AQUA: System and Techniques for Approximate Query Answering

Phillip B. Gibbons, Viswanath Poosala, Swarup Acharya, Yair Bartal, Yossi Matias,* S. Muthukrishnan, Sridhar Ramaswamy, Torsten Suel

Information Sciences Research Center
Bell Laboratories
600 Mountain Avenue
Murray Hill NJ 07974

February 24, 1998

Abstract

In large data recording and warehousing environments, it is often advantageous to provide fast, approximate answers to queries. The goal is to provide an estimated response in orders of magnitude less time than the time to compute an exact answer, by avoiding or minimizing the number of accesses to the base data.

This paper presents the Approximate QUery Answering (AQUA) System, for fast, highly-accurate approximate answers to queries. Aqua provides approximate answers using small, pre-computed synopses (samples, counts, etc.) of the underlying base data. An important feature of Aqua is that it provides accuracy guarantees without any a priori assumptions on either the data distribution, the order in which the base data is loaded, or the layout of the data on the disks. Currently, the system provides fast approximate answers for queries with selects, aggregates, group bys and/or joins (especially, the multi-way foreign key joins that are popular in OLAP).

We present several new techniques for improving the accuracy of approximate query answers for this class of queries. We show how join sampling can significantly improve the approximation quality. We describe how biased sampling can be used to overcome the problem of group size disparities in group by operations. Moreover, we present efficient algorithms for incremental maintenance of join samples, biased samples, and all other synopses used in the current Aqua system. Analytical bounds and experimental results on TPC-D queries demonstrate Aqua's effectiveness, even in the presence of data distribution changes.

Aqua is the first system to provide fast (no accesses to the base data at query time), highly-accurate approximate answers for a broad class of queries that arise in data warehousing scenarios.

Finally, the area of approximate query answers is not well-understood (e.g., what is a concise approximate answer for a set-valued query?). This paper attempts to clarify issues related to approximate query answers by presenting a taxonomy for (metrics for evaluating) approximate query engines and the approximate answers they provide.

*Current address: Tel-Aviv University, Ramat Aviv, Tel-Aviv 69978 Israel.

1 Introduction

Traditional query processing has focused solely on providing exact answers to queries, in a manner that seeks to minimize response time and maximize throughput. However, there are a number of environments for which the response time for an exact answer is often slower than is desirable. First, in large data recording and warehousing environments, providing an exact answer to a complex query can take minutes to hours, due to the amount of disk I/O required. For environments with terabytes or more of data, even a single scan of the data can take tens of minutes. Second, in distributed data recording and warehousing environments, some of the data may be remote, resulting in slow response times, and may even be currently unavailable, so that an exact answer is not an option until the data again becomes available [FJS97]. Finally, in environments with stringent response time requirements, even a single access at a particular level of the storage hierarchy may be unacceptably slow, e.g., for sub-millisecond response time, a single disk access is too slow.

Environments for which providing an exact answer results in undesirable response times motivate the study of techniques for providing *approximate* answers to queries. The goal is to provide an estimated response in orders of magnitude less time than the time to compute an exact answer, by avoiding or minimizing the number of accesses to the base data.

There are a number of scenarios for which an exact answer may not be required, and a user may prefer a fast, approximate answer. For example, during a drill-down query sequence in ad-hoc data mining, the earlier queries in the sequence are used solely to determine what the interesting queries are [GM95, HHW97]. An approximate answer can also provide feedback on how well-posed a query is. Moreover, it can provide a tentative answer to a query when the base data is unavailable. Another example is when the query requests numerical answers, and the full precision of the exact answer is not needed, e.g., a total, average, or percentage for which only the first few digits of precision are of interest (such as the leading few digits of a total in the millions, or the nearest percentile of a percentage). Finally, note that techniques for fast approximate answers can also be used in a more traditional role within the query optimizer to estimate plan costs; such an application demands very fast response times but not exact answers.

Despite some recent work in approximate query answers (e.g., [VL93, GM95, HHW97, BDF$^+$97, GM98]), the state-of-the-art is quite limited in its speed, scope and accuracy.

This paper describes the **Approximate QUery Answering (AQUA)** System, the first system designed to provide fast, highly-accurate approximate answers to a broad class of aggregate and set-valued queries. Aqua provides approximate answers in orders of magnitude less response time than previous systems (e.g., [VL93, HHW97]), by (typically) avoiding disk accesses at query time. Another important feature of Aqua is that it provides accuracy guarantees without any a priori assumptions on either the data distribution, the order in which the base data is loaded, or the physical layout of the data on the disks. Currently, the system provides fast approximate answers for queries with selects, aggregates, group bys and/or joins (especially, the multi-way foreign key joins that are popular in OLAP).

This paper presents several new techniques for improving the accuracy of approximate query answers for this class of queries. We show how join sampling overcomes a problem with joins and

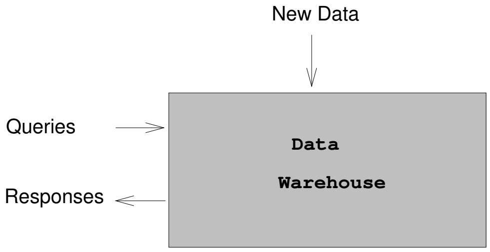


Figure 1: A traditional data warehouse.

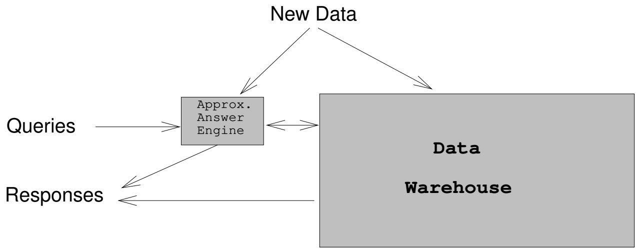


Figure 2: Data warehouse set-up for providing approximate query answers.

can significantly improve the approximation quality. We describe how biased sampling can be used to deal with group size disparities in group by operations. Moreover, we present efficient algorithms for incremental maintenance of join samples, biased samples, and all other synopses used in the current Aqua system.

Experimental results on queries from the TPC-D benchmark demonstrate the effectiveness of Aqua in providing highly-accurate answers without accessing the base data at query time, and in maintaining this effectiveness in the presence of data distribution changes. In addition, we present both analytical bounds and experimental results comparing various strategies for allocating sample sizes.

The area of approximate query answering is not well-understood. E.g., what is a (concise) approximate answer for a set-valued query? How do we evaluate and compare approximate query systems? This paper attempts to address some of these issues by presenting a general framework for approximate query answering and a taxonomy for (metrics for evaluating) approximate query engines and the approximate answers they provide. This discussion generalizes the one in [GM98] from simple aggregates to group by and set-valued queries.

Road map. Section 2 presents our framework/taxonomy/metrics. Section 3 describes the Aqua system design. Section 4 describes the problem with joins and our solution. Section 5 presents analytical bounds based on the allocation of sample sizes. Section 6 discusses our technique for group bys. Section 7 presents the experimental set-up and results obtained. Section 8 discusses related work. A summary of the Hoeffding-based confidence bounds we use appear in the appendix.

2 A framework for approximate query answering

Figure 1 depicts a traditional data warehouse set-up, in which the base data resides in a data warehouse that is updated as new data arrives, and each query is answered exactly using the data warehouse. In contrast, Figure 2 depicts a set-up for approximate query answering, which includes an approximate query engine in addition to the data warehouse. To facilitate in answering queries, the approximate query engine can store various summary information on the data, which we denote *synopsis data structures* or *synopses*. Examples of synopses for a relational data warehouse include histograms and sample rows of large relations and all the rows of small relations, projected on the columns of interest. These synopses can be maintained by: (1) observing the new data as it is loaded into the data warehouse, (2) periodically returning to the data warehouse to update the information, and/or

The exact answer			
region	**type**	**avg. sales**	**min. sales**
eastern	retail	12435	4035
eastern	outlet	7389	1227
central	retail	14837	3928
western	retail	16726	4399
western	outlet	8874	389

An approximate answer			
region	**type**	**avg. sales**	**min. sales**
eastern	retail	12000 ± 800	4100 ± 400
eastern	outlet	7200 ± 800	1200 ± 400
central	retail	14500 ± 800	3800 ± 400
central	outlet	< 500	< 500
western	retail	17000 ± 800	4100 ± 400
western	outlet	8900 ± 800	< 500

Figure 3: An example approximate answer for a group by query.

(3) returning to the data warehouse at query time.

Queries are sent to the approximate answer engine. Whenever possible, the engine uses its synopses to promptly return a query response, consisting of an approximate answer and an accuracy measure (e.g., a 95% confidence interval for numerical answers). In *continuous reporting*[1], the engine proceeds to provide a series of ⟨approximate answer, accuracy measure⟩ pairs for the query, with each subsequent pair providing a more accurate answer (e.g., [HHW97]). In *discrete reporting*, only one or a few such pairs are provided by the engine. The user posing the query can decide whether to abort the query processing and be content with the current approximate answer or to proceed to the next approximation or to an exact answer from the base data. Alternatively, the user may pose his/her next query, while allowing the current query to proceed, in order to allow for subsequent verification of the approximate answer.

