Offering a Precision-Performance Tradeoff for Aggregation Queries over Replicated Data*

Chris Olston, Jennifer Widom
Stanford University
{olston, widom}@db.stanford.edu

Abstract

Strict consistency of replicated data is infeasible or not required by many distributed applications, so current systems often permit *stale replication*, in which cached copies of data values are allowed to become out of date. Queries over cached data return an answer quickly, but the stale answer may be unboundedly imprecise. Alternatively, queries over remote master data return a precise answer, but with potentially poor performance. To bridge the gap between these two extremes, we propose a new class of replication systems called TRAPP (*Tradeoff in Replication Precision and Performance*). TRAPP systems give each user fine-grained control over the tradeoff between precision and performance: Caches store ranges that are guaranteed to bound the current data values, instead of storing stale exact values. Users supply a quantitative *precision constraint* along with each query. To answer a query, TRAPP systems automatically select a combination of locally cached bounds and exact master data stored remotely to deliver a *bounded answer* consisting of a range that is no wider than the specified precision constraint, that is guaranteed to contain the precise answer, and that is computed as quickly as possible. This paper defines the architecture of TRAPP replication systems and covers some mechanics of caching data ranges. It then focuses on queries with aggregation, presenting optimization algorithms for answering queries with precision constraints, and reporting on performance experiments that demonstrate the fine-grained control of the precision-performance tradeoff offered by TRAPP systems.

1 Introduction

Many environments that replicate information at multiple sites permit *stale replication*, rather than enforcing exact consistency over multiple copies of data. Exact (transactional) consistency is infeasible from a performance perspective in many large systems, for a variety of reasons as outlined in [GHOS96], and for many distributed applications exact consistency simply is not a requirement.

The World-Wide Web is a very general example of a stale replication system, where master copies of pages are maintained on Web servers and stale copies are cached by Web browsers. In the Web architecture, reading the stale cached data kept by a browser has significantly better performance than retrieving the master copy from the Web server (accomplished by pressing the browser's "refresh" button), but the cached copy may be arbitrarily out of date. Another example of a stale replication system is a data warehouse, where we can view the data objects at operational databases as master copies, and data at the warehouse (or at multiple "data marts") as stale cached copies. Querying the cached data in a warehouse is typically much faster than querying the master copies at the operational sites.

*This work was supported by the National Science Foundation under grant IIS-9811947, by NASA Ames under grant NCC2-5278, and by a National Science Foundation graduate research fellowship.

1.1 Running Example

As a scenario for motivation and examples throughout the paper, we will consider a simple replication system used for monitoring a wide-area network linking thousands of computers. We assume that each node (computer) in the network tracks the average latency, bandwidth, and traffic level for each incoming network link from another node. Administrators at monitoring stations analyze the status of the network by collecting data periodically from the network nodes. For each link $N_i \rightarrow N_j$ in the network, each monitoring station will cache the latest latency, bandwidth, and traffic level figures obtained from node N_j. Administrators want to ask queries such as:

Q1 What is the bottleneck (minimum bandwidth link) along a path $N_1 \rightarrow N_2 \rightarrow \cdots \rightarrow N_k$?

Q2 What is the total latency along a path $N_1 \rightarrow N_2 \rightarrow \cdots \rightarrow N_k$?

Q3 What is the average traffic level in the network?

Q4 What is the minimum traffic level for fast links (*i.e.*, links with high bandwidth and low latency)?

Q5 How many links have high latency?

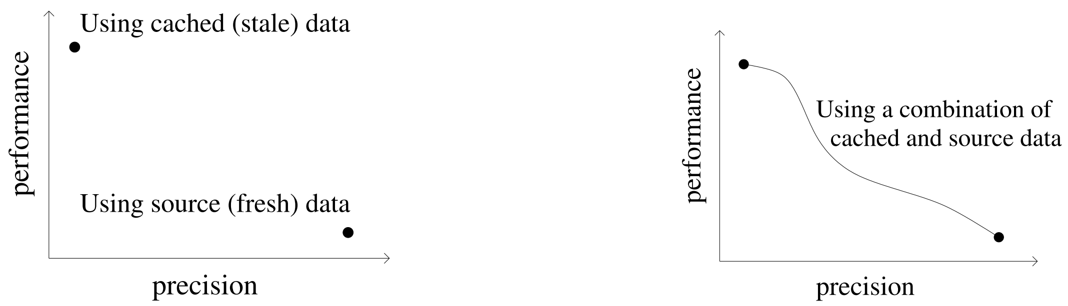
Q6 What is the average latency for links with high traffic?

While administrators would like to obtain current and precise answers to these kinds of queries, collecting new data values from each relevant node every time a query is posed would take too long and might adversely affect the system. Requiring that all nodes constantly send their updated values to the monitors is also expensive and generally unnecessary. This paper develops a new approach to replication and query processing that allows the user to control the tradeoff between precise answers and high performance. In our example, the latency, bandwidth, and traffic level figures at each monitor are cached as *ranges*, rather than exact values, and nodes send updates only when an exact value moves outside of a cached range. Queries such as *Q1–Q6* above can be executed over the cached ranges and themselves return a range that is guaranteed to contain the current exact answer. When an administrator poses a query, he can provide a *precision constraint* indicating how wide a range is tolerable in the answer.

For example, suppose the administrator wishes to sample the peak latency periodically in some critical area, in order to decide how much money should be invested in upgrading the network. To make this decision, the administrator does not need to know the precise peak latency at each query, but may wish to obtain an answer to within 5 milliseconds of precision. Our system automatically combines cached ranges with precise values retrieved from the nodes in order to answer queries within the specified precision as quickly as possible.

1.2 Precision-Performance Tradeoff

In general, stale replication systems potentially offer the user two modes of querying. In the first mode, which we call the *precise mode*, queries are sent to the sources to get a precise (up-to-date) answer but with potentially poor performance. Alternatively, in what we call the *imprecise mode*, queries are executed over cached data to get an imprecise (possibly stale) answer very quickly. In imprecise mode, usually no guarantees are given as to exactly how imprecise the answer is, so the user is left to guess the degree of



(a) In current replication systems

(b) In TRAPP systems

Figure 1: Precision-performance tradeoff.

imprecision based on knowledge of data stability and/or how recently caches were updated. Figure 1(a) illustrates the precision-performance tradeoff between these two extreme query modes.

The discrepancy between the extreme points in Figure 1(a) leads to a dilemma: answers obtained in imprecise mode without any precision guarantees may be unacceptable, but the only way to obtain a guarantee is to use precise mode, which can place heavy load on the system and lead to unacceptable delays. Many applications actually require a level of precision somewhere between the extreme points. In our running example (Section 1.1), an administrator posing a query with a *quantitative precision constraint* like "within 5 milliseconds" should be able to find a middle ground between sacrificing precision and sacrificing performance.

To address this overall problem, we propose a new kind of replication system, which we call TRAPP (*Tradeoff in Replication Precision and Performance*). TRAPP supports a continuous, monotonically decreasing tradeoff between precision and performance, as characterized in Figure 1(b). Each query can be accompanied by a custom precision constraint, and the system answers the query by combining cached and source data so as to optimize performance while guaranteeing to meet the precision constraint. The extreme points of our system correspond to the precise and imprecise query modes defined above.

1.3 Overview of Approach

In addition to introducing the overall TRAPP architecture, in this paper we focus on a specific TRAPP replication system called TRAPP/AG, for queries with aggregation over numeric (real) data. The conventional precise answer to a query with an outermost aggregation operator is a single real value. In TRAPP/AG, we define a *bounded imprecise answer* (hereafter called *bounded answer*) to be a pair of real values L_A and H_A that define a range $[L_A, H_A]$ in which the precise answer is guaranteed to lie. Precision is quantified as the width of the range $(H_A - L_A)$, with 0 corresponding to exact precision and ∞ representing unbounded imprecision. A precision constraint is a user-specified constant $R \geq 0$ denoting the maximum acceptable range width, *i.e.*, $0 \leq H_A - L_A \leq R$.

To be able to give guaranteed bounds $[L_A, H_A]$ as query answers, TRAPP/AG requires cooperation between data sources and caches. Specifically, let us suppose that when a source refreshes a cache's value for a data object O, along with the current exact value for O the source sends a range $[L, H]$ called the *bound* of O. (We actually cover a more general case where the bound is a function of time.) The source guarantees

	link		*latency*		*bandwidth*		*traffic*		*refresh*	*weights*		
	from	*to*	*cached*	*precise*	*cached*	*precise*	*cached*	*precise*	*cost*	W	W'	W''
1	N_1	N_2	[2, 4]	3	[60, 70]	61	[95, 105]	98	3	2	10	29.5
2	N_2	N_4	[5, 7]	7	[45, 60]	53	[110, 120]	116	6	2	10	2
3	N_3	N_4	[12, 16]	13	[55, 70]	62	[95, 110]	105	6		15	41.5
4	N_2	N_3	[9, 11]	9	[65, 70]	68	[120, 145]	127	8		25	2
5	N_4	N_5	[8, 11]	11	[40, 55]	50	[90, 110]	95	4	3	20	36.5
6	N_5	N_6	[4, 6]	5	[45, 60]	45	[90, 105]	103	2	2	15	31.5

Figure 2: Sample data for network monitoring example.

that the actual value for O will stay in this bound, or if the value does exceed the bound then the source will immediately send a new refresh. Thus, the cache stores the bound $[L, H]$ for each data object O instead of an exact value, and the cache can be assured that the current master value of O is within the bound. When the cache answers a query, it can use the bound values it stores to compute an answer, also expressed in terms of a bound.

The small table in Figure 2 shows sample data cached at a network monitoring station (recall Section 1.1), along with the current precise values at the network nodes. The *weights* may be ignored for now. Each row in Figure 2 corresponds to a network link between the *link from* node and the *link to* node. Recall that precise master values for *latency*, *bandwidth*, and *traffic* for incoming links are measured and stored at the *link to* node. In addition, for each link, the monitoring station stores a bounded value for *latency*, *bandwidth*, and *traffic*. The cache can use these bounded values to compute bounded answers to queries.

Suppose a bounded answer to a query with aggregation is computed from cached values, but the answer does not satisfy the user's precision constraint, *i.e.*, the answer bound is too wide. In this case, some data must be refreshed from sources to improve precision. We assume that there is a known quantitative *cost* associated with refreshing data objects from their sources, and this cost may vary for each data item (*e.g.*, in our example it might be based on the node distance or network path latency). We show sample refresh costs for our example in Figure 2. Our system uses optimization algorithms that attempt to find the best combination of cached bounds and master values to use in answering a query, in order to minimize the cost of refreshing while still guaranteeing the precision constraint. In this way, TRAPP/AG offers a continuous precision-performance tradeoff: Relaxing the precision constraint of a query enables the system to rely more on cached data, which improves the performance of the query. Conversely, tightening the constraint causes the system to rely more on master data, which degrades performance but yields a more precise answer.

1.4 Contributions

The specific contributions of this paper are the following:

- We define the architecture of TRAPP replication systems, which offer each user fine-grained control over the tradeoff between precision and performance, and propose a method for determining bounds.

- We specify how to compute the five standard relational aggregation functions over bounded data values, considering queries with and without selection predicates, and with joins.
- We present algorithms for finding the minimum-cost set of tuples to refresh in order to answer an aggregation query with a precision constraint, with and without selection predicates. (Joins are discussed but optimal algorithms are not provided.) We analyze the complexity of these algorithms, and in the cases where they are exponential we suggest approximations.
- We have implemented all of our algorithms and we present some initial performance results.

