Online Aggregation

Joseph M. Hellerstein
Computer Science Division
University of California, Berkeley
jmh@cs.berkeley.edu

Peter J. Haas
Almaden Research Center
IBM Research Division
peterh@almaden.ibm.com

Helen J. Wang
Computer Science Division
University of California, Berkeley
helenjw@cs.berkeley.edu

Abstract

Aggregation in traditional database systems is performed in batch mode: a query is submitted, the system processes a large volume of data over a long period of time, and, eventually, the final answer is returned. This archaic approach is frustrating to users and has been abandoned in most other areas of computing. In this paper we propose a new *online aggregation* interface that permits users to both observe the progress of their aggregation queries and control execution on the fly. After outlining usability and performance requirements for a system supporting online aggregation, we present a suite of techniques that extend a database system to meet these requirements. These include methods for returning the output in random order, for providing control over the relative rate at which different aggregates are computed, and for computing running confidence intervals. Finally, we report on an initial implementation of online aggregation in POSTGRES.

1 Introduction

Aggregation is an increasingly important operation in today's relational database management systems (DBMS's). As data sets grow larger and both users and user interfaces become more sophisticated, there is a growing emphasis on extracting not just specific data items, but also general characterizations of large subsets of the data. Users want this aggregate information right away, even though producing it may involve accessing and condensing enormous amounts of information.

Unfortunately, aggregation processing in today's database systems closely resembles the batch processing of the 1960's. When users submit an aggregation query to the system, they are forced to wait without feedback while the system churns through millions of records or more. Only after a significant period of time does the system respond with the (usually small) final answer. A particularly frustrating aspect of this problem is that aggregation queries are typically used to get a "rough picture" of a large body of information, and yet they are computed with painstaking precision, even in situations where an acceptably precise approximation might be available very quickly.

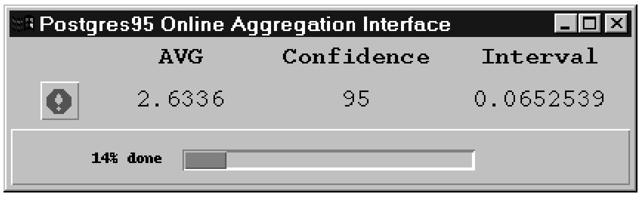

Figure 1: An online aggregation interface for Query 1.

We propose changing the interface to aggregation processing and, by extension, changing aggregation processing itself. The idea is to perform aggregation *online* in order to allow users both to observe the progress of their queries and to control execution on the fly. In this paper we present motivation, methodology, and some initial results on enhancing a relational DBMS to support online aggregation. This enhancement requires changes not only to the user interface, but also to the techniques used for query optimization and execution. We also show how both new and existing statistical estimation techniques can be incorporated into the system to help the user assess the proximity of the running aggregate to the final result; the proposed interface makes these techniques accessible even to users with little or no statistical background. As discussed below, the online aggregation interface described here goes well beyond merely providing a platform for such statistical estimation techniques, and permits an interactive approach to both formal and informal data exploration and analysis.

1.1 A Motivating Example

As a very simple example, consider the query that finds the average grade in a course:

```
Query 1:                                          AVG
SELECT AVG(final_grade)                   ------------
  FROM grades                             | 2.631046 |
 WHERE course_name = 'CS186';             ------------
```

If there is no index on the `course_name` attribute, this query scans the entire `grades` table before returning the answer shown above.

As an alternative, consider the user interface in Figure 1, which could appear immediately after the user submits the query. This interface can begin to display output as soon as the system retrieves the first tuple that satisfies the `WHERE` clause. The output is updated regularly, at a speed that is comfortable to the human observer. The `% done` and status bar display give an indication of the amount of processing remaining before completion. The `AVG` field shows the running aggregate, i.e., an estimate of the final result based on all the records retrieved so far. The `Confidence` and `Interval` fields give a probabilistic estimate of the proximity of the current running aggregate to the final result — according to Figure 1, for example, the current average is within $\pm.0652539$ of the final result with 95% probability. The interval 2.6336 ± 0.0652539 is called a *running confidence interval*. As soon as the query completes, this statistical information becomes unnecessary and need no longer be displayed.

This interface is significantly more useful than the "blinking cursor" or "wristwatch icon" traditionally presented to users during aggregation processing. It presents information at all times, and more importantly it gives the user control over processing. The user is allowed to trade accuracy for time, and to do so on the fly, based on changing or unquantifiable human factors including time constraints, accuracy needs, and priority of other tasks. Since the user sees the ongoing processing, there is no need to specify these factors in advance.

Obviously this example is quite simple; more complex examples are presented below. Even in this very simple example, however, the user is given considerably more control over the system than was previously available. In the rest of the paper we highlight additional ways that a user can control aggregation (Sections 1.2 and 2). We discuss a number of system issues that need to be addressed in order to best support this sort of control (Section 3), provide formulas for computing `Confidence` and `Interval` parameters (Section 4 and the Appendix), and present results from our initial implementation of online aggregation in POSTGRES (Section 5).