2.1 What is an approximate answer?

For queries whose answer is an aggregate value (e.g., the result of AVG, SUM, COUNT), the notion of an approximate answer is an intuitive one: it is simply an estimated value for the answer and an accuracy measure. This can be extended to a collection of aggregate values, such as arises with an SQL *group by* operation: an approximate answer is an ⟨estimated value, accuracy measure⟩ pair for each such aggregate value, labeled with the attributes that define the aggregate (the group). An example is given in Figure 3. In this example, the approximate answer provides accuracy measures as confidence intervals for each estimate, for some confidence probability that would also be specified (e.g., 95% confidence intervals). Note that in several cases an upper bound, denoted a *sanity bound*, is provided instead of an estimate. Finally, note that an approximate answer can include rows not in the exact answer, as in this example (for the *central outlet* group), and vice-versa.

For set-valued queries, it is less intuitive what an approximate answer should be. Since the number of tuples in the exact answer may be quite large, we often do not want to return a tuple for each tuple in the exact answer. In order to ensure very fast response times, we seek to return only a small number of representative tuples, together with meta-information on the entire set of tuples. Thus an approximate answer consists of both estimates on meta-information for the exact answer, including

[1] Denoted "progressive resolution refinement" in [BDF+97].

an estimated (or actual) count of the number of tuples in the exact answer, and representative tuples from the exact answer. Each meta-information estimate includes an accuracy measure. Representative tuples can be classified as *certain* or *possible*, depending on whether or not the approximate engine is certain that the tuple is in the exact answer [VL93]. Possible tuples are reported along with some measure of their similarity to tuples in the exact answer. Examples include tuples that are in the exact answer with a given confidence probability or tuples that may not meet a selection criterion (such as a min or a max) that is computed by the query, but are close to it.

We classify the certain tuples as *randomly-selected* if the tuples reported are a uniform random sample of the set of output tuples, as *biased-selected* if the tuples reported are biased according to a specific criterion, or as *arbitrary*. Randomly-selected tuples have the advantage that they are uniformly representative of the entire set of output tuples. Biased-selected tuples have the advantage if the bias criterion is in line with the "most interesting" output tuples, e.g., the query requests tuples that lie above a certain threshold and the reported tuples are biased towards those that exceed the threshold by the largest amount, as in [GM98]. In such cases, biased-selected may be preferred to randomly-selected. On the other hand, if the criterion for what makes an output tuple interesting is not known, or there are conflicting criteria, then a uniform random sample is a natural choice. Representative tuples may or may not contain all the columns in the full tuple.

There are a number of possible accuracy measures for an approximate answer, depending on the type of query. For numerical answers, a natural accuracy measure is a confidence interval, consisting of an accuracy interval $[a, b]$ and a confidence probability p. The confidence interval asserts that the exact value is between a and b with probability at least p. It is also useful to have the approximate answer be an *unbiased* estimator of the exact value, that is, the expected value of the approximate answer is equal to the exact value. Accuracy measures and similarity measures can be classified as either (provably) *guaranteed* or *heuristic*. Common heuristic measures include those based on assumptions on the distribution of the values within a histogram bucket, on the independence of attributes, on the uniformity of joins, and on the randomness of tuples read sequentially from disk. Although guaranteed measures are preferred, in some cases it is difficult to obtain tight guaranteed bounds, and heuristic measures may be more suitable.

Table 1 summarizes the requirements for approximate answers.

2.2 Metrics for evaluating approximate query engines

Approximate query engines can be evaluated according to the following five metrics:

- *coverage*: the range of queries for which approximate answers can be provided.
- *response time*: the time to provide an approximate answer for a query.
- *accuracy*: the accuracy of the answers provided, and the confidence in that accuracy.
- *update time*: the overheads in keeping its synopses up-to-date.
- *footprint*: the storage requirements for its synopses.

Table 1: Approximate answers.

Exact answer	Approximate answer
aggregate values	for each aggregate value: 1. estimated value or sanity bound 2. accuracy measure
set of tuples	1. estimated meta-information on the output, i.e., a collection of $\langle$value, accuracy measure$\rangle$ pairs. 2. representative tuples, either: a. randomly-selected certain b. biased-selected certain, with criterion c. arbitrary certain, or d. possible, with similarity measure

Typically, there will be trade-offs among these metrics, e.g., with continuous reporting, the additional response time for each subsequent answer results in greater accuracy.

3 The Aqua system

The goal of Aqua is to provide highly-accurate answers with minimal response time by:

- *Maintaining a number of synopses on the data.*
- *Updating these synopses primarily by observing the new data as it is loaded into the data warehouse.* We assume that Aqua is run primarily in the context of a data warehousing system in which updates are applied in a "batch" mode. Aqua seeks to minimize disk accesses at update time by observing new data during these batch updates and minimizing accesses to the old data.
- *Providing discrete reporting.* The current Aqua system provides for a single approximate answer, determined by all the synopses at hand. This simplifies the maintenance and use of the synopses, resulting in faster response times and update times. Continuous reporting would require a means for reporting answers with increasing accuracy, resulting in greater complexity and either a sufficiently large footprint to support the highest level of accuracy or a very slow response time if the base data must be heavily accessed.
- *Ensuring guaranteed accuracy measures.* Although heuristic accuracy measures for estimation procedures are common in commercial databases, they are not quite satisfying. This is particularly true for approximate query answers, since the approximate answer is reported to the user. Aqua seeks to push from heuristic confidence to guaranteed confidence.
- *Having a footprint orders of magnitude smaller than the data warehouse.* Aqua seeks to have memory-resident any synopsis that is frequently updated and/or frequently used to respond to

queries, in order to minimize update and response times.[2]

In response to a query, Aqua uses its synopses to produce an approximate answer according to Table 1. For aggregate values, the accuracy measure is a confidence interval. For set-valued queries, the meta-information is an estimate and confidence interval for the size of the exact answer and representative tuples are of type (a), (b) or (d), depending on the query.

3.1 Query processing in Aqua

In this section, we briefly describe the internals of the query processing engine that we developed for Aqua. Our design is motivated by the design principles of the Volcano query processing system [Gra94]. The key features of the Aqua query processor are (a) its rich set of query operators and (b) its easy extensibility. The input to the system is a query plan containing a tree of operators. Operators correspond to distinct query operations, e.g., select, hash or nested loop joins, sort, aggregate, read-from-file, etc. All operators are implemented as *iterators* with a standard interface and executed in a top-down fashion. First, we invoke the *open* call on the root of the plan, which initializes the operator-specific data and calls *open* on each of its children recursively. Next, we repeatedly invoke the *fetch* call on the root. In response to this call, an operator fetches some of its input from its children (or from a database file in case of the *file read* operator), performs the relevant operation (if any), and sends the results upwards. This process terminates when all the inputs to the query are exhausted and no more results can be generated. Finally, we recursively invoke *close* on the operators which performs clean-up operations, e.g., close the open tables, release memory.

An important feature of this design is the isolation of operators from each other, i.e., an operator does not need to know the nature of the operators generating its input and vice versa. For example, in the operator's view, the input could be coming from a simple file scan or from a complex query (these are called *anonymous inputs* in Volcano [Gra94]). This feature enables us to build arbitrarily complex queries in a modular fashion and add (or modify) an existing query operator with localized changes. This is very useful in Aqua because it enables us to implement various novel operators needed in our research very easily. For example, we implemented a *sample* operator, which samples its input stream and outputs randomly chosen tuples, either a fixed number (using reservoir sampling [Vit85]) or a desired fraction of the input stream. Clearly, this is very beneficial in the study of sampling techniques in Aqua.

The query processing engine is used for producing both approximate answers and exact answers.

3.2 Aqua synopses and their maintenance

The basic Aqua query processing engine is augmented with routines to maintain a number of synopses on the data, many of which are normally stored in system catalogs, including:

- For each relation, we maintain a count of the number of tuples in the relation.
- For small relations (hundreds of tuples or less), we store all the tuples in the relation.

[2] For persistence and recovery, combinations of snapshots and/or logs can be stored on disk; alternatively, the synopsis can often be recomputed in one pass over the base data.

- For all other relations, we store random tuples and various other synopses, as discussed in subsequent sections of the paper.
- For each attribute that may be used in an AVG or SUM aggregate, we maintain an upper bound and a lower bound on its range.

For each tuple stored, we retain only attributes of interest. The optimal selection of attributes to retain depends on the mix of queries. In our experiments, we discard descriptive strings such as comments. This reduces the footprint needed for each tuple, since descriptive strings often require many bytes. On the other hand, this choice implies that the system can not provide a reasonable approximate answer for queries on these attributes.

We have developed algorithms for incrementally maintaining the synopses used in Aqua, based on the batch arrival of new data and an occasional access to the (stored) base data. These enable synopses to be kept effectively up-to-date at all times without any concurrency bottleneck. In an online environment in which updates and queries intermix, Aqua can not afford to maintain up-to-date synopses that require examining every tuple, such as the minimum and maximum value of an attribute, without creating a concurrency bottleneck.[3] In such environments, maintenance is performed only periodically. Approximate answers depending on synopses that require examining every tuple would not take into account the most recent trends in the data (i.e., those occurring since maintenance was last performed), and hence could greatly decrease the accuracy guarantees. Note that the incremental maintenance algorithms can be used to compute all synopses from scratch, in one scan of the base data followed by indexed look-ups on a small fraction of the keys, should such a recomputation be necessary.