2 Related Work

There is a large body of work dedicated to systems that improve query performance by giving approximate answers. Early work in this area is reported in [Mor80]. Most of these systems use either precomputation (*e.g.*, [PG99]), sampling (*e.g.*, [HH97]), or both (*e.g.*, [GM98]) to give an answer with statistically estimated bounds, without scanning all of the input data. By contrast, TRAPP systems may scan all of the data (some of which may be bounds rather than exact values), to provide guaranteed rather than statistical results.

The previous work perhaps most similar to the TRAPP idea is *Quasi-copies* [ABGMA88] and *Moving Objects Databases* [WXCJ98]. Like TRAPP systems, these two systems are replication schemes in which cached values are permitted to deviate from master values by a bounded amount. However, unlike in TRAPP systems, these systems cannot answer queries by combining cached and master data, and thus there is no way for users to control the precision-performance tradeoff. Instead, the bound for each data object is set independently of any query-based precision constraints. In Quasi-copies, bounds are set statically by a system administrator. In Moving Objects Databases, bounds are set to maximize a single metric that combines precision and performance, eliminating user control of this tradeoff. Furthermore, neither of these systems support aggregation queries.

The *Demarcation Protocol* [BGM92] is a technique for maintaining arithmetic constraints in distributed database systems. TRAPP systems are somewhat related to this work since the bound of a data value forms an arithmetic constraint on that value. However, the Demarcation Protocol is not designed for modifying arithmetic constraints the way TRAPP systems update bounds as needed. Furthermore, the Demarcation Protocol does not deal with queries over bounded data.

Both [JV96] and [RB89] consider aggregation queries with selections. The APPROXIMATE approach [JV96] produces bounded answers when time does not permit the selection predicate to be evaluated on all tuples. However, APPROXIMATE does not deal with queries over bounded data. The work in [RB89] deals with queries over fuzzy sets. While bounded values can be considered as infinite fuzzy sets, this representation is not practical. Furthermore, the approach in [RB89] does not consider fuzzy sets as approximations of exact values available for a cost.

In the *multi-resolution relational data model* [RFS92], data objects undergo various degrees of lossy compression to reduce the size of their representation. By reading the compressed versions of data objects instead of the full versions, the system can quickly produce approximate answers to queries. By contrast, in TRAPP systems performance is improved by reducing the number of data objects read from remote sources, rather than by reducing the size of the data representation. In *Divergence Caching* [HSW94], a bound is placed on the number of updates permitted to the master copy of a data object before the cache must be refreshed, but there are no bounds on data values themselves.

Another body of work that deals with imprecision in information systems is *Information Quality* (*IQ*) research, *e.g.*, [NLF99]. IQ systems quantify the accuracy of data at the granularity of an entire data server. Since no bounds are placed on individual data values, queries have no concrete knowledge about the precision of individual data values from which to form a bounded answer. Therefore, IQ systems cannot give a guaranteed bound on the answer to a particular query.

Finally, data objects whose values are ranges can be considered a special case of constrained values in *Constraint Databases* [KKR90, BK95, BSCE99, Kup93, BL99], or as null variables with local conditions in *Incomplete Information Databases* [AKG87]. However, no work in these areas that we know of considers constrained values as bounded approximations of exact values stored elsewhere. Furthermore, aggregation queries over a set with uncertain membership (*e.g.*, due to selection conditions over bounded values) are not considered.

3 TRAPP System Architecture

The overall architecture of a TRAPP system is illustrated in Figure 3. *Data Sources* maintain the exact value V_i of each data object O_i, while *Data Caches* store bounds $[L_i, H_i]$ that are guaranteed to contain the exact values. Source values may appear in multiple caches (with possibly different bounds), and caches may contain bounded values from multiple sources. A user submits a query to the *Query Processor* at a local data cache, along with a precision constraint. To answer the query while guaranteeing the constraint, the query processor may need to send *query-initiated refresh requests* to the *Refresh Monitor* at one or more sources, which responds with new bounds. The Refresh Monitor at each source also keeps track of the bounds for each of its data objects in each relevant cache. (Note that in the network monitoring application we consider in this paper, each source must only keep track of a small number of bounds. In other applications a source may provide a large number of objects to multiple caches, in which case a scalable trigger system would be of great benefit [HCH$^+$99].) The Refresh Monitor is responsible for detecting whenever the value of a data object exceeds the bound in some cache, and sending a new bound to the cache (a *value-initiated refresh*).

When the cached bound of a data object is refreshed by its source, some cost is incurred. We consider the general case where each object has its own cost to refresh, although in practice it is likely that the cost of refreshing an object depends only on which source it comes from. (It also may be possible to amortize refresh costs for a set of values, as discussed in Section 8.) These costs are used by our algorithms that choose tuples to refresh in order to meet the precision constraint of a query at minimum cost.

The TRAPP architecture as presented in this paper makes some simplifying assumptions. First, although object insertions or deletions do not occur on a regular basis in our example application, insertions and deletions are handled but they must be propagated immediately to all caches. (Section 8.3 discusses how this limitation might be relaxed.) Second, the level of precision offered by our system does not account for elapses of time while sending refresh messages or while processing a single query. We assume that the time to refresh a bound is small enough that the imprecision introduced is insignificant. Furthermore, we

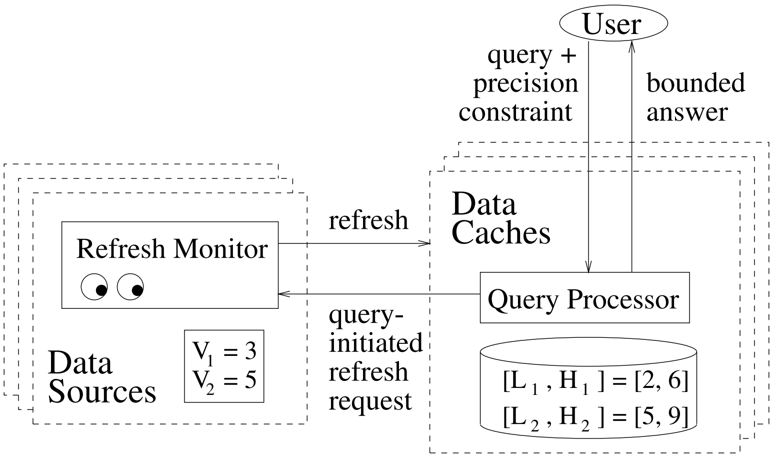


Figure 3: TRAPP system architecture.

assume that value-initiated refreshes do not occur during the time an individual query is being processed. Addressing these issues is a topic for future work as discussed in Section 8.4.

Next, in Section 3.1 we discuss in more detail the mechanics of bounded values and refreshing. Then in Section 3.2 we generalize bound functions to be time-varying functions. In Section 4 we discuss the execution of aggregation queries in the TRAPP/AG system, before presenting our specific optimization algorithms for single-table aggregation queries in Sections 5 and 6. In Section 7 we present some preliminary results for aggregation queries with joins.

3.1 Refreshing Cached Bounds

The master copy of each data object O_i resides at a single source, and for TRAPP/AG we assume it is a single real value, which we denote V_i. Caches store a range of possible values (the *bound*) for each data object, which we denote $[L_i, H_i]$. When a source sends a copy of data object O_i to a cache (a *refresh* event at time $\mathcal{T}_r$), in addition to sending O_i's current precise value, which we denote $V_i(\mathcal{T}_r)$, it sends a bound $[L_i, H_i]$.

As discussed earlier, refreshes occur for one of two reasons. First, if the master value of a data object exceeds its bound stored in some cache (*i.e.*, at current time $\mathcal{T}_c$, $V_i(\mathcal{T}_c) < L_i$ or $V_i(\mathcal{T}_c) > H_i$), then the source is obligated to refresh the cache with the current precise value $V_i(\mathcal{T}_c)$ and a new bound $[L_i, H_i]$—a value-initiated refresh. Second, a query-initiated refresh occurs if a query being executed at a cache requires the current exact value of a data object in order to meet its precision constraint. In this case, the source will send $V_i(\mathcal{T}_c)$ along with a new bound to the cache, and the precise value $V_i(\mathcal{T}_c)$ can be used in the query.

3.2 Bounds as Functions of Time

Section 3.1 presented a simple approach where the bound of each data object O_i is a pair of endpoints $[L_i, H_i]$. A more general and accurate approach is to parameterize the bound by time: $[L_i(\mathcal{T}), H_i(\mathcal{T})]$. In other words, the endpoints of the bound are functions of time $\mathcal{T}$. These functions have the property that

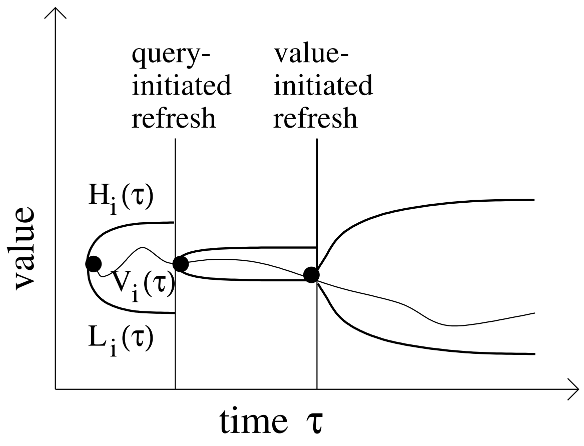


Figure 4: Bound $[L_i(\mathcal{T}), H_i(\mathcal{T})]$ over time, overlaid with precise value $V_i(\mathcal{T})$.

$L_i(\mathcal{T}_r) = H_i(\mathcal{T}_r) = V_i(\mathcal{T}_r)$, where $\mathcal{T}_r$ is the refresh time. That is, the bound at the time of refresh has zero width and both endpoints equal the current value. As time advances past $\mathcal{T}_r$, the endpoints of the bound diverge from $V_i(\mathcal{T}_r)$ such that the bound contains the precise value at all times $\mathcal{T}_c \geq \mathcal{T}_r$: $L_i(\mathcal{T}_c) \leq V_i(\mathcal{T}_c) \leq H_i(\mathcal{T}_c)$. Eventually, when another refresh occurs, the source sends a new pair of bound functions to the cache that replaces the old pair. Figure 4 illustrates the bound $[L_i(\mathcal{T}), H_i(\mathcal{T})]$ of a data object O_i over time, overlaid with its precise value $V_i(\mathcal{T})$.

All of the subsequent algorithms and results in this paper are independent of how bounds are selected and specified. In fact, in the body of the paper we assume that any time-varying bound functions have been evaluated at the current time $\mathcal{T}_c$, and we write $[L_i, H_i]$ to mean $[L_i(\mathcal{T}_c), H_i(\mathcal{T}_c)]$. Also, we write V_i to mean the exact value at the current time: $V_i(\mathcal{T}_c)$. We have done some preliminary work investigating appropriate bound functions, and have deduced that in the absence of additional information about update behavior, appropriate functions are those that expand according to the square-root of elapsed time. That is: $H_i(\mathcal{T}) - L_i(\mathcal{T}) \propto \sqrt{\mathcal{T} - \mathcal{T}_r}$, where $\mathcal{T}_r$ is the time of the most recent refresh. The proportionality parameter, which determines the width of the bound, is chosen at run-time. The interested reader is referred to Appendix A for details.

4 Query Execution for Bounded Answers

Executing a TRAPP/AG query with a precision constraint may involve combining precise data stored on remote sources with bounded data stored in a local cache. In this section we describe in general how bounded aggregation queries are executed, and we present a cost model to be used by our algorithms that choose cached data objects to refresh when answering queries. For the remainder of this paper we assume the relational model, although TRAPP/AG can be implemented with any data model that supports aggregation of numerical values.