1.2 Online Aggregation and Statistical Estimation

Assuming that records are retrieved in random order, a running aggregate can be viewed as a statistical estimator of the final query result. The proximity of the running aggregate to the final result can therefore be expressed, for example, in terms of a running confidence interval as illustrated above. The width of such a confidence interval serves as a measure of the precision of the estimator. Previous work [HOT88, HNSS96, LNSS93] has been concerned with methods for producing a confidence interval with a width that is specified prior to the start of query processing (e.g. "get within 2% of the actual answer with 95% probability"). The underlying idea in most of these methods is to effectively maintain a running confidence interval (not displayed to the user) and stop sampling as soon as the length of this interval is sufficiently small. Hou, et al. [HOT89] consider the related problem of producing a confidence interval of minimal length, given a real-time stopping condition (e.g. "run for 5 minutes only").

A key strength of an online aggregation interface is that confidence intervals can be exploited without requiring *a priori* specification of stopping conditions. Though this may seem a simple point, consider the case of an aggregation query with a `GROUP BY` clause and six groups in its output, as in Figure 2. In an online aggregation system, the user can be presented with six outputs and six "Stop-sign" buttons. In a traditional DBMS, the user does not know the output groups *a priori*, and hence cannot control the query in a group-by-group fashion.

Because the online aggregation interface is natural and

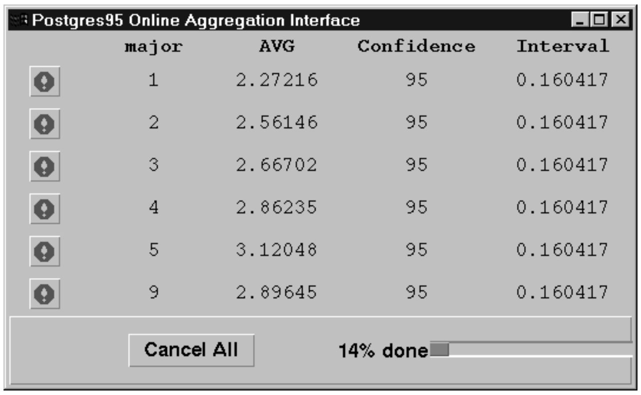

Figure 2: An online aggregation interface with groups.

easy to use, confidence-interval methodology is more accessible to non-statistical users than in a traditional DBMS. Busy end-users are likely to be quite comfortable with the online aggregation "Stop-sign" buttons, since such interfaces are familiar from popular tools like web browsers. End-users are certainly less likely to be comfortable specifying statistical stopping conditions. They are also unlikely to want to specify explicit real-time stopping conditions, given that constraints in a real-world scenario are fluid — often another minute or two of processing "suddenly" becomes worthwhile at a previously imposed deadline. The familiarity and naturalness of the online aggregation interface cannot be overemphasized. It has been shown in the User Interface literature that status bars alone improve a user's perception of the speed of a system [Mye85]. The combination of these status bars with both running estimates of the final result and online processing controls has the potential to significantly increase user satisfaction.

The increase in power of the online aggregation interface over traditional interfaces calls for commensurately more powerful statistical estimation techniques. Some of the previous methods (such as "double sampling" [HOD91]) for computing confidence intervals assume that records are sampled using techniques that are not appropriate in the setting of online aggregation. Previous work also has focused primarily on `COUNT` queries, and a number of the confidence-interval formulas that have been proposed are based on Chebyshev's inequality. We provide confidence-interval formulas (see Section 4 and the Appendix) that are applicable to a much wider variety of aggregation queries. The formulas for "conservative" confidence intervals are based on recent extensions to Hoeffding's inequality [Hoe63] and lead to conservative confidence intervals that are typically much narrower than corresponding intervals based on Chebyshev's inequality.

Although the above discussion has focused on issues pertinent to statistical estimation, it is important to remember that much of the benefit derived from online aggregation is not statistical in nature. The ongoing feedback provided by an online aggregation interface allows intuitive, non-statistical insight into the progress of a query. It also permits ongoing non-textual, non-statistical representations of a query's output. One common example of this is the appearance of data on a map or graph as they are retrieved from the database.

1.3 Other Related Work

An interesting new class of systems is developing to support so-called On-Line Analytical Processing (OLAP) [CCS93]. Though none of these systems support online aggregation to the extent proposed here, one system — Red Brick — supports running count, average, and sum functions. One of the features of OLAP systems is their support for super-aggregation ("roll-up"), sub-aggregation ("drill-down") and cross-tabulation. The CUBE operator [GBLP96] has been proposed as an addition to SQL to allow standard relational systems to support these kinds of aggregation. Computing CUBE queries typically requires significant processing [AAD+96], and batch-style aggregation systems will be very unpleasant to use for these queries. Moreover, it is likely that accurate computation of the entire data cube will often be unnecessary; online approximations of the various aggregates are likely to suffice in numerous situations.

Other recent results on relational aggregation have focused on new transformations for optimizing queries with aggregation [CS96, GHQ95, YL95, SPL96, SHP+96]. The techniques in these papers allow query optimizers more latitude in reordering operators in a plan. They are therefore beneficial to any system supporting aggregation, including online aggregation systems.