Most of the synopses mentioned above can be maintained using known techniques. Counters are maintained by incrementing them as tuples are inserted and decrementing them as tuples are deleted. Uniform random samples are maintained as tuples are inserted and deleted using the algorithm from [GMP97b]. Maximum and minimum values for attributes are maintained under insertions by comparing the new tuple with the current maximum or minimum. Under deletions, if the maximum or minimum is deleted, we can either (1) ignore the deletion, resulting in a conservative bound, (2) revisit the relation to extract the new maximum or minimum, (3) maintain a set of the largest and smallest values, and only perform (1) or (2) if the entire set is deleted, or (4) maintain a histogram on the number of values within each range, where the ranges could be, e.g., powers of two. This would provide estimates on the maximum and minimum within a factor of 2 using only a logarithmic number of buckets, and without resorting to (2).

4 The problem with joins and a solution

A natural set of synopses for an approximate query engine would include a uniform random sample of each base relation. However, the problem with using samples of base relations to provide approximate

[3] Note that most of the synopses in Aqua are sampling-based, and hence require only infrequent updates (see Section 7.4).

answers for queries with joins is that, in general, the quality of the approximation suffers greatly from even a single join, for two reasons:

1. The join of two uniform random samples is not a uniform random sample of the output of the join. Except in the special case where for both relations, each tuple joins with at most one tuple in the other relation, the join operator results in dependencies among join tuples [GGMS96].

2. The join of two random samples is typically a small number of tuples, even when the join selectivity is fairly high. For example, if the majority of the tuples in one relation *each* joined with a fixed set S of tuples comprising a tiny fraction of the tuples in the other relation, then with high probability, *none* of these tuples will be in the join of the samples of the relations since the tuples in S will not appear in the sample with high probability.

Indeed the best known confidence interval bounds for such approximations are quite pessimistic [Haa98] (see Section A). For example, it follows from the bounds there that when the join-size is not large, as is frequently the case, the sample size must be at least quadratic in the maximum value of the join attribute, or a sizeable fraction of the relations, before we get any nontrivial confidence interval! Note that this problem arises even with *foreign key joins*[4] (see Section 7.3).

We have developed a solution, called *join samples*, that works well for any acyclic data warehouse schema with only foreign key joins. Such schema are common in data warehouses [Sch97], and indeed the TCP-D benchmark reflects this scenario with its schema. We first describe a solution that maintains entire tuples from the output of various joins. Later, we will reduce the space needed by storing only attributes of interest and by removing redundant sub-tuples. The basic idea is to leverage Lemma 4.1 and Lemma 4.2 below by maintaining one join sample for each base relation.

Consider a directed acyclic graph, G, with a vertex for each base relation and a directed edge from a vertex u to a vertex $v \neq u$ if there are one or more attributes in (the relation corresponding to) u that form a foreign key for (the relation corresponding to) v. The edge is labeled with the foreign key. An example is given in Figure 4 for the TPC-D benchmark.

Lemma 4.1 *The subgraph of G on the k nodes in any k-way foreign key join must be a connected subgraph with a single root node.*

Proof. Consider an ordering $r_1, \ldots, r_k$ on the relations that satisfies the k-way foreign key join property given above. The proof is by induction, with the base case of a single node r_1. Let $1 < i \leq k$ and $s_{i-1} = r_1 \bowtie \cdots \bowtie r_{i-1}$. Assume that the subgraph G_{i-1} on the $i-1$ nodes in s_{i-1} is connected with a single root node r_1. Since $s_{i-1} \bowtie r_i$ is a 2-way foreign key join, the join attribute must be a key in r_i. Thus there is an edge directed from some node in G_{i-1} to r_i, implying that $G_i = G_{i-1} \cup r_i$ is a connected subgraph of G. Hence there is a directed path in G from r_1 to r_i. Since G is acyclic, $r_i \neq r_1$, so r_1, which by the inductive assumption is the only root node in G_{i-1}, is the only root node of G_i. The lemma follows by induction. ∎

[4] A 2-way join $r_1 \bowtie r_2$, $r_1 \neq r_2$, is a foreign key join if the join attribute is a foreign key in r_1 (i.e., a key in r_2). For $k \geq 3$, a k-way join is a foreign key join if there is an ordering $r_1, r_2, \ldots, r_k$ of the relations being joined such that for $i = 2, 3, \ldots, k$, $s_{i-1} \bowtie r_i$ is a 2-way foreign key join, where s_{i-1} is the relation obtained by joining $r_1, r_2, \ldots, r_{i-1}$.

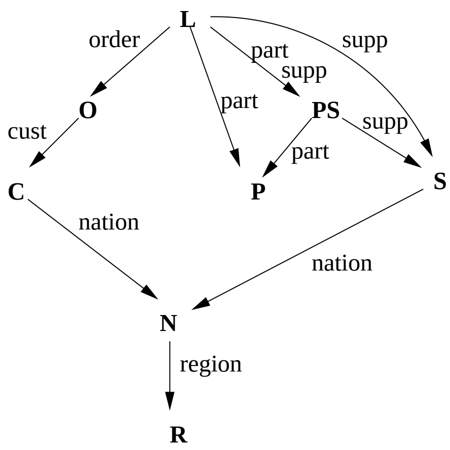


Figure 4: Directed graph for the TPC-D schema.

We denote the relation corresponding to the root node as the *source* relation for the k-way foreign key join.

Lemma 4.2 *There is a 1-1 correspondence between tuples in r_1 and tuples in any k-way foreign key join with source relation r_1.*

Proof. By the definition of a join, for each tuple τ in the output of a join, there exists a tuple τ' in r_1 such that τ projected on the attributes in r_1 is τ'. Conversely, we claim that for each tuple τ' in r_1 there is exactly one tuple τ in the k-way foreign key join. The claim is shown by induction. Consider an ordering $r_1, \ldots, r_k$ on the relations that satisfies the k-way foreign key join property given above. The claim trivially holds for the base case of a single relation r_1. Let $1 < i \leq k$ and $s_{i-1} = r_1 \bowtie \cdots \bowtie r_{i-1}$. Assume inductively that for each tuple τ' in r_1 there is exactly one tuple τ in s_{i-1}. Since $s_{i-1} \bowtie r_i$ is a 2-way foreign key join, the join attribute must be a key in r_i. Thus there is at most one tuple in r_i joining with each tuple in s_{i-1}, and furthermore, due to foreign key integrity constraints, there is at least one such tuple. Hence, for each tuple τ' in r_1 there is exactly one tuple τ in $s_i = s_{i-1} \bowtie r_i$. The claim, and hence the lemma, follows by induction. ∎

From Lemma 4.1, we have that each node can be the source relation only for k-way foreign key joins involving its descendants in G. For each relation r, there is some *maximum foreign key join* with r as the source relation. For example, in Figure 4, $C \bowtie N \bowtie R$ is the maximum foreign key join with source relation C, and $L \bowtie O \bowtie C \bowtie N1 \bowtie R1 \bowtie PS \bowtie P \bowtie S \bowtie N2 \bowtie R2$ is the maximum foreign key join with source relation L.

Join samples. For each node u in G, corresponding to a relation r_1, we define $\mathcal{J}(u)$ to be the output of the maximum foreign key join $r_1 \bowtie r_2 \bowtie \cdots \bowtie r_\kappa$ with source r_1. (If u has no descendants in G, then $\kappa = 1$ and $\mathcal{J}(u) = r_1$.) Let S_u be a uniform random sample of r_1. We define a *join sample*, $\mathcal{J}(S_u)$, to be the output of $S_u \bowtie r_2 \bowtie \cdots \bowtie r_\kappa$. Our synopsis consists of $\mathcal{J}(S_u)$ for all u in G.

The utility of this synopsis can be observed by the following theorem, which is an immediate consequence of Lemma 4.2.

Theorem 4.3 *Let $r_1 \bowtie \cdots \bowtie r_k$, $k \geq 2$, be an arbitrary k-way foreign key join, with source relation r_1. Let u be the node in G corresponding to r_1, and let S_u be a uniform random sample of r_1. Let A be the set of attributes in $r_1, \ldots, r_k$.*

- *$\mathcal{J}(S_u)$ is a uniform random sample of $\mathcal{J}(u)$ of size $|S_u|$.*
- *$r_1 \bowtie \cdots \bowtie r_k = \pi_A \mathcal{J}(u)$, i.e., the projection of $\mathcal{J}(u)$ on the attributes in $r_1, \ldots, r_k$.*
- *$\pi_A \mathcal{J}(S_u)$ is a uniform random sample of $r_1 \bowtie \cdots \bowtie r_k$ ($= \pi_A \mathcal{J}(u)$) of size $|S_u|$.*

Thus we can extract from our synopsis a uniform random sample of the output of any k-way foreign key join, $k \geq 2$.

Two joins are *distinct* if they do not join the same set of relations. The next lemma shows that a single join sample can be used for a large number of distinct joins, especially for the star-like schemas popular for data warehouses.