For now we consider single-table TRAPP/AG queries of the following form. Joins are addressed in Section 7.

```
SELECT  AGGREGATE(T.a) WITHIN R
FROM    T
WHERE   PREDICATE
```

AGGREGATE is one of the standard relational aggregation functions: COUNT, MIN, MAX, SUM, or AVG. PREDICATE is any predicate involving columns of table T and possibly constants. R is a nonnegative real constant specifying the precision constraint, which requires that the bounded answer $[L_A, H_A]$ to the query satisfies $0 \leq H_A - L_A \leq R$. If R is omitted then $R = \infty$ implicitly.

To compute a bounded answer to a query of this form, TRAPP/AG executes several steps:

1. Compute an initial bounded answer based on the current cached bounds and determine if the precision constraint is met. If not:

2. An algorithm CHOOSE_REFRESH examines the cache's copy of table T and chooses a subset of T's tuples T_R to refresh. The source for each tuple in T_R is asked to refresh the cache's copy of that tuple.

3. Once the refreshes are complete, recompute the bounded answer based on the cache's now partially refreshed copy of T.

Our CHOOSE_REFRESH algorithm ensures that the answer after step 3 is guaranteed to satisfy the precision constraint.

Sections 5 and 6 present details based on each specific aggregation function, considering queries with and without selection predicates. For each type of aggregation query we address the following two problems:

- How to compute a bounded answer based on the current cached bounds. This problem corresponds to steps 1 and 3 above.

- How to choose the set of tuples to refresh. This problem corresponds to step 2 above. A CHOOSE_REFRESH algorithm is *optimal* if it finds the cheapest subset T_R of T's tuples to refresh (*i.e.*, the subset with the least total cost) that guarantees the final answer to the query will satisfy the precision constraint for any precise values of the refreshed tuples within the current bounds.

We are assuming that the cost to refresh a set of tuples is the sum of the costs of refreshing each member of the set, in order to keep the optimization problem manageable. This simplification ignores possible amortization due to batching multiple requests to the same source. Also recall that we assume a separate refresh cost may be assigned to each tuple, although in practice all tuples from the same source may incur the same cost.

Note that the entire set T_R of tuples to refresh is selected before the refreshes actually occur, so the precision constraint must be guaranteed for any possible precise values for the tuples in T_R. A different approach is to refresh tuples one at a time (or one source at a time), computing a bounded answer after each refresh and stopping when the answer is precise enough. See Section 8.2 for further discussion.

5 Aggregation without Selection Predicates

This section specifies how to compute a bounded answer from bounded data values for each type of aggregation function, and describes algorithms for selecting refresh sets for each aggregation function. For now, we assume that any selection predicate in the TRAPP/AG query involves only columns that contain exact values. Thus, in this section we assume that the selection predicate has already been applied and the aggregation is to be computed over the tuples that satisfy the predicate. TRAPP/AG queries with selection predicates involving columns that contain bounded values are covered in Section 6, and joins involving bounded values are discussed in Section 7.

Suppose we want to compute an aggregate over column $T.a$ of a cached table T. The value of $T.a$ for each tuple t_i is stored in the cache as a bound $[L_i, H_i]$. While computing the aggregate, the query processor has the option for each tuple t_i of either reading the cached bound $[L_i, H_i]$ or refreshing t_i to obtain the master value V_i. The cost to refresh t_i is C_i. The final answer to the aggregate is a bound $[L_A, H_A]$.

5.1 Computing MIN with No Selection Predicate

Computing the bounded MIN of $T.a$ is straightforward:

$$[L_A, H_A] = [\min_{t_i \in T}(L_i), \min_{t_i \in T}(H_i)]^1$$

The lowest possible value for the minimum (L_A) occurs if for all $t_i \in T$, $V_i = L_i$, *i.e.*, each value is at the bottom of its bound. Conversely, the highest possible value for the minimum (H_A) occurs if $V_i = H_i$ for all tuples. Returning to our example of Section 1.1, suppose we want to find the minimum bandwidth link along the path $N_1 \rightarrow N_2 \rightarrow N_4 \rightarrow N_5 \rightarrow N_6$, *i.e.*, query *Q1*. Applying the bounded MIN of *bandwidth* to tuples T = $\{1, 2, 5, 6\}$ in Figure 2 yields $[40, 55]$.

Choosing an optimal set of tuples to refresh for a MIN query with a precision constraint is also straightforward, although the algorithm's justification and proof of optimality is nontrivial (see Appendix B). The CHOOSE_REFRESH$_{\text{NO_SEL/MIN}}$ algorithm chooses T_R to be all tuples $t_i \in T$ such that $L_i < \min_{t_k \in T}(H_k) - R$, where R is the precision constraint, independent of refresh cost. That is, T_R contains all tuples whose lower bound is less than the minimum upper bound minus the precision constraint. If B-tree indexes exist on both the upper and lower bounds,[2] the set T_R can be found in time less than $O(|T|)$ by first using the index on upper bounds to find $\min_{t_k \in T}(H_k)$, and then using the index on lower bounds to find tuples that satisfy $L_i < \min_{t_k \in T}(H_k) - R$. Without these two indexes, the running time for CHOOSE_REFRESH$_{\text{NO_SEL/MIN}}$ is $O(|T|)$.

Consider again our example query *Q1*, which finds the minimum bandwidth along path $N_1 \rightarrow N_2 \rightarrow N_4 \rightarrow N_5 \rightarrow N_6$. CHOOSE_REFRESH$_{\text{NO_SEL/MIN}}$ with $R = 10$ would choose to refresh tuple 5, since it is the only tuple among $\{1, 2, 5, 6\}$ whose low value is less than $\min_{t_k \in \{1,2,5,6\}}(H_k) - R = 55 - 10 = 45$. After refreshing, tuple 5's bandwidth value turns out to be 50, so the new bounded answer is [45, 50].

The MAX aggregation function is symmetric to MIN. See Appendix C.1 for details.

[1] In this and all subsequent formulas, we define $\min(\emptyset) = +\infty$ and $\max(\emptyset) = -\infty$.

[2] Section 8.3 briefly discusses indexing time-varying range endpoints, a problem on which we are actively working.

5.2 Computing SUM with No Selection Predicate

To compute the bounded SUM aggregate, we take the sum of the values at each extreme:

$$[L_A, H_A] = [\sum_{t_i \in T} L_i, \sum_{t_i \in T} H_i]$$

The smallest possible sum occurs when all values are as low as possible, and the largest possible sum occurs when all values are as high as possible. In our running example, the bounded SUM of *latency* along the path $N_1 \rightarrow N_2 \rightarrow N_4 \rightarrow N_5 \rightarrow N_6$ (query *Q2*) using the data from Figure 2 is [19, 28].

The problem of selecting an optimal set T_R of tuples to refresh for SUM queries with precision constraints is better attacked as the equivalent problem of selecting the tuples not to refresh: $\overline{T_R} = T - T_R$. We first observe that $H_A - L_A = \sum_{t_i \in T} H_i - \sum_{t_i \in T} L_i = \sum_{t_i \in T}(H_i - L_i)$. After refreshing all tuples $t_j \in T_R$, we have $H_j - L_j = 0$, so these values contribute nothing to the bound. Thus, after refresh, $\sum_{t_i \in T}(H_i - L_i) = \sum_{t_i \in \overline{T_R}}(H_i - L_i)$. These equalities combined with the precision constraint $H_A - L_A \leq R$ give us the constraint $\sum_{t_i \in \overline{T_R}}(H_i - L_i) \leq R$. The optimization objective is to satisfy this constraint while minimizing the total cost of the tuples in T_R. Observe that minimizing the total cost of the tuples in T_R is equivalent to maximizing the total cost of the tuples not in T_R. Therefore, the optimization problem can be formulated as choosing $\overline{T_R}$ so as to maximize $\sum_{t_i \in \overline{T_R}} C_i$ under the constraint $\sum_{t_i \in \overline{T_R}}(H_i - L_i) \leq R$.

It turns out that this problem is isomorphic to the well-known *0/1 Knapsack Problem* [CLR90], which can be stated as follows: We are given a set S of items that each have weight W_i and profit P_i, along with a knapsack with capacity M (*i.e.*, it can hold any set of items as long as their total weight is at most M). The goal of the Knapsack Problem is to choose a subset S_K of the items in S to place in the knapsack that maximizes total profit without exceeding the knapsack's capacity. In other words, choose S_K so as to maximize $\sum_{i \in S_K} P_i$ under the constraint $\sum_{i \in S_K} W_i \leq M$. To state the problem of selecting refresh tuples for bounded SUM queries as the 0/1 Knapsack Problem, we assign $S = T$, $S_K = \overline{T_R}$, $P_i = C_i$, $W_i = (H_i - L_i)$, and $M = R$.

Unfortunately, the 0/1 Knapsack Problem is known to be NP-Complete [GJ79]. Hence all known approaches to solving the problem optimally, such as dynamic programming, have a worst-case exponential running time. Fortunately, an approximation algorithm exists that, in polynomial time, finds a solution having total profit that is within a fraction ϵ of optimal for any $0 < \epsilon < 1$ [IK75]. The running time of the algorithm is $O(n \cdot \log n) + O((\frac{3}{\epsilon})^2 \cdot n)$. We use this algorithm for $\text{CHOOSE\_REFRESH}_{\text{NO\_SEL/SUM}}$. Adjusting parameter ϵ in the algorithm allows us to trade off the running time of the algorithm against the quality of the solution.

In the special case of uniform costs ($C_i = C_j$ for all tuples t_i and t_j), all knapsack objects have the same profit P_i, and the 0/1 Knapsack Problem has a polynomial algorithm [CLR90]. The optimal answer then can be found by "placing objects in the knapsack" in order of increasing weight W_i until the knapsack cannot hold any more objects. That is, we add tuples to $\overline{T_R}$ starting with the smallest $H_i - L_i$ bounds until the next tuple would cause $\sum_{t_i \in \overline{T_R}}(H_i - L_i) > R$. If an index exists on the bound width $H_i - L_i$ (see Section 8.3), this algorithm can run in sublinear time. Without an index on bound width, the running time of this algorithm is $O(n \cdot \log n)$, where $n = |T|$.

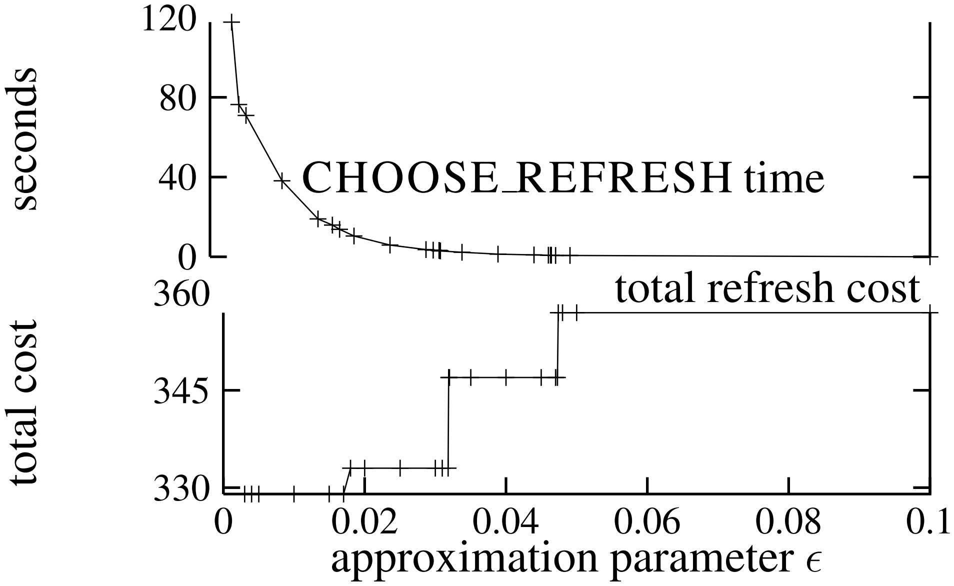


Figure 5: CHOOSE_REFRESH$_{\text{NO_SEL/SUM}}$ time and refresh cost for varying ϵ.