There has been some initial work on "fast-first" query processing, which attempts to quickly return the first few tuples of a query. Antoshenkov and Ziauddin report on the Oracle Rdb (formerly DEC Rdb/VMS) system, which addresses the issues of fast-first processing by running multiple query plans simultaneously; this intriguing architecture requires some unusual query processing support [AZ96]. Bayardo and Miranker propose optimization and execution techniques for fast-first processing using nested-loops joins [BM96]. Much of this work is potentially applicable to online aggregation. The performance goals of online aggregation are somewhat more complex than those of fast-first queries, as we describe in Section 2.

A different but related notion of online query processing was implemented in a system called APPROXIMATE [VL93]. This system defines an approximate relational algebra which it uses to process standard relational queries in an iteratively refined manner. If a query is stopped before completion, a superset of the exact answer is returned in a combined extensional/intensional format. This model is different from the type of data browsing we address with online aggregation: it is dependent on carefully designed metadata and does not address aggregation or statistical assessments of precision.

2 Usability and Performance Goals

In this section, we outline usability and performance goals that must be considered in the design of a system for online aggregation. These goals are different than those in either a traditional or real-time DBMS. In subsequent sections, we describe how these goals are met in our initial implementation.

2.1 Usability Goals

Continuous Observation: As indicated above, statistical, graphical, and other intuitive interfaces should be presented to allow users to observe the processing, and get a sense of the current level of precision. The set of interfaces must be extensible, so that an appropriate interface can be presented for each aggregate function, or combination of functions.

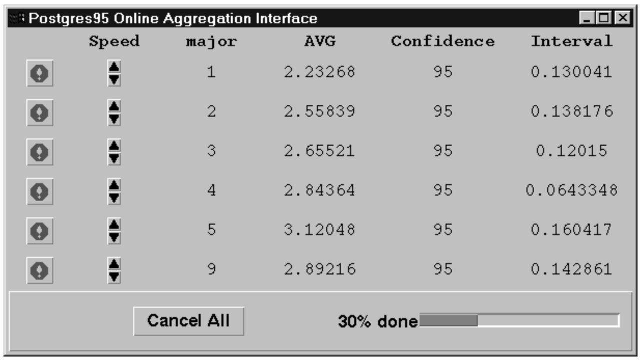

Figure 3: A speed-controllable multi-group online aggregation interface.

A good Application Programming Interface (API) must be provided to facilitate this.

Control of Time/Precision: Users should be able to terminate processing at any time, thereby controlling the tradeoff between time and precision. Moreover, this control should be offered at a relatively fine granularity. As an example, consider the following query:

```
Query 2:
SELECT AVG(final_grade) FROM grades
GROUP BY major;
```

The output of this query in an online aggregation system can be a set of interfaces, one per output group, as in Figure 2. The user should be able to terminate processing of each group individually. Such precise control is permitted by the interface in Figure 2.

Control of Fairness/Partiality: Users should be able to control the relative rate at which different running aggregates are updated. When aggregates are computed simultaneously for more than one group (as in Query 2 above), and each group is equally important, the user may want to ensure that either (i) the running aggregates are all updated at the same rate or (ii) the widths of the running confidence intervals all decrease at the same rate. (In the latter case, courses with higher variability among grades are updated more frequently than courses with lower variability.) Ideally, of course, the user would not like to pay an overall performance penalty for this fairness. In many cases it may be beneficial to extend the interface so that users can dynamically control the rate at which the running aggregate for each group is updated relative to the others. Such an extension allows users to express partiality in favor of some groups over others. An example of such an interface appears in Figure 3.

2.2 Performance Goals

Minimum Time to Accuracy: In online aggregation, a key performance metric is the time required to produce a useful estimate of the final answer. The definition of a "useful" answer depends, of course, upon the user and the situation. As in traditional systems, some level of accuracy must be reached for an answer to be useful. As in real-time systems, an answer that is a second too late may be entirely useless. Unlike either traditional or real-time systems, some answer is always available, and therefore the definition of "useful" can be based on both kinds of stopping conditions — statistical and real-time — as well as on dynamic and subjective user judgments.

Minimum Time to Completion: It is desirable to minimize the time required to produce the final answer, though this goal is secondary to the performance goal given above. We conjecture that, for large queries, users of an online aggregation typically will terminate processing long before the final answer is produced.

Pacing: The running aggregates should be updated at a regular rate, to guarantee a smooth and continuously improving display. The output rate need not be as regular as that of a video system, for instance, but significant updates should be available often enough to prevent frustration for the user, without being so frequent that they overburden the user or user interface.

3 Building a System for Online Aggregation

We have developed an initial prototype of our ideas in the POSTGRES DBMS. In this section we describe two approaches we followed in trying to add online aggregation to POSTGRES. The first approach was trivial to implement, but suffered from serious deficiencies in both usability and performance. The second approach required significant modifications to POSTGRES internals, but met our goals effectively.

3.1 A Naive Approach

Since POSTGRES already supports arbitrary user-defined output functions, it is possible to use it without modification to produce simple running aggregates like those in Red Brick. Consider Query 3, which requests the average of all grades:

```
Query 3:
SELECT running_avg(final_grade),
       running_confidence(final_grade),
       running_interval(final_grade)
  FROM grades;
```

In POSTGRES, we can write a C function `running_avg` that returns a float by computing the current average after each tuple. We can also write functions `running_confidence` and `running_interval`, based on the statistical results we present in Section 4. Note that the `running_*` functions are not registered as aggregate functions with POSTGRES, but rather as standard user-defined functions. As a result, POSTGRES returns `running_*` values for every tuple that satisfies the `WHERE` clause. In Section 5, we present performance results demonstrating the prohibitive costs of handling all these tuples.

