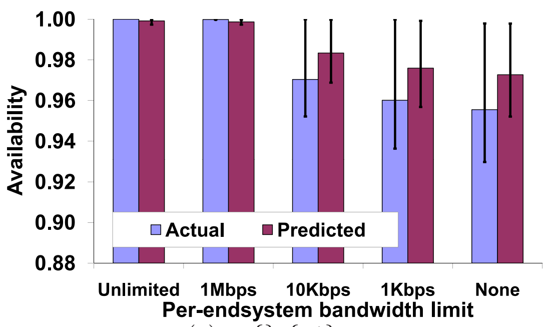
6.1 Centralized view materialization

The first set of experimental results presented evaluates the effectiveness of centralized view materialization. In particular we evaluate the feasibility of materializing different views, the accuracy of bandwidth cost estimation, and the sensitivity of bandwidth prediction to the per-endsystem bandwidth caps.

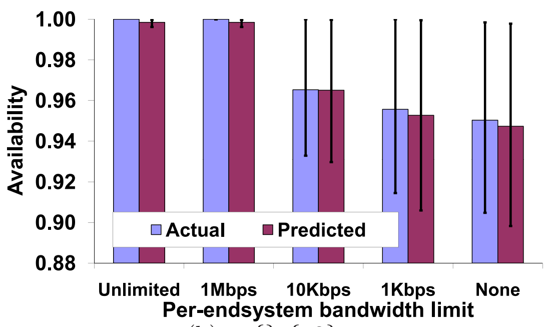
We ran three simulations using the specific view configurations: $< \{v1\}, \{\} >$, $< \{v2\}, \{\} >$ and $< \{v3\}, \{\} >$, i.e., configurations where exactly one view is centralized and no views are in-network replicated. In each run we recorded the tuple $< B_C(t), A_C(t) >$ returned by cost estimation, which captures the estimated inbound bandwidth on the centralized DBMS and the estimated availability for a week, using 5-min buckets. The predicted mean incoming bandwidth on the centralized DBMS, as well as the 90% confidence interval, i.e., the 5th and 95th percentiles of our 5-min buckets are generated using $B_C(t)$. We then materialized the configuration and measured, over the following 7 days, the actual mean bandwidth as well as the 5th and 95th percentiles. While the mean provides an indication of the long-term bandwidth requirements, the 95th percentile measures the short-term "burst" load inflicted on the inbound network link to the centralized DBMS. In order to examine the sensitivity of the predictor to the per-endsystem bandwidth caps we ran each experiment with 4 different values of $L_e = \infty$, 1 Mbps, 10 Kbps and 1 Kbps.

Figure 4 shows the results for each of the three view configurations. Within each configuration, the actual and predicted mean bandwidths and confidence intervals (shown as error bars) are shown for different values of per-endsystem bandwidth L_e. Each graph also shows the estimated and actual median availability above the appropriate column. When L_e is low the data availability is correctly predicted as being too low to be useful. When L_e is sufficiently large the data availability is correctly predicted as being high. The inbound bandwidth to the centralized DBMS is predicted with good accuracy across all values of L_e, despite a factor of 5 variation in bandwidth usage across the different views. This is important as the inbound bandwidth determines the feasibility of materializing the view.

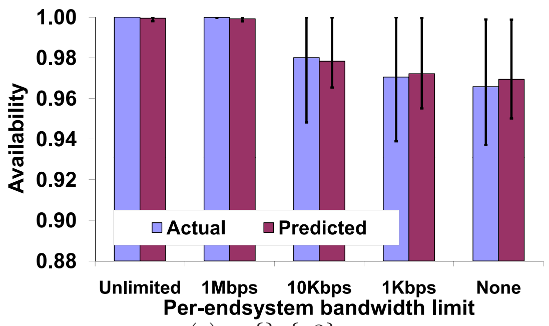
Figure 4(a) shows that $v1$, which centralizes *all* tuples, has a mean bandwidth cost of 1.8 Mbps and a 95th percentile of 7.8 Mbps, even for the small sample of 456 endsystems. A similar per-endsystem data rate for an enterprise network of 300,000 endsystems would result in a mean incoming bandwidth of 1.2 Gbps and a 95th percentile of 5.1 Gbps, which is clearly infeasible for a single system to handle. The value of cost estimation is to avoid such situations before they occur.