Consider again query *Q2* that asks for the total latency along path $N_1 \rightarrow N_2 \rightarrow N_4 \rightarrow N_5 \rightarrow N_6$. Figure 2 shows the correspondence between our problem and the Knapsack Problem by specifying the knapsack "weight" $W = H - L$ for the *latency* column of each tuple in $\{1, 2, 5, 6\}$. Using the exponential (optimal) knapsack algorithm to find the total latency along path $N_1 \rightarrow N_2 \rightarrow N_4 \rightarrow N_5 \rightarrow N_6$ with $R = 5$, tuples 2 and 5 are "placed in the knapsack" (whose capacity is 5), leaving $T_R = \{1, 6\}$. The bounded SUM of *latency* after refreshing tuples 1 and 6 is [21, 26].

5.2.1 Performance Experiments

CHOOSE_REFRESH$_{\text{NO_SEL/SUM}}$ uses the approximation algorithm from [IK75] to quickly find a cheap set of tuples T_R to refresh such that the precision constraint is guaranteed to hold. We implemented the algorithm and ran experiments using 90 actual stock prices that varied highly in one day. The high and low values for the day were used as the bounds $[L_i, H_i]$, the closing value was used as the precise value V_i, and the refresh cost C_i for each data object was set to a random number between 1 and 10. Running times were measured on a Sun Ultra-1 Model 140 running SunOS 5.6. In Figure 5 we fix the precision constraint $R = 100$ and vary ϵ in the knapsack approximation in order to plot CHOOSE_REFRESH time and total refresh cost of the selected tuples. Smaller values for ϵ increase the CHOOSE_REFRESH time but decrease the refresh cost. However, since the CHOOSE_REFRESH time increases quadratically while the refresh cost only decreases by a small fraction, it is not in general advantageous to set ϵ below 0.1 (which comes very close to optimal) unless refreshing is extremely expensive.

In Figure 6 we fix the approximation parameter $\epsilon = 0.1$ and vary R in order to plot precision (precision constraint R) versus performance (total refresh cost) for our CHOOSE_REFRESH$_{\text{NO_SEL/SUM}}$ algorithm. This graph, a concrete instantiation of Figure 1(b), clearly shows the continuous, monotonically decreasing tradeoff between precision and performance that characterizes TRAPP systems.

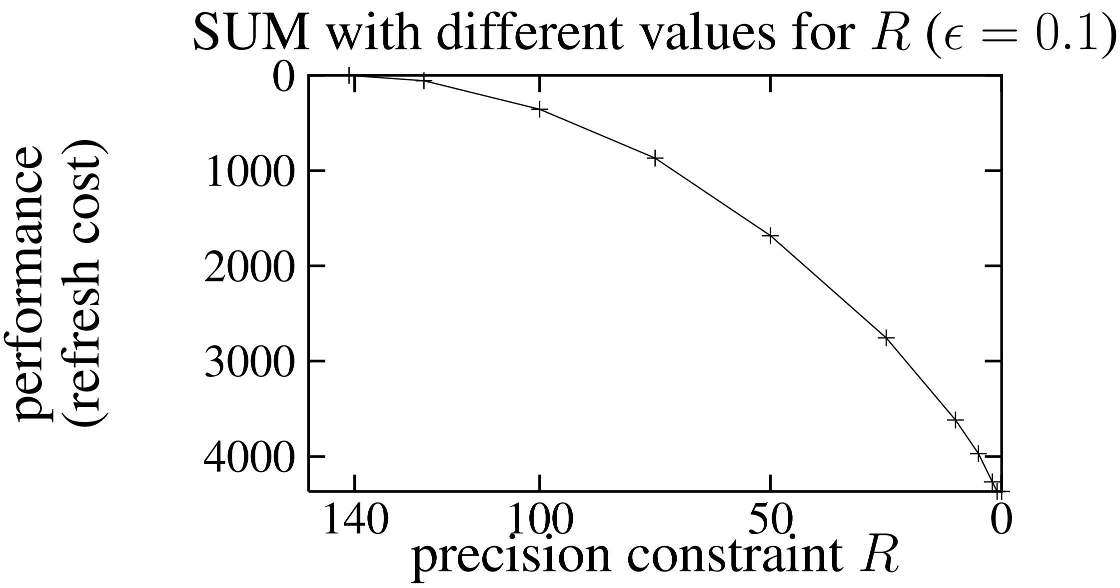


Figure 6: Precision-performance tradeoff for CHOOSE_REFRESH$_{\text{NO_SEL/SUM}}$.

5.3 Computing COUNT with No Selection Predicate

When no selection predicate is present, computing COUNT amounts to computing the cardinality of the table. Since we currently require all insertions and deletions to be propagated immediately to the data caches (Section 3), the cardinality of the cached copy of a table is always equal to the cardinality of the master copy, so there is no need for refreshes.

5.4 Computing AVG with No Selection Predicate

When no selection predicate is present, the procedure for computing the AVG aggregate is as follows. First, compute $COUNT$, which as discussed in Section 5.3 is simply the cardinality of the cached T. Then, compute the bounded SUM as described in Section 5.2 with $R = R \cdot COUNT$ to produce $[L_{SUM}, H_{SUM}]$. Finally, let:

$$[L_A, H_A] = [\frac{L_{SUM}}{COUNT}, \frac{H_{SUM}}{COUNT}]$$

Since the bound width $H_A - L_A = \frac{H_{SUM} - L_{SUM}}{COUNT}$, by computing SUM such that $H_{SUM} - L_{SUM} \leq R \cdot COUNT$, we are guaranteeing that $H_A - L_A \leq R$, and the precision constraint is satisfied. The running time is dominated by the running time of the CHOOSE_REFRESH$_{\text{NO_SEL/SUM}}$ algorithm, which is given in Section 5.2.

Consider query *Q3* from Section 1.1 to compute the average traffic level in the entire network, and let precision constraint $R = 10$. We first compute $COUNT = 6$, and then compute SUM with $R = R \cdot COUNT = 10 \cdot 6 = 60$. The column labeled W' in Figure 2 shows the knapsack weight assigned to each tuple based on the cached bounds for *traffic*. Using the optimal Knapsack algorithm, the SUM computation will cause tuples 5 and 6 to be refreshed, resulting in a bounded SUM of [618, 678]. Dividing by $COUNT = 6$ gives a bounded AVG of [103, 113].

	$(bandwidth > 50) \wedge (latency < 10)$		$latency > 10$		$traffic > 100$	
	before refresh	*after refresh*	*before refresh*	*after refresh*	*before refresh*	*after refresh*
1	T^+	T^+	T^-	T^-	$T^?$	T^-
2	$T^?$	T^+	T^-	T^-	T^+	T^+
3	T^-	T^-	T^+	T^+	$T^?$	T^+
4	$T^?$	T^+	$T^?$	T^-	T^+	T^+
5	$T^?$	T^-	$T^?$	T^+	$T^?$	T^-
6	$T^?$	T^-	T^-	T^-	$T^?$	T^+

Figure 7: Classification of tuples into T^-, $T^?$, and T^+ for three selection predicates.

6 Modifications to Incorporate Selection Predicates

When a selection predicate involving bounded values is present in the query, both computing bounded aggregate results and choosing refresh tuples to meet the precision constraint become more complicated. This section presents modifications to the algorithms in Section 5 to handle single-table aggregation queries with selection predicates. We begin by introducing techniques common to all TRAPP/AG queries with predicates, regardless of which aggregation function is present.

Consider a selection predicate involving at least one column of T that contains bounded values. The system can partition T into three disjoint sets: T^-, $T^?$, and T^+. T^- contains those tuples that cannot possibly satisfy the predicate given current bounded data. T^+ contains tuples that are guaranteed to satisfy the predicate given current bounded data. All other tuples are in $T^?$, meaning that there exist some precise values within the current bounds that will cause the predicate to be satisfied, and other values that will cause the predicate not to be satisfied. The process of classifying tuples into T^-, $T^?$, and T^+ when the selection predicate involves at least one column with bounded values is detailed in Appendix D. The most interesting aspect is that filters over T that find the tuples in T^+ and $T^?$ can always be expressed as simple predicates over bounded value endpoints, and all of our algorithms for computing bounded answers and choosing tuples to refresh examine only tuples in T^+ and $T^?$. Therefore, the classification can be expressed as SQL queries and optimized by the system, possibly incorporating specialized indexes as discussed in Section 8.3.

For examples in the remainder of this section we refer to Figure 7, which shows the classification for three different predicates over the data from Figure 2, both before and after the exact values are refreshed.

6.1 Computing MIN with a Selection Predicate

When a selection predicate is present, the bounded MIN answer is:

$$[L_A, H_A] = [\min_{t_i \in T^+ \cup T^?}(L_i), \min_{t_i \in T^+}(H_i)]$$

In the "worst case" for L_A, all tuples in $T^?$ satisfy the predicate (*i.e.*, they turn out to be in T^+), so the smallest lower bound of any tuple that might satisfy the predicate forms the lower bound for the answer. In the "worst case" for H_A, tuples in $T^?$ do not satisfy the predicate (*i.e.*, they turn out to be in T^-), so the smallest upper bound of the tuples guaranteed to satisfy the predicate forms the only guaranteed upper bound for the answer. In our running example, consider query *Q4*: find the minimum *traffic* where $(bandwidth > 50) \wedge (latency < 10)$. The result using the data from Figure 2 and classifications from Figure 7 is [90, 105].

CHOOSE_REFRESH$_{\text{MIN}}$ chooses T_R to be exactly the tuples $t_i \in T^+ \cup T^?$ such that $L_i < \min_{t_k \in T^+}(H_k) - R$. This algorithm is essentially the same as CHOOSE_REFRESH$_{\text{NO_SEL/MIN}}$, and is correct and optimal for the same reason (see Appendix B). The only additional case to consider is that refreshing tuples in $T^?$ may move them into T^-. However, such tuples do not contribute to the actual MIN, and thus do not affect the bound of the answer $[L_A, H_A]$. Hence, the precision constraint is still guaranteed to hold. As with CHOOSE_REFRESH$_{\text{NO_SEL/MIN}}$ the running time for CHOOSE_REFRESH$_{\text{MIN}}$ can be sublinear if B-tree indexes are available on both the upper and lower bounds. Otherwise, the worst-case running time for CHOOSE_REFRESH$_{\text{MIN}}$ is $O(n)$.

For our query *Q4* with precision constraint $R = 10$, CHOOSE_REFRESH$_{\text{MIN}}$ chooses T_R = {5, 6}, since tuples 5 and 6 may pass the selection predicate and their low values are less than $\min_{t_k \in T^+}(H_k) - R = 105 - 10 = 95$. After refreshing, tuples 5 and 6 turn out not to pass the selection predicate, so the bounded MIN is [95, 105].

The MAX aggregation function is symmetric to MIN. See Appendix C.2 for details.

6.2 Computing SUM with a Selection Predicate

To compute SUM in the presence of a selection predicate:

$$[L_A, H_A] = [\sum_{t_i \in T^+} L_i + \sum_{t_i \in T^? \wedge L_i < 0} L_i, \sum_{t_i \in T^+} H_i + \sum_{t_i \in T^? \wedge H_i > 0} H_i]$$

The "worst case" for L_A occurs when all and only those tuples in $T^?$ with negative values for L_i satisfy the selection predicate and thus contribute to the result. Similarly, the "worst case" for H_A occurs when only tuples in $T^?$ with positive values for H_i satisfy the predicate.