POSTGRES's extensibility features make it convenient for supporting simple running aggregates such as this. Unfortunately, POSTGRES is less useful for more complicated aggregates: since our running functions are not in fact POSTGRES aggregates, they cannot be used with an SQL `GROUP BY` clause. A number of other performance and functionality problems arise in even the most forward-looking of today's database systems, because they are all based on the traditional performance goal of minimizing time to a complete answer. As we present our more detailed approach, it should be clear that it goes much further in meeting our performance and usability goals than this naive solution.

3.2 Modifying a DBMS to Support Online Aggregation

Online aggregation should not be implemented as a user-level addition to a traditional DBMS. In this section, we describe modifications to a database engine to support online aggregation. We have implemented the bulk of of these techniques in POSTGRES, and present some performance results in Section 5.

3.2.1 Random Access to Data

Running aggregates are computed correctly regardless of the order in which records are accessed. However, statistically meaningful estimates of the precision of running aggregates are available only if records are retrieved in random order. Practically speaking, this means that an online aggregation system should avoid access methods in which the attribute values of a tuple affect the order in which the tuple is retrieved. This can be guaranteed in a number of ways:

1. **Heap Scans:** In traditional Heap File access methods, records are stored in an unspecified order, so simple heap scans can be effective for online aggregation. It should be noted, however, that the order of a heap file often does reflect some logical order, based on either the insertion order or some explicit clustering. If this order is correlated with the values of some attributes of the records (as may be the case after a bulk load, or for clustered heap files), an online aggregation system should note that fact in the system statistics, so that online aggregation queries over these attributes can choose an alternative access method.
2. **Index Scans:** Scanning an index returns tuples either in order based on some attributes (e.g. in a B+-tree index), or in groups based on some attributes (e.g. in Hash or multi-dimensional indices). Both of these techniques are inappropriate for online aggregation queries over the indexed attributes. For example, if a column contains 10,000 copies of the value `0`, and 10,000 copies of the value `100`, an ordered or grouped access to the tuples will return wildly skewed online estimates for the average of this column. However, if the attributes that are indexed are not the same as those being aggregated in the query, an index scan should produce an appropriately random access to the values in the attributes that are being aggregated, assuming no correlation between attributes.
3. **Sampling from Indices:** Olken presents techniques for pseudo-random sampling from various index structures [Olk93]. These techniques are ideal for producing meaningful confidence intervals. On the other hand, they can be less efficient than heap scans or even standard index scans, since they require repeated probing of random index buckets, and therefore defeat optimizations like clustering and prefetching.

Heap scans are often the method of choice for large aggregation queries. One of the other access methods may be more appropriate, however, when the heap file is ordered on the aggregation attributes or when it is crucial to have statistically valid running confidence intervals. Our implementation in POSTGRES supports heap scans and index scans; we do not currently support a sampling access method.

3.2.2 Fair, Non-Blocking GROUP BY and DISTINCT

An online aggregation system should begin returning answers as soon as possible. Moreover, if aggregates for multiple groups are being displayed simultaneously, it is often important that the groups receive updates in a fair manner. A traditional technique for grouping is to sort the input relation by the aggregation fields, and then collect the groups by scanning the output of the sort. This presents two problems. First, sorting is a *blocking* algorithm: no outputs can be produced until the entire input has been processed into sorted runs, which can take considerable time. Second, the results for groups are computed in their entirety *one at a time*: the aggregate for the first group is computed to completion before the second group is considered, and so on. Thus sort-based grouping algorithms are inappropriate for online aggregation.

An alternative is to hash the input relation on its grouping columns. Hashing provides a *non-blocking* approach to grouping: as soon as a tuple is read from the input, an updated estimate of the aggregate for its group can be produced. Moreover, groups at the output can be updated as often as one of their constituent records is read from the input. On the other hand, a drawback of hashing is that it does not scale gracefully with the number of grouping values — when the hash table exceeds the size of its associated buffer space, the hashing algorithm will begin to thrash. This problem is alleviated by using unary Hybrid Hashing [Bra84]. It may be expected that the number of distinct groups in a query should be relatively small, and hence naive hashing may be acceptable in many cases. A recent optimization of unary Hybrid Hashing called Hybrid Cache [HN96] guarantees performance that is equivalent to naive hashing for the cases where the hash table fits in memory, and scales gracefully when the hash table grows too large.

SQL supports aggregates of the form *aggregate*(`DISTINCT` *columns*). For such aggregates, the system must remove duplicates from the aggregation columns before computing the aggregate. Grouping and duplicate elimination are very similar, and both can be accomplished via either sorting or hashing. As with grouping, duplicates should be eliminated via hashing in an online aggregation system. In this scenario it is not unusual for the hash table to grow quite large, and techniques like Hybrid Cache can prove very important.

The original version of POSTGRES used sorting to remove duplicates and form groups, so we modified it to do naive hashing for these operations. We plan an implementation of Hybrid Cache in our next online aggregation system.