(a) $< \{\}, \{v1\} >$



(b) $< \{\}, \{v2\} >$



(c) $< \{\}, \{v3\} >$

Figure 5: Availability prediction for in-network view materialization. Histograms give the mean availability, error bars give the 5th and 95th percentiles.

6.2 In-network view materialization

The next set of experiments evaluates prediction performance for in-network view materialization. We ran three simulations using the specific view configurations: $< \{\}, \{v1\} >$, $< \{\}, \{v2\} >$ and $< \{\}, \{v3\} >$, i.e., configurations with exactly one in-network replicated view and no centralized views. In each run we recorded $A_P(t)$ returned by cost estimation, capturing the expected availability. We ran each experiment with 5 different values of $L_e = \infty$, 1 Mbps, 10 Kbps, 1 Kbps and *None*. In the *None* case there is no replication of view tuples, representing the data availability for in-situ querying. We always attempt to maintain 8 replicas of the data. However, the actual number of replicas maintained depends on the per-endsystem bandwidth limit.

Figure 5 shows the results for each of the three view configurations. Within each configuration, the actual and predicted availability and the confidence intervals (shown as error bars) are shown for different values of L_e. As expected, as L_e drops the actual availability decreases and this is accurately predicted. The results also show the benefit of view materialization in general, as the availability for in-situ querying (*none*) is lower than that for all other configurations.

To conclude, the results for both the centralized and in-network view materialization experiments demonstrate the availability gain achieved by materializing the view. The results also show that the proposed cost estimation techniques accurately predict the required bandwidth and achieved availability of view materialization.

7. AUTOMATED TUNING

There has been considerable work on automated database tuning tools such as AutoAdmin [2, 7], the DB2 Design Advisor [21] and the Oracle SQL Access Advisor as part of their Automatic SQL Tuning feature [10], which rely on "what-if" queries. We believe that the view cost estimation proposed in this paper represents an important step to allowing such tuning tools for highly distributed databases.

To automate the selection of materialized views, we propose to combine our cost estimation techniques with previously published methods for auto-tuning in centralized databases. As a specific example, we consider AutoAdmin [1, 2, 6], a tool that aids DBAs by automatically proposing materialized views to improve the performance of a given workload on an existing DB configuration. AutoAdmin combines three techniques:

- Generation of candidate materialized views using syntactic analysis and pairwise view merging,
- Heuristic searching of the configuration space defined by the candidate views, and
- *What-if* cost estimation of hypothetical configurations.

We believe candidate view generation can be used without major modifications, other than ensuring that we log the query workload. The heuristic search technique will need to be extended. Currently, AutoAdmin uses a $Greedy(m, k)$ technique [6] to first exhaustively search the space of all configurations with up to m views, and then greedily adds up to $k - m$ additional views, where m is a small number, e.g. $m = 2$. The number of configurations examined by $Greedy(m, k)$ on a set of N candidate views is $O(N^m + Nk)$. We need to adapt the technique to make the search efficient in terms of the network overhead, which is directly proportional to the number of estimation queries executed.

For the search, a key observation is that the cost metrics for centralized views are additive. Thus we can estimate the cost of each candidate view independently, and hence derive the cost estimate of any configuration of centralized views. However, the cost metric for in-network replication is not additive, since each endsystem could have its own bandwidth constraints, and hence cost is measured in terms of availability. Hence we cannot consider views independently, and therefore each in-network configuration needs to be run on an entire configuration. The searching also requires estimation of query-time cost, which in this case simply becomes a function of availability and view materialization location (in-network or centralized).

8. RELATED WORK

Prior work on infrastructures supporting highly distributed databases has focused on two extremes. Systems like PIER [13] replicate all the tuples in the network whereas systems like Seaweed [16] do no replication and rely on in-situ querying. The approach proposed in this paper is between these two extremes. We assume that we can, if necessary, perform in-situ querying and use replication of a subset of the tuples as an optimization to improve availability. This is fundamentally different from proposals to redistribute (rather than replicate) tuples for load-balancing and other purposes [12]. The replicated subset of the data is specified as a view, and to achieve scalability we need to provide cost estimates of view maintenance to ensure that we do not cause network overload.