The CHOOSE_REFRESH$_{\text{SUM}}$ algorithm is similar to CHOOSE_REFRESH$_{\text{NO_SEL/SUM}}$, which maps the problem to the 0/1 Knapsack Problem (Section 5.2). The following two modifications are required. First, we ignore all tuples $t_i \in T^-$. Second, for tuples $t_i \in T^?$, we set W_i to one of three possible values. If $L_i \geq 0$, let W_i = $H_i - 0$ = H_i. If $H_i \leq 0$, let W_i = $0 - L_i$ = $-L_i$. Otherwise, let W_i = $(H_i - L_i)$ as before. The idea is that we want to effectively extend the bounds for all tuples in $T^?$ to include 0, since it is possible that these tuples are actually in T^- and thus do not contribute to the SUM (*i.e.*, contribute value 0). In the knapsack formulation, to extend the bounds to 0 we need to adjust the weights as specified above.

6.3 Computing COUNT with a Selection Predicate

The bounded answer to the COUNT aggregation function in the presence of a selection predicate is:

$$[L_A, H_A] = [|T^+|, |T^+| + |T^?|]$$

For example, consider query *Q5* from Section 1.1 that asks for the number of links that have $latency > 10$. Figure 7 shows the classification of tuples into T^-, $T^?$, and T^+. Since $|T^+| = 1$ and $|T^?| = 2$, the bounded COUNT is [1, 3].

The CHOOSE_REFRESH$_{\text{COUNT}}$ algorithm is based on the fact that $H_A - L_A = |T^?|$, and that refreshing a tuple in $T^?$ is guaranteed to remove it from $T^?$. Given these two facts, the optimal CHOOSE_REFRESH$_{\text{COUNT}}$ algorithm is to let T_R be the $\lceil |T^?| - R \rceil$ cheapest tuples in $T^?$. Using a B-tree index on cost, this algorithm runs in sublinear time. Otherwise, the worst-case running time for CHOOSE_REFRESH$_{\text{COUNT}}$ requires a sort and is $O(n \cdot \log n)$.

Consider again query *Q5* and suppose $R = 1$. Since $|T^?| = 2$, CHOOSE_REFRESH$_{\text{COUNT}}$ selects T_R = $\{5\}$, which is the $\lceil |T^?| - R \rceil = \lceil 2 - 1 \rceil = 1$ cheapest tuple in $T^?$. After updating this tuple (which turns out to be in T^+), the bounded COUNT is [2, 3].

6.4 Computing AVG with a Selection Predicate

6.4.1 Computing the Bounded Answer

Computing the bounded AVG when a predicate is present is somewhat more complicated than computing the other aggregates. With a predicate, COUNT is a bounded value as well as SUM, so it is no longer a simple matter of dividing the endpoints of the SUM bound by the exact COUNT value (as in Section 5.4). To compute the lower bound on AVG, we start by computing the average of the low endpoints of the T^+ bounds, and then average in the low endpoints of the $T^?$ bounds one at a time in increasing order until the point at which the average increases. Computing the upper bound on AVG is the reverse. For example, consider query *Q6* from Section 1.1 that asks for the average latency for links having $traffic > 100$. To compute the lower bound, we start by averaging the low endpoints of T^+ tuples 2 and 4, and then average in the low endpoints of $T^?$ tuples 1 and then 6 to obtain a lower bound on average latency of 5. We stop at this point since averaging in further $T^?$ tuples would increase the lower bound. Appendix E formalizes this computation, which has a worst-case running time of $O(n \cdot \log n)$.

A looser bound for AVG can be computed in linear time by first computing SUM as $[L_{SUM}, H_{SUM}]$ and COUNT as $[L_{COUNT}, H_{COUNT}]$ using the algorithms from Sections 6.2 and 6.3, then setting:

$$[L_A, H_A] = [\min(\frac{L_{SUM}}{H_{COUNT}}, \frac{L_{SUM}}{L_{COUNT}}), \max(\frac{H_{SUM}}{L_{COUNT}}, \frac{H_{SUM}}{H_{COUNT}})]$$

In our example, $[L_{SUM}, H_{SUM}] = [14, 55]$ and $[L_{COUNT}, H_{COUNT}] = [2, 6]$. Thus, the linear algorithm yields [2.3, 27.5]. Notice that this bound is indeed looser than the [5, 11.3] bound achieved by the $O(n \cdot \log n)$ algorithm above.

6.4.2 Choosing Tuples to Refresh

CHOOSE_REFRESH$_{\text{AVG}}$ is our most complicated scenario. Details are provided in Appendix F. Here we give a very brief description.

Our CHOOSE_REFRESH$_{\text{AVG}}$ algorithm uses the fact that a loose bound on AVG can be achieved as a function of the bounds for SUM and COUNT, as in the linear algorithm in Section 6.4.1 above. We choose refresh tuples that provide bounds for SUM and COUNT such that the bound for AVG as a function of the bounds for SUM and COUNT meets the precision constraint. This interaction is accomplished by using a modified version of the CHOOSE_REFRESH$_{\mathrm{SUM}}$ algorithm that understands how the choice of refresh tuples for SUM affects the bound for COUNT. This algorithm sets a precision constraint for SUM that takes into account the changing bound for COUNT to guarantee that the overall precision constraint on AVG is met. CHOOSE_REFRESH$_{\mathrm{AVG}}$ preserves the Knapsack Problem structure. Therefore, choosing refresh tuples for AVG can be accomplished by solving the 0/1 Knapsack Problem, and it has the same complexity as CHOOSE_REFRESH$_{\mathrm{NO_SEL/SUM}}$ (see Section 5.2).

In our example query *Q6* above, if we set $R = 2$ then CHOOSE_REFRESH$_{\mathrm{AVG}}$ chooses a knapsack capacity of $M = 4$ and assigns a weight to each tuple as shown in the column labeled W'' in Figure 2. The knapsack optimally "contains" tuples 2 and 4. After refreshing the other tuples $T_R = \{1, 3, 5, 6\}$, the bounded AVG is [8, 9].

7 Aggregation Queries with Joins

Computing the bounded answer to an aggregation query with a join expression (*i.e.*, with multiple tables in the `FROM` clause) is no different from doing so with a selection predicate: in most SQL queries, a join is expressed using a selection predicate that compares columns of more than one table. Our method for determining membership of tuples in T^+, $T^?$, and T^- applies to join predicates as well as selection predicates. As before, the classification can be expressed as SQL queries and optimized by the system to use standard join techniques, possibly incorporating specialized indexes as discussed in Section 8.3.

On the other hand, choosing tuples to refresh is significantly more difficult in the presence of joins. First, since there are several "base" tuples contributing to each "aggregation" (joined) tuple, we can choose to refresh any subset of the base tuples. Each subset might shrink the answer bound by a different amount, depending how it affects the T^+, $T^?$, T^- classification combined with its effect on the aggregation column. Second, since each base tuple can potentially contribute to multiple aggregation tuples, refreshing a base tuple for one aggregation tuple can also affect other aggregation tuples. These interactions make the problem quite complex. We have considered various heuristic algorithms that choose tuples to refresh for join queries. Currently, we are investigating the exact complexity of the problem and hope to find an approximation algorithm with a tunable ϵ parameter, as in the approximation algorithm for CHOOSE_REFRESH$_{\mathrm{SUM}}$.

8 Status and Future Work

We have implemented all of the bounded aggregation functions and CHOOSE_REFRESH algorithms presented in this paper, and implementation of the source-cache cooperation discussed in Sections 3.1 and 3.2 is underway. In addition to testing our algorithms in a realistic environment, we plan to study how the choice of bound width (Section 3.2 and Appendix A) affects the refresh frequency, and we plan to investigate alternative methods of choosing bound functions.

This paper represents our initial work in TRAPP replication systems, so there are numerous avenues for future work. We divide the future directions into four categories: additional functionality (Section 8.1), choosing tuples to refresh (Section 8.2), improving performance (Section 8.3), and real-time and availability issues (Section 8.4).

8.1 Additional Functionality

- **Expanding the class of aggregation queries we consider.** We want to devise algorithms for other aggregation functions, such as MEDIAN (for which we have preliminary results [FMP+00]) and TOP-n. In addition, we would like to extend our results to handle grouping on bounded values, enabling GROUP-BY and COUNT UNIQUE queries. We would also like to handle nested aggregation functions such as MAX(AVG), which requires understanding how the precision of the bounded results of the inner aggregate affects the precision of the outer aggregate.
- **Looking beyond aggregation queries.** We believe that the TRAPP idea can be expanded to encompass other types of relational and non-relational queries having different precision constraints. In our running example (Section 1.1), suppose we wish to find the lowest latency path in the network from node N_i to node N_j. A precision constraint might require that the value corresponding to the answer returned by TRAPP (*i.e.*, the latency of the selected path) is within some distance from the value of the precise best answer.
- **Allowing users to express relative instead of absolute precision constraints.** A relative precision constraint might be expressed as a constant $P \geq 0$ that denotes an absolute precision constraint of $2 \cdot A \cdot P$, where A is the actual answer. The difficulty is that A is not known in advance. Based on the bound on A derived in the first pass from cached data alone, it is possible to find a conservative absolute precision constraint $R \leq 2 \cdot A \cdot P$ to use in our algorithms. However, it might be possible to redesign our algorithms to perform better with relative bounds.
- **Considering probabilistic precision guarantees.** TRAPP systems as defined in this paper improve performance by providing bounded answers, while offering absolute guarantees about precision. As discussed in Section 2, other approaches improve performance by giving probabilistic guarantees about precision. An interesting direction is to combine the two for even better performance: provide bounded answers with probabilistic precision guarantees.
- **Considering applying our TRAPP ideas to *multi-level* replication systems, where each data object resides on one source and there is a hierarchy of data caches.** Refreshes would then occur between a cache and the caches or sources one level below, with a possible cascading effect. A current example of such a scenario is Web caching systems (*e.g.*, *Inktomi Traffic Server* [Ink99]), which reside between Web servers and end-user Web browsers.
- **Extending data visualization techniques to take advantage of TRAPP.** We are currently investigating ways to extend data visualization systems (*e.g.*, [OWA+98]) to display images based on bounded data instead of precise data, perhaps by drawing fuzzy regions to indicate uncertainty. A visualization in a TRAPP setting could be modeled as a continuous query in which precision constraints are formulated in the visual domain and upheld by TRAPP.

8.2 Choosing Tuples to Refresh

- **Adapting our** CHOOSE_REFRESH **algorithms to take refresh batching into account.** If multiple query-initiated refreshes are sent to the same source, the overall cost may be less than the sum of the individual costs. We would like to adapt our CHOOSE_REFRESH algorithms to take into account such cases where refreshing one tuple reduces the cost of refreshing other tuples. In fact, the same adaptation may help us develop CHOOSE_REFRESH algorithms for queries involving join and group-by expressions. In both of these cases, refreshing a tuple for one purpose (one group or joined tuple) may reduce the subsequent cost for another purpose (group or joined tuple).

- **Considering iterative** CHOOSE_REFRESH **algorithms.** Rather than choosing a set of tuples in advance that guarantees adequate precision regardless of actual exact values, we could refresh tuples iteratively until the precision constraint is met. In addition to developing the alternative suite of algorithms, it will be interesting to investigate in which contexts an iterative method is preferable to the batch method presented in this paper. Also, we could use an iterative method to give bounded aggregation queries an "online" behavior [HAC+99], where the user is presented with a bounded answer that gradually refines to become more precise over time. In this scenario, the goal is to shrink the answer bound as fast as possible.

8.3 Improving Performance

- **Delaying the propagation of insertions and deletions to data caches.** We are currently investigating ways in which discrepancies in the number of tuples can be bounded, and the computation of the bounded answer to a query can take into account these bounded discrepancies. Sources will then no longer be forced to send a refresh every time an object is inserted or deleted.