3.2.3 Index Striding

Even with hash-based grouping, updates to a particular group will be available only as often as constituent records appear in the input of the grouping operator. Given a random delivery of tuples at the input, updates for groups with few members will be very infrequent. To prevent this problem, it would be desirable to read tuples from the input in a round-robin fashion — that is, to provide random delivery of values within each group, but to choose from the groups in order (a tuple from Group 1, a tuple from Group 2, a tuple from Group 3, and so on). To support equal-width confidence intervals or partiality constraints, it may be desirable to use a weighted round-robin scheme that fetches from some groups more often than others.

We support this behavior with a technique called *index striding*. Given a B-tree index on the grouping columns,[1] on the first request for a tuple we open a scan on the leftmost edge of the index, where we find a key value k_1. We assign this scan a search key (or "SARG" [SAC+79]) of the form $[= k_1]$. After fetching the first tuple with key value k_1, on a subsequent request for a tuple we open a second index scan with search key $[> k_1]$, in order to quickly find the next group in the table. When we find this value, k_2, we change the second scan's search key to be $[= k_2]$, and return the tuple that was found. We repeat this procedure for subsequent requests until we have a value k_n such that a search key $[> k_n]$ returns no tuples. At this point, we satisfy requests for tuples by fetching from the scans $[= k_1]$, ... , $[= k_n]$ in a (possibly weighted) round-robin fashion.

With appropriate buffering, striding any index is at least as efficient as scanning a relation via an unclustered index — each tuple of the relation should be fetched exactly once, though each fetch may require a random I/O. This performance is improved if either (i) the index is the primary access method for the relation, (ii) the relation is clustered by the grouping columns, or (iii) the index keys contain both the grouping and aggregation columns, with the grouping columns as a prefix. In all of these cases, the performance of the index stride will be as good as that of scanning a relation via a clustered secondary index: no block of the relation will be fetched more than once.

An important advantage of index striding is that it allows control over delivery of tuples across groups. In particular, it can assure that each group is updated at the output at an appropriate rate based on default settings or online user modifications. A final advantage is that when a user requests that a group be stopped, the other groups will begin to deliver tuples more quickly than they did before.

We extended POSTGRES to support index striding with weighted round-robin scheduling. Using this technique, we support the "Stop-sign" and "Speed" buttons of Figure 3. Index striding supports many of our usability and performance goals.

3.2.4 Non-Blocking Join Algorithms

In order to guarantee reasonably interactive display of online aggregations, it is important to avoid algorithms that block during query processing. In this section we present an initial discussion of standard join algorithms with regard to their blocking properties. We plan to do a quantitative evaluation of these tradeoffs in future work, but this initial analysis already points out some important trends.

Sort-merge join is clearly unacceptable for online aggregation queries, since sorting is a blocking operation. Merge join (without sort) is acceptable in most cases. Complications arise, however, because of the sorted output of a merge join. As with access methods that provide tuples in sorted order, join methods that generate sorted output can cause problems in terms of statistics, and also in terms of fairness in grouping. So merge join is useful in some cases and not others, and must be chosen with care.

Hybrid hash join [DKO+84] blocks for the time required to hash the inner relation. This may be acceptable if the inner relation is small, and particularly if it fits into the buffer space available. The Pipeline hash join technique of [WA91] is a non-blocking hash join that treats its inner and outer relations symmetrically. Pipeline hash join is typically less efficient (in terms of completion time) than hybrid hash join since it shares buffers among both the inner outer relations. However, it may be appropriate for online aggregation if both relations in the join are large.

The "safest" join algorithm for online aggregation is nested-loops join, particularly if there is an index on the inner relation. It is non-blocking, and produces outputs in the same order as the outermost relation. There are recent results on optimizing a pipeline of nested-loops joins to improve the

[1] Index striding is naturally applicable to other types of indices as well, but we omit discussion here due to space constraints.

speed of access to the first few tuples [BM96]. However, with a large, unindexed inner relation, the rate of production of nested-loops join may be so slow (albeit steady), that it will be unacceptable even for online aggregation.

Clearly there are a number of choices for join strategies that satisfy the goals of online aggregation in certain situations. As in traditional query processing, an optimizer must be used to choose between these strategies, and we discuss this issue next.

3.2.5 Optimization

A thorough understanding of query optimization for online aggregation will require (i) a quantitative specification of performance goals for online processing, and (ii) an accurate cost model for relational operators within that framework. We consider our work to date to be too preliminary for such specific analyses. However, some basic observations can vastly improve the quality of plans produced for online aggregation, and we present these points here.

First, sorting can be avoided entirely in an online aggregation system, unless explicitly requested by the user. In scenarios where sorting is quick (e.g. for small relations), alternative algorithms based on hashing or iteration should be comparably fast anyway.

Second, the notion of "interesting orders" [SAC+79] in a traditional optimizer must be extended for online aggregation. As shown in Section 3.2.4, it is undesirable to produce results that are ordered on the aggregation or grouping columns. Hence certain operations (e.g. scans and joins) should be noted to have "interestingly bad" orders, and may often be pruned from the space of possible sub-solutions during optimization.