Lemma 4.4 *From a single join sample for a node whose maximum foreign key join has κ relations, we can extract a uniform random sample of the output of between $\kappa - 1$ and $2^{\kappa-1} - 1$ distinct foreign key joins.*

Proof. The former case arises if all the descendants of the node form a line in G. The latter case arises if the node is the root of a star of all its descendants, as in a star schema. ■

Note that since Lemma 4.2 fails to apply in general for any relation other than the source relation, the joining tuples in any relation r other than the source relation will not in general be a uniform random sample of r. Thus distinct join samples are needed for each node.

A limitation of our solution of maintaining join samples is that for worst case schemas, the size of the maximum foreign key join can be exponential in the number of relations in the schema:

Lemma 4.5 *There exists foreign key schema with t relations such that the maximum foreign key join has $4 \cdot 2^{(t-1)/3} - 3$ relations.*

Proof. Consider a "coat hanger" H_i with root r_i. H_{i+1} has root r_{i+1} with two children l and r each of which join to r_i. It is easy to verify that the coat hanger H_i has $3i + 1$ nodes. Consider t relations which are the nodes of $H_{(t-1)/3}$ with edges between them depicting the foreign key relationships. Then it is easy to verify that the maximum foreign key join has $4 \cdot 2^{(t-1)/3} - 3$ relations. ■

In such cases, we can decide how much of the maximum foreign key join to materialize based on the joins actually arising in queries.

4.1 Maintaining the join samples

We can maintain the samples S_u under insertions and deletions to the relation u using the algorithm in [GMP97b]. We show next how to maintain $\mathcal{J}(S_u)$ for all u under insertions and deletions to any relation. We rely on the integrity constraint on each foreign key to enable a faster maintenance algorithm.

Our algorithm for maintaining a join sample $\mathcal{J}(S_u)$ for each u is as follows. Let p_u be the current probability for including a newly arriving tuple for relation u in the random sample S_u. On an insert of a new tuple τ into a base relation corresponding to a node u in G, we do the following. Let $u \bowtie r_2 \bowtie \cdots \bowtie r_\kappa$ be the maximum foreign key join with source u. (1) We add τ to S_u with probability p_u. (2) If τ is added to S_u, we add to $\mathcal{J}(S_u)$ the tuple $\{\tau\} \bowtie r_2 \bowtie \cdots \bowtie r_\kappa$. This can be computed by performing at most $\kappa - 1$ look-ups to the base data, one each in $r_2, \ldots, r_\kappa$. (For any key already in $\mathcal{J}(S_u)$, the look-ups for it or any of its "descendants" are not needed.) (3) If τ is added to S_u and S_u exceeds its target size, then select uniformly at random a tuple τ' to evict from S_u. Remove the tuple in $\mathcal{J}(S_u)$ corresponding to τ'.

On a delete of a tuple τ from u, we first determine if τ is in S_u. If τ is in S_u, we delete it from S_u, and remove the tuple in $\mathcal{J}(S_u)$ corresponding to τ. As in [GMP97b], if the sample becomes too small due to many deletions to the sample, we repopulate the sample by rescanning the base relations.

Note that this algorithm only performs look-ups to the base data with (small) probability p_u. Also, when a tuple is inserted into a base relation u, we never update join samples for any ancestors of u. Such updates would be costly, since these operations would be performed for every insert and for each ancestor of u. Instead, we rely on the integrity constraints to avoid these costly updates.

Theorem 4.6 *The above algorithm properly maintains all S_u as uniform random samples of u and properly maintains all join samples $\mathcal{J}(S_u)$.*

Proof. (idea) Due to the integrity constraints, for each edge from w to u, there is exactly one tuple in u joining with each tuple in w at all times. Thus any subsequent tuple inserted into u can not join with any tuple already in w, and any tuple deleted from u can not join with a tuple still in w. ∎

4.2 Reducing the space needed

Recall that in Aqua, we only store attributes of interest and we store all tuples of small relations. This reduces the columns stored for join sample tuples. To further reduce the footprint for join samples, we can renormalize the tuples in $\mathcal{J}(S_u)$ into their constituent relations and remove duplicates. To the extent that foreign keys are many-to-one, this will reduce the space, although the key will then be replicated. With this approach, when a tuple in S_u is deleted, one can either (1) immediately determine which tuples in other relations to remove, if any, by either linear search, maintaining reference counts, etc., or (2) leave the other tuples in, and then garbage collect periodically by materializing $\mathcal{J}(S_u)$ and discarding unused tuples. Alternatively, we can renormalize as above, but take the union, excluding the S_u's, of $\mathcal{J}(S_u)$ for all u, and remove duplicates.

Lemma 4.7 *For any node u whose maximum foreign key join is a κ-way join, the number of tuples in its renormalized join sample $\mathcal{J}(S_u)$ is at most $\kappa|S_u|$.*

Proof. Each tuple in the (unnormalized) $\mathcal{J}(S_u)$ contributes κ tuples to the renormalized $\mathcal{J}(S_u)$ (before duplicate removal). ∎

As an example, consider the schema in Figure 4. If we store a single copy of N and R, and hence remove them from G, then for L, PS, O, C, P, and S, the value of κ is $6, 3, 2, 1, 1$, and 1, respectively. If we take $|S_u|$ to be the same for all u in $G - \{N, R\}$, then for all data distributions, the number of tuples in the synopsis is at most $14|S_u| + |N| + |R|$. To the extent that foreign keys are many-to-one, the space can be considerably smaller than this upper bound.

5 Analytical bounds based on sample sizes

The current Aqua system focuses on *guaranteed* bounds, which provide guarantees to the user, but may be overly pessimistic in some cases. Aqua provides confidence intervals based on Hoeffding bounds (these bounds are summarized in the appendix). Since Aqua maintains join samples, $\mathcal{J}(S_u)$, we can report confidence intervals based on Hoeffding-based formulas for *single-table queries* only [Haa96], which are much faster to compute and much more accurate than the formulas involving joins.[5] To apply Hoeffding bounds, we use the bounds Aqua maintains on the minimum and maximum value for each attribute to compute guaranteed bounds on the minimum and maximum value of the expression occurring in a query, by considering how the attribute bounds may combine in the worst case. To the extent that a query predicate limits the minimum and maximum of any subexpression in the expression, better bounds are used.

In contrast to Hoeffding-based bounds, the *large sample* bounds in [HHW97, Haa98] are only heuristic bounds. Large sample bounds contain the final answer with a probability *approximately* equal to p and are based upon central limit theorems. As noted in [HHW97], the true probability can be much less than the nominal probability p. These papers do not report a method for determining when a finite sample is sufficiently large so that the bounds apply, and indeed the sample size needed can vary widely depending on the distribution of the values. Observing the values occurring in a sample is not sufficient in this regard. Thus, although it would be straightforward to consider large sample bounds in the context of Aqua, we have focused instead on guaranteed bounds, and have not implemented large sample bounds in the current system.

Evaluating sample size allocations. We now present a strategy for evaluating the effectiveness of an allocation of sample sizes among the join samples for each relation. Our goal is to provide simple, analytical bounds for the errors incurred by a broad class of queries.

We begin by considering the following simple characterization of a set, S, of queries with selects, aggregates, group bys and foreign key joins. For each relation, R_i, we have the fraction, f_i, of the queries in S for which R_i is either the source relation in a foreign key join or the sole relation in a query without joins. Next, we consider a range of representative (single table) selectivities, Q, for the predicates in queries, where the selectivities are based on the single table materialized foreign key join. (Such selectivities are the additional predicate selectivities beyond any join selectivities.) These selectivities could be determined by the query mix, but for simplicity and generality, we will assume representative selectivities of $q \in Q' = \{.01, .02, .05, .1, .2, .5, 1\}$.

[5] For queries with non-foreign-key joins, we resort to the (much weaker) multi-table formulas from [Haa96].

In what follows, we restrict our attention to the COUNT aggregate; this aggregate may be the most important for Aqua since it is used to provide size estimates for all set-valued queries, in addition to its use in aggregate queries. It is also fairly simple to analyze. We measure the effectiveness of a sample for a COUNT aggregate by the size of its relative error bound. For concreteness, we use Hoeffding-based error bounds that provide bounds on the relative error that are guaranteed to hold with 90% probability. Consider a predicate with (unknown) selectivity q followed by a COUNT, on a relation of m tuples. Let $\mathrm{Error}_q(n)$ be the relative error bound for the estimate based on a sample of size $n \ll m$. Let n' be the number of sampled tuples that satisfy the predicate. Then $\mu_n = \frac{m}{n} \cdot n'$ is an unbiased estimator for the unknown count $q \cdot m$, and Hoeffding [Hoe63] showed that $\Pr\left(|\mu_n - q \cdot m| \le m\sqrt{\frac{1}{2n}\ln\frac{2}{1-p}}\right) \ge p$ (see Appendix A). Dividing through by $q \cdot m$ to get relative error and taking $p = .9$ yields

$$\mathrm{Error}_q(n) = \frac{\sqrt{\ln(20)}}{q\sqrt{2n}} \approx \frac{1.22}{q\sqrt{n}}.$$

Thus the relative error bound for COUNT decreases with the square root of the sample size.