- **Investigating specialized bound functions suitable for update patterns with known properties.** The bound function shape we suggested in this paper (Section 3.2) is based on the assumption that no information about the update pattern is available.

- **Considering ways to amortize refresh costs by *refresh piggybacking* and *pre-refreshing*.** When a (value- or query-initiated) refresh occurs, the source may wish to "piggyback" extra refreshes along with the one requested. These extra refreshes would consist of values that are likely to need refreshing in the near future, *e.g.*, if the precise value is very close to the edge of its bound. The amount of refresh piggybacking to perform would depend on the benefit of doing so versus the added overhead. Additionally, it might be beneficial to perform *pre-refreshing*, by sending unnecessary refreshes when system load is low that may be useful in future processing.

- **Investigating storage, indexing, and query processing issues over bounded values.** We are currently designing and evaluating schemes for indexing bounds that are functions of time with a square-root shape, as discussed in Section 3.2. Also, we plan to weigh the advantages of using functions for bounds versus potential indexing improvements when bounds are constants. We also plan to study ways in which cached data objects stored as pairs of bound functions might be compressed. Without compression, caches must store two values for each data object (Appendix A), and sources must transmit these two values for each tuple being refreshed. Furthermore, the Refresh Monitor at each source must keep track of the bound functions for each remotely cached data object. Compression issues can be addressed without affecting the techniques presented in this paper: our CHOOSE_REFRESH algorithms are independent of which bound functions are used or how they are represented, and we have not yet focused on query processing issues.

8.4 Real-time and Consistency Issues

- **Handling refresh delay.** Since message-passing over a network is not instantaneous, in a value-initiated refresh there is some delay between the time a master value exceeds a cached bound and the time the cache is refreshed. Consequently, a cached bound can be "stale" for a short period of time. One way to avoid this problem is by pre-refreshing a value when it is close to the edge of its bound.
- **Evaluating concurrency control solutions.** If value-initiated refreshes are permitted to occur during the CHOOSE_REFRESH computation or while a query is being evaluated (or in between), the answer could reflect inconsistent data or could fail to satisfy the precision constraint. One solution is to implement multiversion concurrency control [BHG87], which would permit refreshes to occur at any time, while still allowing each in-progress query to read data that was current when the query started.

Acknowledgments

We thank Hector Garcia-Molina, Taher Haveliwala, Rajeev Motwani, and Suresh Venkatasubramanian for useful discussions. We also thank Joe Hellerstein and some anonymous referees for helpful comments on an initial draft. Finally, we thank Sergio Marti for useful discussions about network monitoring.

References

[ABGMA88] R. Alonso, D. Barbara, H. Garcia-Molina, and S. Abad. Quasi-copies: Efficient data sharing for information retrieval systems. In *Proceedings of the International Conference on Extending Database Technology*, pages 443–468, Venice, Italy, March 1988.

[AKG87] S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 34–48, San Francisco, California, May 1987.

[BGM92] D. Barbara and H. Garcia-Molina. The Demarcation Protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *Proceedings of the International Conference on Extending Database Technology*, pages 373–387, Vienna, Austria, March 1992.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BK95] A. Brodsky and Y. Kornatzky. The LyriC language: Querying constraint objects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 35–46, San Jose, California, May 1995.

[BL99] M. Benedikt and L. Libkin. Exact and approximate aggregation in constraint query languages. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 102–113, Philadelphia, Pennsylvania, May 1999.

[BSCE99] A. Brodsky, V. E. Segal, J. Chen, and P. A. Exarkhopoulo. The CCUBE constraint object-oriented database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 577–579, Philadelphia, Pennsylvania, June 1999.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[FMP$^+$00] T. Feder, R. Motwani, R. Panigrahy, C. Olston, and J. Widom. Computing the median with uncertainty. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*, Portland, Oregon, May 2000.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, June 1996.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, New York, 1979.

[GKP89] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, Massachusetts, 1989.

[GM98] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 331–342, Seattle, Washington, June 1998.

[HAC$^+$99] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. Haas. Interactive data analysis with CONTROL. *IEEE Computer*, August 1999.

[HCH$^+$99] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *Proceedings of the 15th International Conference on Data Engineering*, pages 266–275, Sydney, Austrialia, March 1999.

[HH97] J. M. Hellerstein and P. J. Haas. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 171–182, Tucson, Arizona, May 1997.

[HSW94] Y. Huang, R. Sloan, and O. Wolfson. Divergence caching in client-server architectures. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 131–139, Austin, Texas, September 1994.

[IK75] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, October 1975.

[Ink99] Inktomi. Inktomi traffic server, 1999. http://www.inktomi.com /products/network/traffic/product.html.

[JV96] N. Jukic and S. Vrbsky. Aggregates for approximate query processing. In *Proceedings of ACMSE*, pages 109–116, April 1996.

[KKR90] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 299–313, Nashville, Tennessee, April 1990.

[Kup93] G. M. Kuper. Aggregation in constraint databases. In *Proceedings of the First Workshop on Principles and Practice of Constraint Programming*, Newport, Rhode Island, April 1993.

[Lip79] W. Lipski, Jr. On semantic issues connected with incomplete information databases. *ACM Transactions on Database Systems*, 4(3):262–296, September 1979.

[Mor80] J. P. Morgenstein. Computer based management information systems embodying answer accuracy as a user parameter. Ph.D. thesis, U.C. Berkeley Computer Science Division, 1980.

[NLF99] F. Naumann, U. Leser, and J. Freytag. Quality-driven integration of heterogeneous information systems. In *Proceedings of the Twenty-Fifth International Conference on Very Large Data Bases*, Edinburgh, U.K., September 1999.

[OWA+98] C. Olston, A. Woodruff, A. Aiken, M. Chu, V. Ercegovac, M. Lin, M. Spalding, and M. Stonebraker. DataSplash. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 550–552, Seattle, Washington, June 1998.

[PG99] V. Poosala and V. Ganti. Fast approximate query answering using precomputed statistics. In *Proceedings of the IEEE International Conference on Data Engineering*, page 252, Sydney, Australia, March 1999.

[RB89] E. A. Rundensteiner and L. Bic. Aggregates in possibilistic databases. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 287–295, Amsterdam, The Netherlands, August 1989.

[RFS92] R. L. Read, D. S. Fussell, and A. Silberschatz. A multi-resolution relational data model. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 139–150, Vancouver, Canada, August 1992.

[WXCJ98] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Proceedings of the Tenth International Conference on Scientific and Statistical Database Management*, pages 111–122, Capri, Italy, July 1998.

A Choosing Good Bound Functions

Returning to the issues alluded to in Section 3.2, we now briefly discuss how good bound functions are selected. To make the problem of choosing good bound functions more manageable, we separate it into two subproblems: (1) choosing the overall *shape* of the bound functions, which we will determine statically and call $f(\mathcal{T})$; and (2) choosing a *width parameter* of the bound for each data object, which is done by the sources at run-time. Assuming we choose a monotonically increasing function of time $f(\mathcal{T})$, for data object O_i we let lower bound $L_i(\mathcal{T}) = V_i(\mathcal{T}_r) - W_i \cdot f(\mathcal{T} - \mathcal{T}_r)$ and upper bound $H_i(\mathcal{T}) = V_i(\mathcal{T}_r) + W_i \cdot f(\mathcal{T} - \mathcal{T}_r)$, with the width parameter $W_i \geq 0$ chosen by a run-time algorithm. Now that we have decomposed the overall problem into two subproblems, we are faced with the tasks of selecting a function $f(\mathcal{T})$ (the shape) and an algorithm for choosing W_i (the width parameter).

Notice that representing pairs of bound functions this way has the added benefit that they can be encoded by two numbers: the current value $V_i(\mathcal{T}_r)$ and the width parameter W_i, which are transmitted from a source to a cache at refresh time. In addition, the cache must be able to compute $\mathcal{T} - \mathcal{T}_r$, *i.e.*, the elapsed time since the refresh. If the message-passing delay is non-negligible, then the source must transmit the refresh time $\mathcal{T}_r$ along with $V_i(\mathcal{T}_r)$ and W_i, and clocks must be synchronized within a negligible threshold.

In terms of shape, *i.e.*, function $f(\mathcal{T})$, in the absence of more information we can model the changing value of a data object as a random walk in one dimension. This model is natural for common settings where updates tend to be small increments or decrements to the current value ("escrow transactions"). In the random walk model the value either increases or decreases by a constant amount s at each time step. After $\mathcal{T}$ steps, the probability distribution of the value is a binomial distribution with variance $s^2 \cdot \mathcal{T}$ [GKP89]. Chebyshev's Inequality [GKP89] gives an upper bound on the probability P that the value is beyond any distance k from the starting point: $P \leq \mathcal{T} \cdot (\frac{s}{k})^2$. Therefore, using any fixed probability P (say 5%), $k \leq (\frac{s}{\sqrt{P}}) \cdot \sqrt{\mathcal{T}}$, so the value is within $(\frac{s}{\sqrt{P}}) \cdot \sqrt{\mathcal{T}}$ units of the starting point. Thus the function of time that bounds the value with probability $1 - P$ is proportional to $\sqrt{\mathcal{T}}$. In other words, as the value varies over time, a tight bound has approximately the shape of the square-root function.[3] So, we use $f(\mathcal{T}) = \sqrt{\mathcal{T}}$ for the shape of our bound functions. Thus, bound functions are of the form $[V_i(\mathcal{T}_r) - W_i \cdot \sqrt{\mathcal{T} - \mathcal{T}_r}, V_i(\mathcal{T}_r) + W_i \cdot \sqrt{\mathcal{T} - \mathcal{T}_r}]$. The curves in Figure 4 illustrate square-root functions with varying widths.

Now we sketch a dynamic algorithm to choose a bound width parameter W_i that attempts to minimize the number of refreshes. To avoid value-initiated refreshes (due to updates to the master value), the bound should be wide enough to make it unlikely that the value will exceed the bound. On the other hand, to avoid query-initiated refreshes (due to precision constraints of queries), the bound should be as narrow as possible. Unfortunately, since decreasing the chance of one type of refresh increases the chance of the other, it is not obvious how best to choose a bound width W_i that minimizes the total probability that a refresh will be required.

Since both of the factors that affect the choice of bound width—the variation of data values (which

[3]Intuitively, it makes sense that the result should be a function with a negative second derivative. Note that initially, when $\mathcal{T}$ is small, it is not unlikely for a randomly varying value to move several steps in the same direction, so the function increases rapidly. However, as $\mathcal{T}$ grows large, it becomes less likely that the value will continue to move in the same direction, so the function increases less dramatically.

causes value-initiated refreshes) and the precision requirements of user queries (which cause query-initiated refreshes)—are difficult to predict, we propose an adaptive algorithm that adjusts W_i as conditions change. The strategy is as follows: First start with some value for W_i. Each time a value-initiated refresh occurs (a signal that the bound was too narrow), increase W_i when sending the new bound. Conversely, each time a query-initiated refresh occurs (a signal that the bound was too wide), decrease W_i. This strategy should find a middle ground between very wide bounds that the value never exceeds yet are exceedingly imprecise, and very narrow bounds that are precise but need to be refreshed constantly as the value fluctuates.

As future work, we plan to refine the details of the suggested technique and perform experiments to determine how well it eventually balances the conflicting requirements of queries and updates. We also plan to consider other bound functions for cases where update patterns are known and do not conform to the random walk model.