Third, blocking sub-operations (e.g. processing the inner relation of a Hybrid Hash Join) should have costs that are disproportionate to their processing time. The cost model for an operation in an online aggregation system should be broken into two parts: time t_d spent in blocking operations ("dead time" from the user's perspective), and time t_o spent producing tuples for the output ("output time"). An appropriate cost function for online aggregation should have the form $f(t_o) + g(t_d)$, where f is a linear function, and g is super-linear (e.g. exponential). This will "tax" operations with large amounts of dead time, and may naturally prune inappropriate plans like those that include sorting.

Fourth, some preference should be given to plans that maximize user control, such as those that use index striding. To guarantee this, there must be a way to characterize the controllability features of an operator and weigh the benefit of these features against raw performance considerations.

Finally, tradeoffs need to be evaluated between the output rate of a query and its time to completion. In many cases, the best "batch" plan (e.g. a merge join on sorted relations based on the aggregation attributes) may be so much faster than the best non-blocking plan (e.g. a nested loops join on these relations) that the "batch" behavior may be preferable even in an online environment. The point at which this tradeoff happens is clearly dependent on a user's desires. An interesting direction that we intend to explore in future work is to devise natural controls that allow relatively naive users to set their preferences in this regard. Running multiple versions of a query as in Rdb [AZ96] is a natural way to make this decision on the fly, at the expense of wasted computing resources.

3.2.6 Aggregate Functions

In order to produce running aggregates, the standard set of aggregate functions must be extended. First, aggregate functions must be written that provide running estimates. For single-table queries, running computation of `SUM`, `COUNT`, and `AVG` aggregates is straightforward, and running computation of `VAR` and `STD DEV` aggregates can be accomplished using numerically stable algorithms as in Chan, Golub, and LeVeque [CGL83]. In addition, new aggregate functions must be defined that return running confidence intervals for these estimates. We provide formulae for a variety of such confidence intervals in Section 4 and in the Appendix. Extensible systems like POSTGRES make implementation of these database extensions relatively easy, since new aggregate functions can be added by users. Finally, the query executor must be modified to provide running aggregate values as needed for display, and an API must be provided to control the rate at which the values are provided. We discuss this issue next.

3.2.7 API

The traditional SQL cursor interface is not sufficiently robust to support the kinds of feedback we wish to pass from the user interface to the database server. In an extensible DBMS like POSTGRES, one can circumvent this problem by submitting additional queries, which call user-defined functions, which in turn modify the processing in the DBMS. We introduced four such functions in POSTGRES. The first three are `stopGroup`, `speedUpGroup`, and `slowDownGroup`. Each takes as arguments a cursor and a group value, and is handled accordingly by the backend to stop, speed up, or slow down processing on a group within a query (e.g. by changing the round-robin schedule in an index stride). The fourth function, `setSkipFactor`, takes a cursor name and an integer as arguments, and sets a *skip factor* for the cursor. If the skip factor is set to k, then the DBMS only ships an update to the user interface after k input tuples have been processed by the aggregate. This update frequency can affect both the readability and the performance of the user interface, particularly if the user interface is running on a different machine than the DBMS. Our full user interface for POSTGRES includes a control for the skip factor, and is shown in Figure 4. All the functionality of this interface has been implemented in POSTGRES. The window of the interface grows dynamically as new groups are discovered during processing.

We emphasize that this user interface is merely an example of what can be done with an appropriate system and API. Our solution of using queries with user-defined functions was a (rather inelegant) workaround for the insufficient API provided by SQL. A subsidiary goal of this work is to push for extensions to the SQL API to support interfaces for online control of queries.

4 Running Confidence Intervals

The precision of a running aggregate can be indicated by means of an associated running confidence interval. Suppose that n records have been retrieved in random order and a running aggregate Y_n has been computed. For a prespecified *confidence parameter* $p \in (0, 1)$, the idea, as shown in the examples above, is to display a *precision parameter* ϵ_n such that Y_n is within $\pm\epsilon_n$ of the final answer μ with probability approximately equal to p. Equivalently, the random

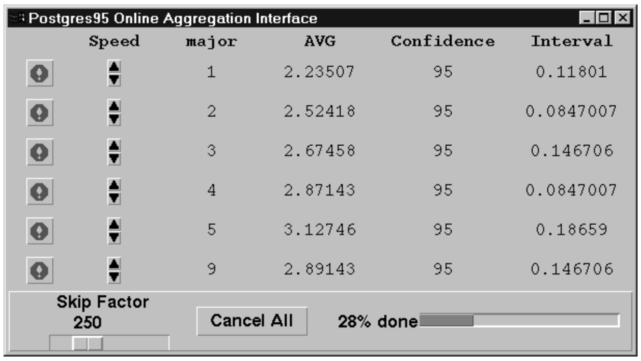

Figure 4: The full POSTGRES online aggregation interface.

interval $[Y_n - \epsilon_n, Y_n + \epsilon_n]$ contains μ with probability approximately equal to p. (In the previous examples of the interface, the confidence parameter p is labeled `Confidence` and the precision parameter ϵ_n is labeled `Interval`.) A large value of ϵ_n serves to warn the user that the records seen so far may not be sufficiently representative of the entire database, and hence the current estimate of the query result may be far from the final result. Moreover, as discussed above, the user can terminate processing of the aggregation query when ϵ_n decreases to a desired level.