Let $\mathrm{Error}(n)$ be the average relative error bound over the representative selectivities Q, i.e., $\mathrm{Error}(n) = \frac{1}{|Q|}\sum_{q\in Q}\mathrm{Error}_q(n)$. Using the example representative selectivities Q', we have that

$$\mathrm{Error}(n) \approx \frac{1.22}{7\sqrt{n}} \cdot \sum_{q\in Q'} \frac{1}{q} = \frac{32.86}{\sqrt{n}}.$$

Thus the average relative error bound decreases with the square root of the sample size, and is independent of the relation size m. Moreover, roughly 4K samples suffice to have an average error bound within a factor of 2.[6]

Finally, we evaluate an allocation of sample sizes over all relations (for COUNT aggregates) as the weighted sums of the average relative error bounds. Let $n_1, n_2, \ldots, n_t$ be the sample sizes allocated to the relations $R_1, R_2, \ldots, R_t$ in the schema for join samples. Then the weighted average relative error is

$$\sum_{i=1}^{t} f_i \cdot \mathrm{Error}(n_i) \approx 32.86 \sum_{i=1}^{t} \frac{f_i}{\sqrt{n_i}}.$$

6 Using biased samples

Group by operators can pose difficulties for sampling-based estimation. Groups with relatively few members in the relation are expected to have relatively few members (possibly none) in a uniform random sample; this implies that the accuracy of estimations for such groups can be quite poor. Hellerstein *et al* [HHW97] dealt with this problem in their work on online aggregation, providing a special B-tree-based indexing mechanism to allow different-sized groups to be accessed at equal rates.

In Aqua, we can improve the accuracy of approximations under group bys without special indexing mechanisms or (random) disk accesses, by biasing our samples according to the groups. In this approach, we assume that we have *a priori* knowledge of the group by attributes, but no other information about how the groups are populated (e.g., we do not know which groups are empty). The

[6] Note that this sample size is based on Hoeffding bounds, which are often quite conservative.

technique works well for group bys on attributes in source relations of queries; for other group bys, the update time overheads to maintain the biased samples may be too large. (This is similar to [HHW97], where an a priori knowledge of groups is necessary to ensure that a B-tree exists on the appropriate attributes, and the scheme does not work well with joins.)

Consider one such a priori group by. We maintain a table of the groups that have occurred, together with a count of the number of tuples currently in the group. To ensure adequate representation in the sample for small groups, we will sample at a higher rate for such groups.

When a new tuple is inserted into a relation, we determine its group. If it is an existing group, we increment the count for the group. Otherwise, we add a new entry to the table, with count 1. We add the tuple to the sample according to the desired sample rate for a group of its size.

Since each group is its own uniform random sample, we have considerable flexibility in deciding sample rates (e.g., we need not be fair). If we wish to maintain a constant total sample size n, divided evenly among the (unknown number of) groups, we perform reservoir sampling [Vit85] on each group such that if we have observed g groups, we maintain a target sample size of n/g for each group. When a new group appears, we decrease the target sample size and (lazily) evict random tuples from each existing group. If the number of groups becomes large, we may wish to keep track of only the most popular groups. As which particular groups are and are not the most popular may change over time, we can use the algorithm in [GM98] to maintain a list of the (approximately) most popular groups.

Quantifying the advantages of biased samples. We can quantify analytically the advantages of biased samples in producing smaller confidence intervals for aggregates. Consider a sample of size n from a relation of size $m \gg n$. Consider COUNT, SUM, and AVG over expressions in the relation and let $\text{MIN} \geq 0$ and MAX be lower and upper bounds on the expression. The advantages arise from (1) maintaining the counts of each group, (2) ensuring that all groups are represented in the sample, and (3) allowing for more balanced sample sizes for each group. We consider each advantage in turn.

Maintaining the count, m', of each group not only allows for accurate COUNT answers, but also improves the Hoeffding-based confidence bounds for SUM from $m \cdot \text{MAX}\sqrt{\frac{1}{2n}\ln\frac{2}{1-p}}$ to $m' \cdot (\text{MAX} - \text{MIN})\sqrt{\frac{1}{2n'}\ln\frac{2}{1-p}}$, where $n' > 0$ is the number of sample tuples in the group (see Appendix A).

The second advantage can be considered independently of the first by assuming that counts of groups are maintained in both the uniform and biased sampling cases. In a uniform random sample, each group of size m' is expected to appear in the sample $m' \cdot \frac{n}{m}$ times, and will fail to appear in the sample with probability $> (1 - \frac{m'}{m-n})^n \approx e^{-m'n/m}$. For example, a group of size $m' = \frac{m}{10n}$ has over a 90% probability of not occurring in the sample. For any group not appearing in the sample (i.e. $n' = 0$), then $m \cdot \text{MAX}\sqrt{\frac{1}{2n}\ln\frac{2}{1-p}}$ is a sanity (upper) bound for SUM and SUM is deterministically in $[m' \cdot \text{MIN}, m' \cdot \text{MAX}]$. All that can be said for AVG is that it is deterministically in $[\text{MIN}, \text{MAX}]$. With biased sampling, we can ensure that all groups (or the most popular groups, if there are too many groups) have some minimum representation in the sample, at the expense of less samples for the larger groups.

The third advantage can be considered independently of the first two by assuming that we add a single random representative of each group to the uniform sample. The advantage can be seen by considering the AVG aggregate. By Hoeffding-based bounds for AVG due to Haas [Haa96] (see

Table Name	# of Columns	Cardinality
Customer	8	$15K$
Lineitem	16	$600K$
Nation	4	25
Order	9	$150K$
Part	9	$20K$
Partsupp	5	$80K$
Region	3	5
Supplier	7	$1K$

Table 2: Features of relations in the TPC-D benchmark.

Appendix A), the average confidence bound over $g < n$ groups is proportional to $\frac{1}{g}\sum_{i=1}^{g}\frac{1}{\sqrt{n_i}}$, where n_i is the size of the sample for group i. This is minimized by taking $n_i = \frac{n}{g}$ for all i, which can be achieved with biased sampling using the reservoir sampling approach described above. With uniform sampling, the n_i are expected to be proportional to the group size, and hence can vary widely. In the worst case of a single representative for all but one of the groups, the average confidence bound is a factor of $\approx \sqrt{n/g}$ worse with uniform sampling than with biased sampling that takes $n_i = \frac{n}{g}$.

7 Experimental evaluation of Aqua

In this section, we present the results of an experimental evaluation of the Aqua system. Using uniformly distributed as well as skewed data from the TPC-D benchmark, we show the effectiveness of Aqua in providing guaranteed and highly-accurate answers.

The rest of this section is organized as follows. We begin by describing our experimental testbed. We then evaluate the performance of Aqua in answering queries approximately using both join samples and samples on base relations. We finally show that join samples can be maintained with very little overhead and that they can be used to provide very good approximate answers, even when updates significantly change the characteristics of the underlying data.

7.1 Experimental testbed

We ran our experiments on data from the TPC-D decision support benchmark. We used a scale factor of 0.1 for generating our test data. This results in a database that is approximately 100 megabytes. Table 2 summarizes the important features of the 8 relations in the TPC-D database. The default data generator used in the TPC-D benchmark generates uniformly distributed data. In addition to this we modified the generator in order to generate databases in which some of the attributes were skewed according to Zipfian distributions.

Our experiments were run on a lightly loaded 296MHz UltraSPARC-II machine having 256 megabytes of memory and running Solaris 5.6. All data was kept on a local disk with a streaming throughput of about 5 megabytes/second.

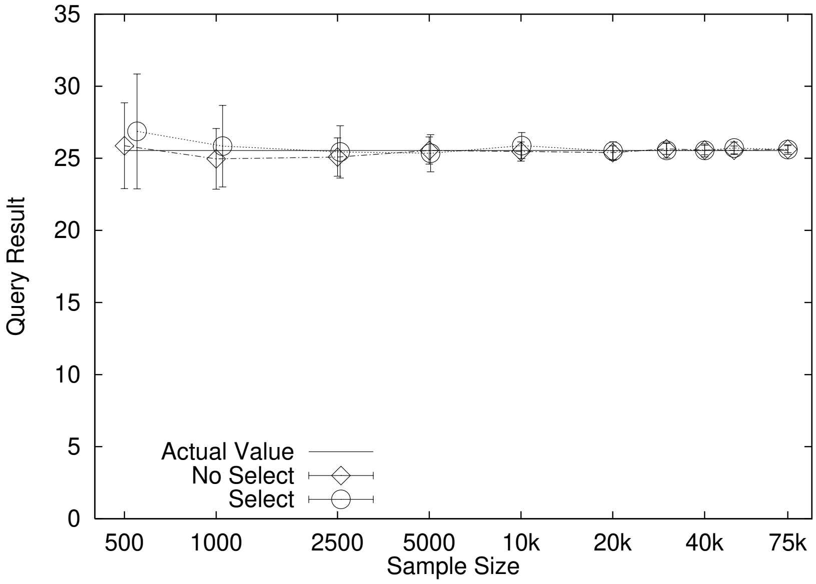


Figure 5: This figures shows approximate answers and confidence bounds (at 95% confidence) while computing an aggregate value of an attribute from a base relation. The estimates and confidence bounds for the "select" case have been shifted slightly to the right for visual clarity.