Mid-tier transparent database caching [15] reduces load on back-end databases by caching query results at inexpensive intermediate nodes. Cached results are dynamically maintained, and the differences in latency between querying the cache and the back-end database are exposed to the optimizer. We use replication to increase availability rather than reduce load, and we are interested in much larger system scale.

Segev et al. [19, 18] propose "distributed views" with batched, differential updates where a single centralized DBMS pushes entire views to remote sites for efficiency and load balancing. This is very different from our scenario, where the tuples are initially distributed over a very large network. Further, in the case of in-network replication, a single view can be horizontally partitioned across the entire network.

The centralized views we provide have some similarities with streaming query systems such as [3, 4, 8, 11, 20]. Tuples are routed from endsystems to the central DBMS for processing. However, there is a key difference: we store the tuples for later querying, i.e., for use as a view. In general, streaming query systems route tuples through a network of online processing operators. Results may be stored for later processing but the original tuple is discarded.

9. CONCLUSION

In this paper we have proposed and evaluated mechanisms for cost-aware view materialization for highly distributed datasets. Materialized views are stored either on a centralized database or in-network, where view tuples are replicated over peers. Materializing views increases data availability and reduces query latency, using network bandwidth to replicate tuples.

We have described and evaluated cost estimators for both centralized and in-network view materialization, allowing a DBMS administrator or automated tool to understand the tradeoffs between bandwidth usage and data availability.

Given these two view materialization mechanisms, cost estimators allow an administrator to understand the tradeoffs involved in view materialization. Furthermore, they are a critical component in enabling auto-tuning tools for querying infrastructures designed for large-scale distributed datasets.

10. REFERENCES

[1] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database tuning advisor for Microsoft SQL Server 2005. In *VLDB*, Toronto, Canada, Aug. 2004.

[2] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes for SQL databases. In *VLDB*, pages 496–505, Cairo, Egypt, Aug. 2000.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, pages 261–272, Dallas, TX, May 2000.

[4] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD*, pages 13–24, Baltimore, MD, June 2005.

[5] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *SIGMETRICS*, pages 34–43, Santa Clara, CA, June 2000.

[6] S. Chaudhuri and V. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *VLDB*, pages 146–155, Athens, Greece, Aug. 1997.

[7] S. Chaudhuri and V. Narasayya. Autoadmin "what-if" index analysis utility. In *SIGMOD*, Seattle, WA, USA, 1998.

[8] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *SIGMOD*, pages 379–390, Dallas, TX, May 2000.

[9] E. Cooke, R. Mortier, A. Donnelly, P. Barham, and R. Isaacs. Reclaiming network-wide visibility using ubiquitous end system monitors. In *USENIX*, Boston, MA, June 2006.

[10] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zaït, and M. Ziauddin. Automatic SQL tuning in Oracle 10g. In *VLDB*, pages 1098–1109, 2004.

[11] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, pages 948–959, Toronto, CN, Aug. 2004.

[12] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB*, pages 444–455, Toronto, Canada, 2004.

[13] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB*, pages 321–332, Berlin, Germany, Sept. 2003.

[14] K. Keys, D. Moore, and C. Estan. A robust system for accurate real-time summaries of Internet traffic. In *SIGMETRICS*, pages 85–96, 2005.

[15] P.-Å. Larson, J. Goldstein, H. Guo, and J. Zhou. MTCache: Mid-tier database caching for SQL Server. *IEEE Data Eng. Bull.*, 27(2):35–40, 2004.

[16] D. Narayanan, A. Donnelly, R. Mortier, and A. Rowstron. Delay aware querying with Seaweed. In *VLDB*, Seoul, Korea, Sept. 2006.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, Nov. 2001.

[18] A. Segev and W. Fang. Currency-based updates to distributed materialized views. In *ICDE*, pages 512–520, 1990.

[19] A. Segev and J. Park. Updating distributed materialized views. *IEEE Trans. Knowl. Data Eng.*, 1(2):173–184, 1989.

[20] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed Eddies. In *VLDB*, pages 333–344, Berlin, Germany, Sept. 2003.

[21] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004.