B Proof of Correctness of $\mathrm{CHOOSE\_REFRESH}_{\mathrm{NO\_SEL/MIN}}$

Recall from Section 5.1 that the $\mathrm{CHOOSE\_REFRESH}_{\mathrm{NO\_SEL/MIN}}$ algorithm chooses T_R to be all tuples $t_i \in T$ such that $L_i < \min_{t_k \in T}(H_k) - R$, where R is the precision constraint. To show that this choice for T_R is correct and optimal, we show that every tuple in T_R must appear in every solution, and that this solution is sufficient to guarantee the precision constraint. First, we show that every tuple in T_R must appear in every solution. Consider any $t_i \in T_R$ and suppose we choose to refresh every tuple in T except t_i. It is possible that refreshing all other tuples t_j results in $V_j = H_j$ for each one (*i.e.*, each precise value is at its upper bound). In this case, after refreshing, our new bounded answer will be $[L_A, H_A]$ where $L_A \leq L_i$ and $H_A = \min_{t_k \in T}(H_k)$. Since $L_i < \min_{t_k \in T}(H_k) - R$ by the definition of T_R, $\min_{t_k \in T}(H_k) - L_i > R$, so $H_A - L_A > R$, and the precision constraint does not hold. Thus, every tuple in T_R must be in any solution to guarantee that the precision constraint will hold.

Next, we show that T_R is sufficient to guarantee the precision constraint. Let L_p be $\min_{t_k \in \overline{T_R}}(L_k)$, where $\overline{T_R} = T - T_R$. Note that for all $t_i \in \overline{T_R}$, L_i is within R of $\min_{t_k \in T}(H_k)$, so we have $\min_{t_k \in T}(H_k) - L_p \leq R$. After tuples in T_R have been refreshed, $\min_{t_k \in T}(H_k)$ can only decrease, so we know $H_A - L_p \leq R$. After refreshing the tuples in T_R, they will have a bound width of zero, *i.e.*, $L_i = H_i = V_i$. There are thus two cases that can occur after the tuples $t_i \in T_R$ have been refreshed. First, if any of the values V_i are less than or equal to L_p, then we can compute an exact minimum. Otherwise, if all of the values V_i are greater than L_p, then $L_A = L_p$. Since $H_A - L_p \leq R$, it follows that $H_A - L_A \leq R$.

C Computing MAX

C.1 Computing MAX with No Selection Predicate

The MAX aggregation function is symmetric to MIN. Thus:

$$[L_A, H_A] = [\max_{t_i \in T}(L_i), \max_{t_i \in T}(H_i)]$$

and the CHOOSE_REFRESH$_{\text{NO_SEL/MAX}}$ algorithm chooses T_R to be all tuples $t_i \in T$ such that $H_i > \max_{t_k \in T}(L_k) + R$.

C.2 Computing MAX with a Selection Predicate

The MAX aggregation function is symmetric to MIN. Thus:

$$[L_A, H_A] = [\max_{t_i \in T^+}(L_i), \max_{t_i \in T^+ \cup T^?}(H_i)]$$

and the CHOOSE_REFRESH$_{\text{MAX}}$ algorithm chooses T_R to be all tuples $t_i \in T^+ \cup T^?$ such that $H_i > \max_{t_k \in T^+}(L_k) + R$.

D Classifying Tuples by a Selection Predicate

The algorithms in Section 6 require that we first classify all tuples in T as belonging to one of T^-, T^+, or $T^?$. Let P be the predicate in the user's query, which we assume is an arbitrary boolean expression involving binary comparisons. We define two transformations on predicate P. The $Possible$ transformation yields an expression that finds tuples that could possibly satisfy the predicate based on bounded values. The $Certain$ transformation yields an expression that finds tuples that are guaranteed to satisfy the predicate based on bounded values. We can apply $Certain(P)$ to find tuples in T^+, and $(Possible(P) \wedge \neg Certain(P))$ to find tuples in $T^?$. All other tuples are in T^-.

Since $Certain(P)$ and $Possible(P)$ are predicates to be evaluated on the tuples of table T, they must be expressed in terms of constants, attributes whose values are exact, and endpoints (denoted *min* and *max*) of attributes whose values are ranges. To handle expressions uniformly, we assume that all values are ranges: in the case of a constant value K (respectively an attribute A whose value is exact), we let $K_{min} = K_{max} = K$ (respectively $A_{min} = A_{max} = A$). Figure 8 gives a set of translation rules—primarily equivalences—specifying how boolean expressions are translated into $Certain$ and $Possible$. These rules are applied recursively to the query's selection predicate P to obtain $Certain(P)$ and $Possible(P)$. Note that disjunction for $Certain$ and conjunction for $Possible$ are implications rather than equivalences. Thus, when we translate $Possible(E_1 \wedge E_2)$ into $Possible(E_1) \wedge Possible(E_2)$ we may classify a tuple into $T^?$ when it should really be in T^-. Also, when we translate $Certain(E_1 \vee E_2)$ into $Certain(E_1) \vee Certain(E_2)$ we may classify a tuple into $T^?$ when it should really be in T^+. Cases where we misclassify tuples are extremely unusual (because they involve very special cases of correlation between subexpressions), and note that these misclassifications affect only the optimality and not the correctness of our algorithms.

We now illustrate how to use the rules in Figure 8 to derive expressions for $Certain(P)$ and $Possible(P)$ in terms of range endpoints. For the predicate $P = (bandwidth > 50) \wedge (latency < 10)$, $Certain(P)$ becomes $(bandwidth_{min} > 50) \wedge (latency_{max} < 10)$, and $Possible(P)$ becomes $(bandwidth_{max} > 50) \wedge (latency_{min} < 10)$. The column labeled "$(bandwidth > 50) \wedge (latency < 10)$ before refresh" of Figure 7 shows the resulting classification of tuples in our example data of Figure 2 into T^-, $T^?$, and T^+.

It turns out that this technique is part of a more general mathematical framework introduced in [Lip79] for evaluating predicates over data objects that have a set of possible values (in our case, an infinite set of

expression E	$Possible(E)$	$Certain(E)$
$[x_{min}, x_{max}] = [y_{min}, y_{max}]$	$\Leftrightarrow (x_{min} \leq y_{max}) \wedge (x_{max} \geq y_{min})$	$\Leftrightarrow x_{min} = x_{max} = y_{min} = y_{max}$
$[x_{min}, x_{max}] < [y_{min}, y_{max}]$	$\Leftrightarrow x_{min} < y_{max}$	$\Leftrightarrow x_{max} < y_{min}$
$[x_{min}, x_{max}] \leq [y_{min}, y_{max}]$	$\Leftrightarrow x_{min} \leq y_{max}$	$\Leftrightarrow x_{max} \leq y_{min}$
$\neg E_1$	$\Leftrightarrow \neg Certain(E_1)$	$\Leftrightarrow \neg Possible(E_1)$
$E_1 \vee E_2$	$\Leftrightarrow Possible(E_1) \vee Possible(E_2)$	$\Leftarrow Certain(E_1) \vee Certain(E_2)$
$E_1 \wedge E_2$	$\Rightarrow Possible(E_1) \wedge Possible(E_2)$	$\Leftrightarrow Certain(E_1) \wedge Certain(E_2)$

Figure 8: Translation of range comparison expressions.

points along the range $[L_i, H_i]$). The following relationships translate the notation used in this paper into the notation from [Lip79]: $T^+ = \|T\|_*$, $T^? = \|T\|^* - \|T\|_*$, $T^- = \overline{\|T\|^*}$.

In general, the selection predicate does not influence the evaluation of the aggregate—as we have seen in Section 6, the only information needed from the selection predicate is the classification of tuples into T^+, T^-, and $T^?$. However, a slight refinement can be made if the selection predicate is over the same column as the aggregation.[4] In this special case, each tuple t_i in $T^?$ has a restriction on actual value V_i imposed by the selection predicate, in addition to the bound $[L_i, H_i]$. For example, bound $[L_i, H_i] = [3, 8]$ has an additional restriction on V_i under the predicate < 5, if V_i is to contribute to the result. To take advantage of this additional restriction, the bounds $[L_i, H_i]$ for tuples in $T^?$ can be shrunk before they are input to the result computation or CHOOSE_REFRESH algorithm. For example, if we are aggregating *latency* under the predicate $latency > 10$, we can modify any lower bounds below 10 to 10 by using $[\max(L_i, 10), H_i]$ instead of $[L_i, H_i]$.

E Computing a Tight Bound for AVG with a Selection Predicate

To compute a tight bound for AVG with a selection predicate (recall Section 6.4.1), proceed as follows. First, let $S_L = \sum_{t_i \in T^+} L_i$ and $K_L = |T^+|$, the sum and cardinality of the low values in T^+. Then, let A represent the tuples $t_i \in T^?$, sorted in increasing order by L_i. Let a be the first element of A. If $L_a < \frac{S_L}{K_L}$, then add L_a to S_L and 1 to K_L. Advance a and continue this process until $L_a \geq \frac{S_L}{K_L}$. Similarly, let $S_H = \sum_{t_i \in T^+} H_i$ and $K_H = |T^+|$. Now, let A represent the tuples $t_i \in T^?$, sorted in decreasing order by H_i. Let a be the first element of A. If $H_a > \frac{S_H}{K_H}$, then add H_a to S_H and 1 to K_H. Advance a and continue this process until $H_a \leq \frac{S_H}{K_H}$. Finally, let:

$$[L_A, H_A] = [\frac{S_L}{K_L}, \frac{S_H}{K_H}]$$

For example, consider query *Q6* from Section 1.1 that asks for the average *latency* for links having $traffic > 100$. First, we classify tuples into T^-, $T^?$, and T^+ as shown in Figure 7. Since T^+ = {2, 4}, initially $S_L = 14$ and $K_L = 2$. A is [1, 6, 5, 3], which are the tuples in $T^?$ sorted in increasing order by

[4]More generally, the refinement applies if the selection predicate always restricts the value of the aggregation column. For example, the predicate $T.a < 5 \wedge T.b \neq 2$ always restricts the value of column $T.a$ to be less than 5.

L_i. First, we let a = 1, and since $L_a = 2 < \frac{S_L}{K_L} = \frac{14}{2} = 7$, we set $S_L = S_L + L_a = 14 + 2 = 16$ and $K_L = K_L + 1 = 2 + 1 = 3$. Then, we let a = 6, and since $L_a = 4 < \frac{S_L}{K_L} = \frac{16}{3} = 5.3$, we set $S_L = S_L + L_a = 16 + 4 = 20$ and $K_L = K_L + 1 = 3 + 1 = 4$. Next, we let a = 5, and note that $L_a = 8 \geq \frac{S_L}{K_L} = \frac{20}{4} = 5$, so we stop with $S_L = 20$ and $K_L = 4$. The computation of S_H and K_H proceeds similarly to yield $S_H = 34$ and $K_H = 3$. These results give a bounded AVG of $[\frac{S_L}{K_L}, \frac{S_H}{K_H}] = [\frac{20}{4}, \frac{34}{3}] = [5, 11.3]$. This algorithm for computing a tight bound for AVG has a running time of $O(n \cdot \log n)$.

F Choosing Refresh Tuples for AVG with a Selection Predicate

Refer to Section 6.4.2 and note that we make use of the loose bound for AVG given in Section 6.4.1. CHOOSE_REFRESH$_{\text{AVG}}$ will guarantee that the precision constraint $H_A - L_A \leq R$ is satisfied with:

$$[L_A, H_A] = [\min(\frac{L_{SUM}}{H_{COUNT}}, \frac{L_{SUM}}{L_{COUNT}}), \max(\frac{H_{SUM}}{L_{COUNT}}, \frac{H_{SUM}}{H_{COUNT}})]$$

Although it would be desirable to find a CHOOSE_REFRESH algorithm that guarantees the precision constraint is satisfied for the exact bound $[L_A, H_A] = [\frac{S_L}{K_L}, \frac{S_H}{K_H}]$ described in Appendix E, we have not yet succeeded in finding such an algorithm.