7.2 Computing aggregates on base relations

We first considered the effectiveness of samples when computing aggregate values directly from base relations. The query we use is a piece of Query Q1[7] in the TPC-D benchmark and corresponds to the SQL query:

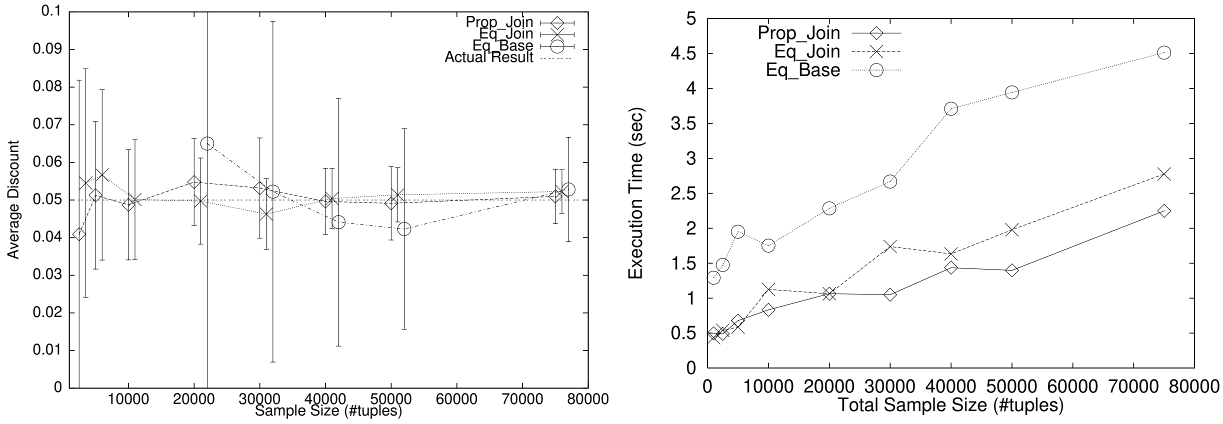
```
select avg(l_quantity) from lineitem
    where l_shipdate <= DATE [indate]
```

([indate] is an input parameter supplied to the query.) Figure 5 shows the results from evaluating the query using samples of various sizes, on uniformly distributed data. The results for skewed data were substantively identical. The figure attests to the fact that one can obtain accurate estimates even from relatively small samples. In addition to plotting the actual value (which, of course, does not change with sample size), the figure plots reported aggregates and bounds for the case when the select condition is applied, as well as for the case when there is no select condition. Note that the confidence bounds are not as good for the case when the select condition is applied since the select effectively reduces the sample size.

7.3 Computing aggregates on results of joins

In this section, we consider an aggregate that is computed on the result of a complex select-join query. The query we use is based on query Q5 in the TPCD benchmark and is an aggregate that is computed

[7]We can provide approximate answers for *all* the aggregates in Query Q1. We have presented results for one of the aggregates to keep the resulting graph simple.



(a) Result Accuracy (b) Execution Time

Figure 6: Behavior of different strategies for allocating samples among base relations samples and join samples shown against increasing *total* space for the samples.

on the join of Lineitem, Customer, Order, Supplier, Nation and Region. Of the six relations involved in the join, the Nation and Region relations are sampled in their entirety by Aqua because of their low cardinality. This effectively makes the query as difficult as estimating an aggregate from a (still complex) four-way join. The SQL statement for the query is:

```
select avg(l_discount) from customer, order, lineitem, supplier, nation, region
        where c_custkey = o_custkey and o_orderkey = l_orderkey
            and l_suppkey = s_suppkey and c_nationkey = s_nationkey
            and s_nationkey = n_nationkey and n_regionkey = r_regionkey
            and r_name = [region]
            and o_orderdate >= DATE [date] and
                o_orderdate < DATE [date] + interval '1' year
```

This query computes the average discount given by suppliers in a nation to customers who are in the same nation. The select conditions take two input parameters that restrict suppliers and customers to be within a specific region and focus on business conducted within a specific time interval.

We considered four strategies for dividing up a sample space among base relations samples and join samples. The sample space was specified in terms of the number of tuples that could be kept in the sample.[8] The four strategies that we considered were:

- *Eq_Base*, which divides up the sample equally amongst the base relations involved in the query,
- *Prop_Base*, which divides up the sample amongst the base relations such that each base relation was allocated a sample that was proportional to its size,
- *Eq_Join*, which is like *Eq_Base*, except that it also allocates space to the join sample, and,
- *Prop_Join*, which is like *Prop_Base*, except that it also allocates space to the join sample.

[8] This is a simplification of the actual problem, since different relations have different sizes.

Figure 6 shows the results from an experiment we conducted to study the behavior of the different strategies. We evaluate these strategies in terms of: (a) the accuracy of the approximate result that they produce and (b) the time that they take to compute the approximate answer. Figure 6(a) shows that computing the aggregate from a join sample not only produces lesser deviation from the actual aggregate value but also provides tighter confidence bounds. For smaller sample sizes, sampling the base relations alone fails to generate any output — *Eq_Base* produces output only beyond 20,000 tuples.[9]

This query effectively demonstrates the need for maintaining join samples. Each join that the Lineitem table undergoes reduces the output of the query by a factor that is proportional to the sampling fraction of the table that Lineitem is being joined with. The result is that even large samples on the base relations simply do not yield any output tuples! This is in stark contrast to the case where even small join samples are very effective in answering aggregate queries very effectively.

Figure 6(b) plots the time taken by the various strategies to execute the query. It took 76.3 seconds to execute the query on the base data (not shown). As expected, the response times increase with increasing sample size but are still negligible compared to the time taken to generate the actual result from the base relations. This demonstrates that it is possible to obtain extremely fast query responses at marginal loss in accuracy.

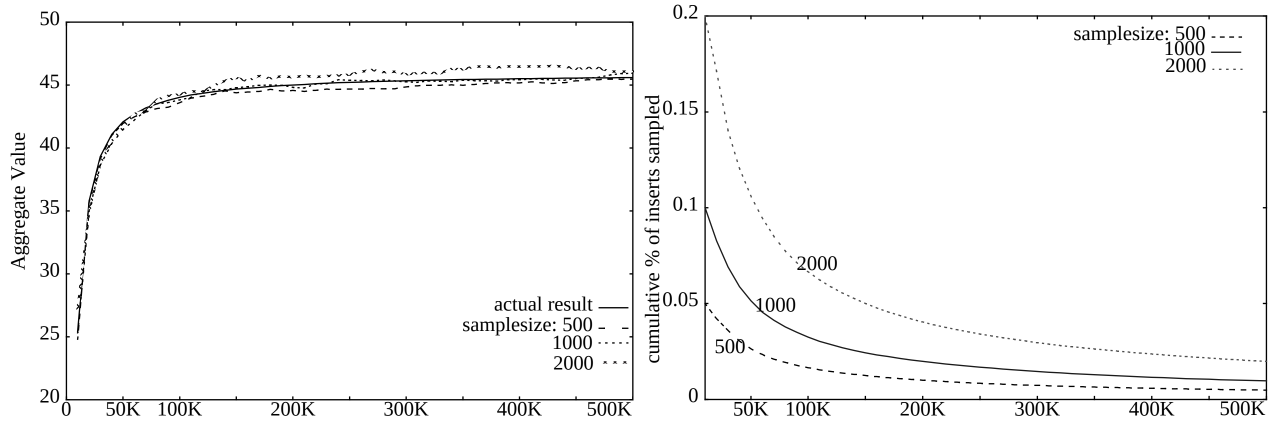
7.4 Maintenance of join samples

In this section, we show experimental results demonstrating that join samples can be maintained with very minimal overhead. Such join samples can give very good approximate answers even when updates significantly change the nature of the underlying data. We base this section on a join between the Lineitem and Order tables. The query used retrieves the average quantity of lineitems that have a particular orderstatus. The SQL statement for the query is:

```
select avg(l_quantity) from lineitem, order
    where l_orderkey = o_orderkey and o_orderstatus = F
```

In order to maintain a sample of the join between Lineitem and Order as Lineitem tuples are inserted, we sample the Lineitem tuples that are inserted using a reservoir sampling algorithm. The sampled Lineitem tuples are joined with the base (not sampled!) table of the Order relation to update the join sample (in accordance with the algorithm in Section 4.1). Figure 7(a) plots the aggregate values computed from join samples of different sizes. Even for extremely small sample sizes, the join sample is able to track the actual aggregate value quite closely despite significant changes in the data distribution. Figure 7(b) then shows that maintenance of join samples is very inexpensive, by plotting the average fraction of the inserted Lineitem tuples that are actually inserted into the join sample. As is clear from the figure, this number is a small fraction of the total number of tuples inserted. (For example, when maintaining a sample of 1000 tuples and processing $500,000$ inserts, we go to the base data only 4822 times.)

[9] *Prop_Base* fails to produce any result for the entire range studied, and thus is not shown.



(a) Result Accuracy (b) Maintenance Cost

Figure 7: The left side of the figure shows aggregate values computed from join samples of various sizes. The right side shows the cost of online maintenance of the samples. Both of them are plotted against $500,000$ updates to the Lineitem table.

7.5 Experimental conclusions

The experimental results in Sections 7.2–7.4 empirically demonstrate several important hypotheses that Aqua is based on. The first is that small samples of relations, especially join samples, can give extremely quick answers of very high quality for many queries. The second is that maintenance of these samples, even join samples, can be performed easily and efficiently. Such samples provide good approximate answers even when the underlying data is rapidly changing in character.