A running confidence interval is statistically meaningful provided that records are retrieved in random order. Under this assumption, we can view the records retrieved so far as a random sample drawn uniformly without replacement from the set of all records in the database.

There are several types of running confidence intervals that can be constructed from n retrieved records:

(i) *Conservative* confidence intervals contain the final answer μ with a probability that is guaranteed to be greater than or equal to p. Such intervals can be based on Hoeffding's inequality [Hoe63] or recent extensions [Haa96a] of this inequality and are valid for all $n \geq 1$.

(ii) *Large-sample* confidence intervals contain the final answer μ with a probability approximately equal to p and are based upon central limit theorems (CLT's). Such intervals are appropriate when n is small enough so that the records retrieved so far can be viewed as a sample drawn effectively *with* replacement but large enough so that approximations based on CLT's are accurate. When n is both small enough and large enough, we say that the *large-sample assumption* holds. Such intervals must be used judiciously: the true probability that a large-sample confidence interval contains μ can be less (sometimes much less) than the nominal probability p. The advantage of large-sample confidence intervals is that, when applicable, they are typically much shorter than conservative confidence intervals.

(iii) *Deterministic* confidence intervals contain μ with probability 1. Such intervals are typically useful only when n is very large. Unlike the other types of confidence interval, a deterministic confidence interval is typically of the form $[Y_n - \underline{\epsilon}_n, Y_n + \overline{\epsilon}_n]$ with $\underline{\epsilon}_n \neq \overline{\epsilon}_n$.

In practice, it may be desirable to dynamically adjust the type of running confidence interval that is displayed based on the current value of n.

We illustrate the construction of conservative and large-sample confidence intervals with a simple example. (We conjecture that users typically will terminate an aggregation query before enough records have been retrieved to form a useful deterministic confidence interval; we therefore do not discuss such intervals further.) Let R be a relation containing m tuples, denoted $t_1, t_2, \ldots, t_m$, and consider a query of the form

```
SELECT AVG(expression) FROM R;
```

where *expression* is an arithmetic expression involving the attributes of R. A typical instance of such a query might look like

```
SELECT AVG(price * quantity) FROM inventory;
```

Denote by $v(i)$ $(1 \leq i \leq m)$ the value of *expression* when applied to tuple t_i. Let L_i be the (random) index of the ith tuple retrieved from R; that is, the ith tuple retrieved from R is tuple t_{L_i}. We assume that all retrieval orders are equally likely, so that $P\{L_i = 1\} = P\{L_i = 2\} = \cdots = P\{L_i = m\} = 1/m$ for each i. After n tuples have been retrieved (where $1 \leq n \leq m$), the running aggregate for the above `AVG` query is given by $\overline{Y}_n = (1/n)\sum_{i=1}^{n} v(L_i)$.

To obtain a conservative confidence interval, we require that there exist constants a and b, known *a priori*, such that $a \leq v(i) \leq b$ for $1 \leq i \leq m$; such constants typically can be obtained from the database system catalog. Denote by μ the final answer to the query, that is, $\mu = (1/m)\sum_{i=1}^{m} v(i)$. Hoeffding's inequality [Hoe63] asserts that

$$P\{\,|\overline{Y}_n - \mu| \leq \epsilon\,\} \geq 1 - 2e^{-2n\epsilon^2/(b-a)^2}$$

for $\epsilon > 0$. Setting the right side of the above inequality equal to p and solving for ϵ, we see that with probability $\geq p$ the running average $\overline{Y}_n$ is within $\pm\epsilon_n$ of the final answer μ, where

$$\epsilon_n = (b-a)\left(\frac{1}{2n}\ln\left(\frac{2}{1-p}\right)\right)^{1/2}. \tag{1}$$

To obtain a large-sample confidence interval, we do not require *a priori* bounds on the function v, but rather that the large-sample assumption hold. Since n is "small enough," the random indices $\{\,L_i\colon\ 1 \leq i \leq n\,\}$ can be viewed as a sequence of independent and identically distributed (i.i.d.) random variables. Set $\sigma^2 = (1/m)\sum_{i=1}^{m}\left(v(i) - \mu\right)^2$. Since n is "large enough," it follows from the standard CLT for i.i.d. random variables that $\sqrt{n}(\overline{Y}_n - \mu)/\sigma$ is distributed approximately as a standardized (mean 0, variance 1) normal random variable. By a standard "continuous mapping" argument [Bil86, Section 25], this assertion also holds when σ^2 is replaced by the estimator $T_{n,2}(v) = (n-1)^{-1}\sum_{i=1}^{n}\left(v(L_i) - \overline{Y}_n\right)^2$. It follows that

$$\begin{aligned} P\{\,|\overline{Y}_n - \mu| \leq \epsilon\,\} &= P\left\{\left|\frac{\sqrt{n}(\overline{Y}_n - \mu)}{T_{n,2}^{1/2}(v)}\right| \leq \frac{\epsilon\sqrt{n}}{T_{n,2}^{1/2}(v)}\right\} \\ &\approx 2\Phi\left(\frac{\epsilon\sqrt{n}}{T_{n,2}^{1/2}(v)}\right) - 1 \end{aligned} \tag{2}$$