CHOOSE_REFRESH$_{\text{AVG}}$ chooses a set of tuples T_R such that after refreshing the tuples in T_R and computing $[L_{SUM}, H_{SUM}]$ and $[L_{COUNT}, H_{COUNT}]$, $\Delta AVG = H_{AVG} - L_{AVG} = \max(\frac{H_{SUM}}{L_{COUNT}}, \frac{H_{SUM}}{H_{COUNT}}) - \min(\frac{L_{SUM}}{H_{COUNT}}, \frac{L_{SUM}}{L_{COUNT}}) \leq R$. To make it possible to choose bounds for SUM and COUNT that will guarantee $\Delta AVG \leq R$, we must formulate ΔAVG as a function of ΔSUM = $H_{SUM} - L_{SUM}$ and $\Delta COUNT$ = $H_{COUNT} - L_{COUNT}$. Based on this function, CHOOSE_REFRESH$_{\text{AVG}}$ chooses an "approximately optimal" set of tuples T_R to refresh that gives values for ΔSUM and $\Delta COUNT$ such that the precision constraint $\Delta AVG \leq R$ is guaranteed to be met.

The relationship between $[L_{SUM}, H_{SUM}]$, $[L_{COUNT}, H_{COUNT}]$, and ΔAVG is:

$$\Delta AVG \leq RHS = \frac{\Delta SUM + (\frac{\max(H_{SUM}, -L_{SUM}, H_{SUM} - L_{SUM})}{L_{COUNT}}) \cdot \Delta COUNT}{\Delta COUNT + L_{COUNT}}$$

To show this inequality, we consider three cases. In case 1, if $L_{SUM} \geq 0$, $\Delta AVG \leq \frac{H_{SUM}}{L_{COUNT}} - \frac{L_{SUM}}{H_{COUNT}}$, which gives:

$$\Delta AVG \leq RHS_1 = \frac{\Delta SUM + (\frac{H_{SUM}}{L_{COUNT}}) \cdot \Delta COUNT}{\Delta COUNT + L_{COUNT}}$$

In case 2, if $H_{SUM} \leq 0$, $\Delta AVG \leq \frac{H_{SUM}}{H_{COUNT}} - \frac{L_{SUM}}{L_{COUNT}}$, which gives:

$$\Delta AVG \leq RHS_2 = \frac{\Delta SUM + (\frac{-L_{SUM}}{L_{COUNT}}) \cdot \Delta COUNT}{\Delta COUNT + L_{COUNT}}$$

Otherwise, in case 3 ($L_{SUM} < 0$ and $H_{SUM} > 0$), $\Delta AVG \leq \frac{H_{SUM}}{L_{COUNT}} - \frac{L_{SUM}}{L_{COUNT}}$, which gives:

$$\Delta AVG \leq RHS_3 = \frac{\Delta SUM + (\frac{H_{SUM} - L_{SUM}}{L_{COUNT}}) \cdot \Delta COUNT}{\Delta COUNT + L_{COUNT}}$$

All three cases are equivalent to $RHS = \frac{\Delta SUM + (\frac{\max(H_{SUM}, -L_{SUM}, H_{SUM}-L_{SUM})}{L_{COUNT}}) \cdot \Delta COUNT}{\Delta COUNT + L_{COUNT}}$. In case 1, $RHS_1 = RHS$ since $L_{SUM} \geq 0$ implies $\max(H_{SUM}, -L_{SUM}, H_{SUM} - L_{SUM}) = H_{SUM}$. Similarly, in case 2, $RHS_2 = RHS$ since $H_{SUM} \leq 0$ implies $\max(H_{SUM}, -L_{SUM}, H_{SUM} - L_{SUM}) = -L_{SUM}$. In case 3, $RHS_3 = RHS$ since the SUM bound straddles 0, which implies $\max(H_{SUM}, -L_{SUM}, H_{SUM} - L_{SUM}) = H_{SUM} - L_{SUM}$.

Since our goal is to express ΔAVG as a function of ΔSUM and $\Delta COUNT$, we must eliminate all other values from the relationship:

$$\Delta AVG \leq \frac{\Delta SUM + (\frac{\max(H_{SUM}, -L_{SUM}, H_{SUM}-L_{SUM})}{L_{COUNT}}) \cdot \Delta COUNT}{\Delta COUNT + L_{COUNT}}$$

To do this elimination, we substitute conservative estimates for the values L_{SUM}, H_{SUM}, and L_{COUNT}. Conservative estimates for these values are obtained by computing SUM and COUNT over the current cached bounds as $[L'_{SUM}, H'_{SUM}]$ and $[L'_{COUNT}, H'_{COUNT}]$. Since, when the refreshes are performed, these bounds can shrink but not grow, $L'_{SUM} \leq L_{SUM}$, $H'_{SUM} \geq H_{SUM}$, and $L'_{COUNT} \leq L_{COUNT}$. Therefore, by examining the inequality relating $[L_{SUM}, H_{SUM}]$, $[L_{COUNT}, H_{COUNT}]$, and ΔAVG, it can be seen that substituting L'_{SUM} for L_{SUM}, H'_{SUM} for H_{SUM}, and L'_{COUNT} for L_{COUNT} makes the right-hand side strictly larger, so it is still an upper bound on ΔAVG. This substitution results in:

$$\Delta AVG \leq \mathcal{F}(\Delta SUM, \Delta COUNT) = \frac{\Delta SUM + (\frac{\max(H'_{SUM}, -L'_{SUM}, H'_{SUM}-L'_{SUM})}{L'_{COUNT}}) \cdot \Delta COUNT}{\Delta COUNT + L'_{COUNT}}$$

Now that we finally have $\mathcal{F}(\Delta SUM, \Delta COUNT)$, an upper bound for ΔAVG as a function of ΔSUM and $\Delta COUNT$ (since L'_{SUM}, H'_{SUM}, and L'_{COUNT} are computed once and used as constants), we can substitute this function for ΔAVG in the precision constraint. Recall that the precision constraint requires that $\Delta AVG \leq R$. Substituting $\mathcal{F}(\Delta SUM, \Delta COUNT)$ for ΔAVG gives $\mathcal{F}(\Delta SUM, \Delta COUNT) \leq R$.

At this point, we have formulated the precision constraint in terms of only ΔSUM and $\Delta COUNT$. Rewriting the precision constraint in terms of ΔSUM gives:

$$\Delta SUM \leq L'_{COUNT} \cdot R - (\frac{\max(H'_{SUM}, -L'_{SUM}, H'_{SUM} - L'_{SUM})}{L'_{COUNT}} - R) \cdot \Delta COUNT$$

This formulation of the precision constraint can be used in place of the original constraint $\Delta AVG \leq R$. Therefore, the CHOOSE_REFRESH$_{\mathrm{AVG}}$ algorithm is free to choose any values for ΔSUM and $\Delta COUNT$ that satisfy the reformulated precision constraint. We have thus reduced the task of choosing refresh tuples for AVG to the task of choosing refresh tuples for SUM under this reformulated constraint.

Normally, to choose refresh tuples for SUM, we have the constraint $\Delta SUM \leq R_{SUM}$. In this case, we instead have the constraint $\Delta SUM \leq L'_{COUNT} \cdot R - (\frac{\max(H'_{SUM}, -L'_{SUM}, H'_{SUM}-L'_{SUM})}{L'_{COUNT}} - R) \cdot \Delta COUNT$, so we let R_{SUM} be the following function of $\Delta COUNT$:

$$R_{SUM}(\Delta COUNT) = L'_{COUNT} \cdot R - (\frac{\max(H'_{SUM}, -L'_{SUM}, H'_{SUM} - L'_{SUM})}{L'_{COUNT}} - R) \cdot \Delta COUNT$$

To see how to choose refresh tuples for SUM when R_{SUM} is a function of $\Delta COUNT$, first recall that the CHOOSE_REFRESH$_{\mathrm{SUM}}$ algorithm chooses refresh tuples for SUM by mapping it to the 0/1 Knapsack Problem, where the knapsack capacity $M = R_{SUM}$. Therefore, for the CHOOSE_REFRESH$_{\mathrm{AVG}}$

algorithm, we need to make the knapsack capacity a function of $\Delta COUNT$. On the surface, it looks as though this modification is not possible since there is no way to make the knapsack capacity a function instead of a constant. Fortunately, making the knapsack capacity a function of $\Delta COUNT$ is possible to fake. First, recall that $\Delta COUNT$ is equal to the number of tuples in $T^?$ that do not get refreshed (and thus remain in $T^?$ after the refreshes are performed). Also, recall that the set of items placed in the knapsack corresponds to $\overline{T_R}$: the set of tuples that will not be refreshed. It follows that $\Delta COUNT$ is equal to the number of $T^?$ tuples in the knapsack. Therefore, when the knapsack is empty, $\Delta COUNT = 0$ and thus the initial knapsack capacity $M = R_{SUM}(0) = L'_{COUNT} \cdot R$. Furthermore, every time a $T^?$ tuple is added to the knapsack, $\Delta COUNT$ increases by 1. Since the function $R_{SUM}(\Delta COUNT)$ is a line with (negative) slope:

$$m = -(\frac{\max(H'_{SUM}, -L'_{SUM}, H'_{SUM} - L'_{SUM})}{L'_{COUNT}} - R)$$

the capacity of the knapsack decreases by the amount $-m$ every time a $T^?$ tuple is added to the knapsack. Observe that decreasing the knapsack capacity when an item is added is equivalent to increasing the weight of the item. Therefore, to simulate shrinking the knapsack by $-m$ every time $\Delta COUNT$ increases by 1, all we have to do is add the quantity $-m$ to the weight of each tuple in $T^?$.

To summarize, the CHOOSE_REFRESH$_{\mathrm{AVG}}$ algorithm is exactly the same as the CHOOSE_REFRESH$_{\mathrm{SUM}}$ algorithm (which maps to the 0/1 Knapsack Problem) with the following modifications: $M = L'_{COUNT} \cdot R$, and for all tuples $t_i \in T^?$, $W_i = W_i + (\frac{\max(H'_{SUM}, -L'_{SUM}, H'_{SUM} - L'_{SUM})}{L'_{COUNT}} - R)$. The values L'_{SUM}, H'_{SUM}, and L'_{COUNT} are found by computing SUM and COUNT over the current cached bounds as $[L'_{SUM}, H'_{SUM}]$ and $[L'_{COUNT}, H'_{COUNT}]$. The running time of CHOOSE_REFRESH$_{\mathrm{AVG}}$ is dominated by the running time of CHOOSE_REFRESH$_{\mathrm{SUM}}$, which is given in Section 6.2.

F.1 Revisiting the Example of Section 6.4.2

Consider query *Q6* that asks for the average *latency* for links having *traffic* > 100, with $R = 2$. First, we classify tuples into T^-, $T^?$, and T^+, as shown in Figure 7. Then, we compute $[L'_{SUM}, H'_{SUM}]$ = [14, 55] and $[L'_{COUNT}, H'_{COUNT}]$ = [2, 6]. We use these values to assign a weight to each tuple by computing the weight used in the CHOOSE_REFRESH$_{\mathrm{SUM}}$ algorithm, and for tuples in $T^?$, adding $\frac{\max(H'_{SUM}, -L'_{SUM}, H'_{SUM} - L'_{SUM})}{L'_{COUNT}} - R = \frac{55}{2} - 2 = 25.5$. The column labeled W'' in Figure 2 shows these weights. Using the Knapsack Problem with M = $L'_{COUNT} \cdot R = 2 \cdot 2 = 4$, the knapsack optimally "contains" tuples 2 and 4. After refreshing the other tuples T_R = $\{1, 3, 5, 6\}$, the bounded AVG is [8, 9].