8 Previous related work

Hellerstein *et al.* [HHW97] proposed a framework for approximate answers of aggregation queries called *online aggregation*, in which the base data is scanned in a certain order at query time and the approximate answer for an aggregation query is updated as the scan proceeds (continuous reporting). A graphical display depicts the answer and a (decreasing) confidence interval as the scan proceeds, so that the user may stop the process at any time. The only synopses maintained are the indexes to enable special treatment of small sets in group bys. Since the reported tuples are retrieved from the base data at query time, the response time will be orders of magnitude slower than in Aqua. If the scan order for a group is random, then randomly-selected certain tuples with guaranteed accuracy measures will be reported. Moreover, considering all groups, biased-selected certain tuples will be reported with the bias in favor of the small sets, as desired. The disadvantage of a random scan order is that the response time is even slower. If the scan order is the order of the data on the disks, then the response time is faster than with random order, but now the reported tuples are arbitrary certain tuples with heuristic accuracy measures (which can be quite inaccurate).

Other systems support limited on-line aggregation features; e.g., the Red Brick system supports running COUNT, AVG, and SUM (see [HHW97]). Since the scan order used to produce these aggregations is not random, only heuristic accuracy measures are possible, and the accuracy can be quite poor.

Table 3: Some related work, evaluated as approximate query engines.

system	design goal/coverage	response time	accuracy	update time	footprint
[HHW97] random	online aggregation	quite slow	good if no joins	fast	small
[HHW97] scan	online aggregation	slow	poor	fast	small
Red Brick	running COUNT, AVG, SUM	slow	poor	very fast	none
Oracle Rdb, [BM96]	fast-first on any query	fairly slow	poor	very fast	none
APPROXIMATE [VL93]	general approx. query eng.	slow	poor	fairly fast	small
Aqua	general approx. query eng.	fast	good	modest	modest

The response time is slow since the tuples are retrieved from the base data at query time; however, since there are no synopses to maintain, there are no overheads at update time and no footprint for synopses.

There have been several recent works on "fast-first" query processing, whose goal is to quickly provide a few tuples of the query answer. Bayardo and Miranker [BM96] devise techniques for optimizing and executing queries using pipelined, nested-loops joins in order to minimize the latency until the first answer is produced. The Oracle Rdb system [AZ96] provides support for running multiple query plans simultaneously, in order to provide for fast-first query processing. These systems report arbitrary certain representative tuples, by accessing the base data at query time. No size estimates or other meta-information are provided with the representative tuples. No synopses need be maintained.

In the APPROXIMATE query processor, developed by Vrbsky and Liu [VL93], an approximate answer to a set-valued query is any superset of the exact answer that is a subset of the cartesian product. The goal of the query processor is to produce monotonically improving approximate answers, by decreasing the superset as the processing proceeds. The query processor uses various class hierarchies to iteratively fetch blocks relevant to the answer, producing tuples certain to be in the answer while narrowing the possible classes that contain the answer. There are no bounds provided on the accuracy, no size estimates or other meta-information, and the representative tuples are arbitrary certain tuples. Other related query processors (see the references in [VL93]) likewise operate on the base data at query time and define an approximate answer for set-valued queries to be subsets and supersets that converge to the exact answer.

Table 3 provides a summary of the comparison between these previous works and Aqua, evaluating the systems using the metrics for approximate query engines described in Section 2.2. This comparison is of course unfair, since none of these other systems (other than APPROXIMATE) were designed to be approximate query engines. However, it reflects the state-of-the-art in approximate query engines prior to Aqua.

Barbará *et al.* [BDF$^+$97] present a survey of *data reduction* techniques; these can be used for a variety of purposes, including providing approximate query answers. Gibbons and Matias [GM98] introduced two sampling-based synopses, concise samples and counting samples, that can be used to obtain larger samples for the same footprint and to improve approximate query answers for hot list queries. Maintenance algorithms were presented for both concise and counting samples. Olken and Rotem [OR92] presented techniques for maintaining random sample views. Matias *et al.* [MVN93, MVY94, MSY96] proposed and studied *approximate data structures* that provide fast approximate answers. These data structures have linear space footprints.

Other works on incremental maintenance of approximate synopses include [FM83, FM85, WVZT90, HNSS95, AMS96, GMP97b, GP97]. Finally, there has been considerable work on sampling-based estimation algorithms for use within a query optimizer (e.g., [HÖT88, HÖT89, LN89, LN90, LNS90, HÖD91, HS92, LS92, LNSS93, HNSS93, HNS94, LN95, HNSS95, GGMS96]).

None of this previous work uses the new techniques described in this paper.

9 Conclusions

This paper describes the Aqua system, for fast, highly-accurate approximate query answers. It is well known that join operators seriously degrade estimation accuracy, so we have devised special techniques for handling the multi-way foreign key joins that are popular in OLAP. Group bys can also degrade estimation accuracy, so we have presented a biased sampling technique for handling group bys. Aqua provides approximate answers using small, precomputed synopses of the underlying base data; we have developed efficient algorithms for incremental maintenance of all synopses used in the current Aqua system. The system provides accuracy guarantees without any a priori assumptions on either the data distribution, the order in which the base data is loaded, or the layout of the data on the disks.

Analytical bounds and experimental results on TPC-D queries demonstrate Aqua's effectiveness, even in the presence of data distribution changes. Aqua is the first system to provide fast (no accesses to the base data at query time), highly-accurate approximate answers for a broad class of queries that arise in data warehousing scenarios.

Since Aqua provides answers typically without accessing the base data, it can be physically distant from the data warehouse during query time, allowing for considerable flexibility. For example, unlike previous systems (such as those in Table 3), Aqua can provide approximate answers even when the base data is unavailable.

While the current system focuses on answers to broad classes of queries, special features can be added to Aqua to improve the accuracy of specific classes of queries, such as those reported in [AMS96, GMP97b, BDF$^+$97, GP97, GM98]. Further details can be found in the Aqua Project White Paper [GMP97a].

Acknowledgements

We thank Minos Garofalakis and Hank Korth for discussions related to this work.

References

[AMS96] N. Alon, Y. Matias, and M. Szegedi. The space complexity of approximating the frequency moments. In *Proc. 28th ACM Symp. on the Theory of Computing*, pages 20–29, May 1996.

[AZ96] G. Antoshenkov and M. Ziauddin. Query processing and optimization in Oracle Rdb. *VLDB Journal*, 5(4):229–237, 1996.

[BDF$^+$97] D. Barbará, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. Ioannidis, H. V. Jagadish, T. Johnson, R. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The New Jersey data reduction report. *Bulletin of the Technical Committee on Data Engineering*, 20(4):3–45, 1997.

[BM96] R. J. Bayardo, Jr. and D. P. Miranker. Processing queries for first-few answers. In *Proc. 5th International Conf. on Information and Knowledge Management*, pages 45–52, 1996.

[FJS97] C. Faloutsos, H. V. Jagadish, and N. D. Sidiropoulos. Recovering information from summary data. In *Proc. 23rd International Conf. on Very Large Data Bases*, pages 36–45, August 1997.

[FM83] P. Flajolet and G. N. Martin. Probabilistic counting. In *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pages 76–82, October 1983.

[FM85] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Computer and System Sciences*, 31:182–209, 1985.

[GGMS96] S. Ganguly, P. B. Gibbons, Y. Matias, and A. Silberschatz. Bifocal sampling for skew-resistant join size estimation. In *Proc. 1996 ACM SIGMOD International Conf. on Management of Data*, pages 271–281, June 1996.

[GM95] P. B. Gibbons and Y. Matias, August 1995. Presentation and feedback during a Bell Labs-Teradata presentation to Walmart scientists and executives on proposed improvements to the Teradata DBS.

[GM98] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proc. ACM SIGMOD International Conf. on Management of Data*, June 1998. To appear.

[GMP97a] P. B. Gibbons, Y. Matias, and V. Poosala. Aqua project white paper. Technical report, Bell Laboratories, Murray Hill, New Jersey, December 1997.

[GMP97b] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *Proc. 23rd International Conf. on Very Large Data Bases*, pages 466–475, August 1997.

[GP97] V. Ganti and V. Poosala. Space-efficient approximation of the data cube. Technical report, Bell Laboratories, Murray Hill, New Jersey, November 1997.

[Gra94] G. Graefe. Volcano, an extensible and parallel dataflow query processing system. *IEEE Trans. on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[Haa96] P. J. Haas. Hoeffding inequalities for join-selectivity estimation and online aggregation. Technical Report RJ 10040, IBM Almaden Research Center, San Jose, CA, 1996.

[Haa98] P. J. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *Proc. 9th International Conf. on Scientific and Statistical Database Management*, 1998. To appear.

[HHW97] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 171–182, May 1997.

[HNS94] P. J. Haas, J. F. Naughton, and A. N. Swami. On the relative cost of sampling for join selectivity estimation. In *Proc. 13th ACM Symp. on Principles of Database Systems*, pages 14–24, May 1994.

[HNSS93] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Fixed-precision estimation of join selectivity. In *Proc. 12th ACM Symp. on Principles of Database Systems*, pages 190–201, May 1993.

[HNSS95] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proc. 21st International Conf. on Very Large Data Bases*, pages 311–322, September 1995.

[HÖD91] W.-C. Hou, G. Özsoyoğlu, and E. Dogdu. Error-constrained COUNT query evaluation in relational databases. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 278–287, May 1991.

[Hoe63] W. Hoeffding. Probability inequalities for sums of bounded random variables. *J. American Statistical Association*, 58:13–30, 1963.

[HÖT88] W.-C. Hou, G. Özsoyoğlu, and B. K. Taneja. Statistical estimators for relational algebra expressions. In *Proc. 7th ACM Symp. on Principles of Database Systems*, pages 276–287, March 1988.

[HÖT89] W.-C. Hou, G. Özsoyoğlu, and B. K. Taneja. Processing aggregate relational queries with hard time constraints. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 68–77, June 1989.

[HS92] P. J. Haas and A. N. Swami. Sequential sampling procedures for query size estimation. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 1–11, June 1992.

[LN89] R. J. Lipton and J. F. Naughton. Estimating the size of generalized transitive closures. In *Proc. 15th International Conf. on Very Large Data Bases*, pages 165–172, August 1989.

[LN90] R. J. Lipton and J. F. Naughton. Query size estimation by adaptive sampling. In *Proc. 9th ACM Symp. on Principles of Database Systems*, pages 40–46, April 1990.

[LN95] R. J. Lipton and J. F. Naughton. Query size estimation by adaptive sampling. *J. Computer and System Sciences*, 51(1):18–25, 1995.

[LNS90] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 1–12, May 1990.

[LNSS93] R. J. Lipton, J. F. Naughton, D. A. Schneider, and S. Seshadri. Efficient sampling strategies for relational database operations. *Theoretical Computer Science*, 116(1-2):195–226, 1993.

[LS92] Y. Ling and W. Sun. A supplement to sampling-based methods for query size estimation in a database system. *SIGMOD Record*, 21(4):12–15, 1992.

[MSY96] Y. Matias, S. C. Sahinalp, and N. E. Young. Performance evaluation of approximate priority queues. Presented at *DIMACS Fifth Implementation Challenge: Priority Queues, Dictionaries, and Point Sets*, organized by D. S. Johnson and C. McGeoch, October 1996.

[MVN93] Y. Matias, J. S. Vitter, and W.-C. Ni. Dynamic generation of discrete random variates. In *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, pages 361–370, January 1993.

[MVY94] Y. Matias, J. S. Vitter, and N. E. Young. Approximate data structures with applications. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 187–194, January 1994.

[OR92] F. Olken and D. Rotem. Maintenance of materialized views of sampling queries. In *Proc. 8th IEEE International Conf. on Data Engineering*, pages 632–641, February 1992.

[Sch97] D. Schneider. The ins & outs (and everything in between) of data warehousing, August 1997. Tutorial in the *23rd International Conf. on Very Large Data Bases.*

[Vit85] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[VL93] S. V. Vrbsky and J. W. S. Liu. Approximate—a query processor that produces monotonically improving approximate answers. *IEEE Trans. on Knowledge and Data Engineering*, 5(6):1056–1068, 1993.

[WVZT90] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, 15(2):208–229, 1990.

A Hoeffding-based bounds

In this appendix, we summarize Hoeffding-based upper bounds for t such that $P(|\mu_n - \mu| \leq t) \geq p$, where μ is the precise result to an aggregate, and μ_n is an estimate based on n samples.

In order to apply these bounds, we maintain in Aqua:

- for a relation R, a count of the number of tuples, m, and a uniform random sample of $n \leq m$ tuples.

- for an attribute A in a relation R that may be used in an AVG or SUM aggregate, an upper bound, MAX, and a lower bound MIN ≥ 0, i.e., the attribute values are nonnegative values in [MIN, MAX].

Queries we consider are the aggregates AVG, SUM, COUNT after join and group by operators, and predicates. Aggregates may be of arbitrary expressions with nonnegative values. We compute upper and lower bounds for the expression by considering the MAX and MIN of the individual attributes in the expression and how they may combine in the worst case. Group by into k groups is solved by applying the bounds separately to each group with an appropriate predicate that filters out that group.

Single relation with no predicates. Aggregate over an expression with values in [MIN, MAX], MIN ≥ 0, with sample size n and relation size m.

Aggregate	Bound	Estimate	Ref.
AVG	$(\text{MAX} - \text{MIN})\sqrt{\frac{1}{2n}\ln\frac{2}{1-p}}$	$\frac{\text{sum of sample}}{n}$	[Hoe63]
SUM	$m(\text{MAX} - \text{MIN})\sqrt{\frac{1}{2n}\ln\frac{2}{1-p}}$	$\frac{m}{n} \cdot (\text{sum of sample})$	[Hoe63]
COUNT	0	m	trivial

Single relation with predicates. Aggregate over an expression such that the values of the expression for tuples that satisfy the predicate are in [MIN, MAX], MIN ≥ 0. These MIN and MAX may not be known (for instance, they cannot be determined from the sample). Conservatively, the MIN and MAX of the expression over the entire relation can be used, and to the extent that the predicate limits the MIN and MAX of any subexpression in the expression, better bounds can be used. Sample size is n, relation size is m, and $n' > 0$ is the number of sampled tuples that satisfy the predicate.

Aggregate	Bound	Estimate	Ref.
AVG	$(\text{MAX} - \text{MIN})\sqrt{\frac{1}{2n'}\ln\frac{2}{1-p}}$	$\frac{\text{sum of sample satisfying pred.}}{n'}$	[Haa96]
SUM	$m \cdot \text{MAX}\sqrt{\frac{1}{2n}\ln\frac{2}{1-p}}$	$\frac{m}{n} \cdot (\text{sum of sample satisfying pred.})$	[Hoe63]
COUNT	$m\sqrt{\frac{1}{2n}\ln\frac{2}{1-p}}$	$\frac{m}{n} \cdot n'$	[Hoe63]

When the number, m', of tuples that satisfy the predicate is known, then we can obtain better bounds for SUM (and trivially, for COUNT). This case arises, for example, when Aqua maintains the size of the groups for an *a priori* group by. The bounds we obtain arise by viewing this scenario as a sample of $n' > 0$ from a relation of m', and then applying the "no predicate" bounds above.

Aggregate	Bound	Estimate
SUM	$m' \cdot (\text{MAX} - \text{MIN})\sqrt{\frac{1}{2n'}\ln\frac{2}{1-p}}$	$\frac{m'}{n'} \cdot (\text{sum of sample satisfying pred.})$
COUNT	0	m'

When $n' = 0$, then when m' is unknown, the estimated SUM and COUNT are zero and the bound reported in the table for m' unknown is a sanity (upper) bound on the aggregate. All that can be said for AVG is that it is deterministically in [MIN, MAX]. When $n' = 0$ and m' is known, then

$m \cdot \text{MAX}\sqrt{\frac{1}{2n}\ln\frac{2}{1-p}}$ is a sanity (upper) bound for SUM and SUM is deterministically in $[m' \cdot \text{MIN}, m' \cdot \text{MAX}]$. The estimated COUNT is trivially m'.

Multiple relations, samples on base relations. Consider two relations, R_1 and R_2, with m_1 and m_2 tuples, respectively, and an aggregate over an expression on the tuples in $R_1 \bowtie R_2$ that satisfy a predicate. (The predicate includes the join criterion, and may or may not include other selectivity criteria – the bounds are the same in either case.) If we have a precomputed sample of $R_1 \bowtie R_2$, then we can apply the single relation bounds given above. If instead all we have are samples, s_1 and s_2, of the individual relations, then the following (weaker) bounds apply. Let n_1 and n_2 be the number of tuples in s_1 and s_2, respectively, let $n = \min(n_1, n_2)$, and let $n' > 0$ be the number of tuples in $s_1 \bowtie s_2$ that satisfy the predicate. Let MIN and MAX be such that the values of the expression for tuples in $R_1 \bowtie R_2$ that satisfy the predicate are in $[\text{MIN}, \text{MAX}]$, $\text{MIN} \geq 0$.

Aggregate	Bound	Estimate	Ref.
AVG	avg is deterministically in $[\text{MIN}, \text{MAX}]$	$\frac{\text{sum of } s_1 \bowtie s_2 \text{ sat. pred.}}{n'}$	trivial
SUM	$m_1 \cdot m_2 \cdot \text{MAX}\sqrt{\frac{1}{2n}\ln\frac{2}{1-p}}$	$\frac{m_1 m_2}{n_1 n_2} \cdot (\text{sum of } s_1 \bowtie s_2 \text{ sat. pred.})$	[Haa96]
COUNT	$m_1 \cdot m_2\sqrt{\frac{1}{2n}\ln\frac{2}{1-p}}$	$\frac{m_1 m_2}{n_1 n_2} \cdot n'$	[Haa96]

Obtaining good bounds for AVG seems to be a difficult problem and none are known [Haa96], so we state the trivial bound in the above table.

The bounds in the table can be extended to $K > 2$ relations in the obvious manner, e.g., the estimate for COUNT is $\frac{m_1 m_2 \cdots m_K}{n_1 n_2 \cdots n_K} \cdot n'$, where n' is the number of tuples in $s_1 \bowtie s_2 \bowtie \cdots \bowtie s_K$ that satisfy the predicate. When $n' = 0$, the estimated SUM and COUNT are zero and the bound reported in the table above is a sanity (upper) bound on the aggregate.