for $\epsilon > 0$, where Φ is the cumulative distribution function of a standardized normal random variable. Let z_p be the $(p+1)/2$ quantile of this distribution, so that $\Phi(z_p) = (p+1)/2$. Then, setting the rightmost term in (2) equal to p and solving for ϵ, we see that a large-sample $(100p)\%$ confidence interval is obtained by choosing

$$\epsilon_n = \left(\frac{z_p^2 T_{n,2}(v)}{n}\right)^{1/2}. \qquad (3)$$

The above example is relatively simple and utilizes well-known results from probability theory. Often, however, the aggregation query consists of the `AVG`, `SUM`, `COUNT`, `VARIANCE`, or `STD DEV` operator applied not to all the tuples in a given base relation, but to the tuples in a result relation that is specified using standard selection, join, and projection operators. Recent generalizations [Haa96a, Haa96b] of Hoeffding's inequality and the standard CLT permit development of confidence-interval formulas for many of these more complex queries. These formulas are summarized in the Appendix.

5 Performance Issues

In this section, we present initial results from an implementation of online aggregation in POSTGRES; these result illustrate the functionality of the system as well as some performance issues. Our implementation is based on the publicly available Postgres95 distribution [Pos95], Version 1.3. Our measurements were performed with the POSTGRES server running on a DEC3000-M400 with 96Mb main memory, a 1Gb disk, and the DEC OSF/1 V3.2 operating system. The client application, written in Tcl/Tk, was run on an HP PA-RISC 715/80 workstation on the same local network.

For these experiments we used enrollment data from the University of Wisconsin, which represents the enrollment history of students over a three-year period. We focus on a single table, `enroll`, which records information about a student's enrollment in a particular class. The table has 1,547,606 rows, and in POSTGRES occupies about 316.6 Mb on disk.

Our first experiment's query simply finds the average grade of all enrollments in the table:

```
Query 4:
SELECT AVG(grade), 0.99 as Confidence,
       consAvgInterval(0.99) as Interval
  FROM enroll;
```

In addition to the average grade, this query also returns a conservative confidence interval for the average grade; this interval contains the final answer with probability at least 99%. The function `consAvgInterval` is based on the formula for ϵ_n given in (1). Both `AVG` and `consAvgInterval` are aggregate functions registered with POSTGRES, and provide running output during the online aggregation.

Figure 5 shows the results of running the query in various configurations of the system. The vertical bar at 642 seconds represents the time taken for POSTGRES to do traditional "batch" processing. Each of the curves represents the half-width (ϵ_n) of a running interval with 99% confidence, based on a sequential scan of the `enroll` table. In each experiment we varied the "skip factor" described in Section 3.2.7 between 10 and 10000.

The first point to note is that online aggregation is extremely useful: reasonable estimates are available quickly. In addition, these experiments illustrate the need for setting the skip factor intelligently. Our client application is written in Tcl/Tk, an interpreted (and hence rather slow) language; for our experiments it also produces an output trace per tuple displayed. As the skip factor is reduced, the client application becomes overburdened and requests tuples at a much slower rate than they can be delivered by the server; note that the confidence intervals for "skip10" are about an order of magnitude wider than those for "skip1000" and "skip10000". A naive implementation of online aggregation, as suggested in Section 3.1, would correspond to a skip-factor setting of 1. Such an implementation would have very poor performance, shipping and displaying as many rows as there are in `enroll`.

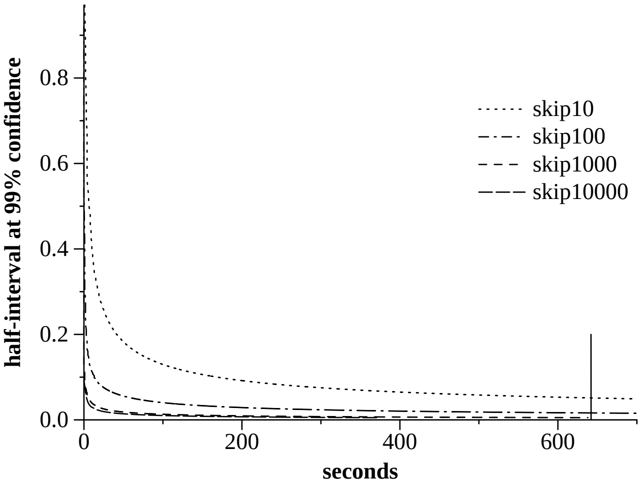

Figure 5: Half-width (ϵ_n) of conservative confidence interval for Query 4.

Our second experiment uses a similar query, which requests the average grade per "college" in the university.

```
Query 5:
SELECT college, AVG(grade),
       0.95 as Confidence,
       consAvgInterval(0.95) as Interval,
  FROM enroll
GROUP BY college;
```

Note that in this query we choose a lower confidence (95%), which allows us to get smaller intervals somewhat more quickly. There are 16 values in the college column. In Figure 6 we present performance for a large group (college=L, 925596 tuples), and for a small group (college=S, 15619 tuples), using a variety of query plans. In each graph, we measure the half-width of the confidence interval over time for (1) a sequential scan, (2) a clustered index stride, (3) an unclustered index stride, and (4) a clustered index stride in which all groups but the one measured are stopped early by pressing the "stop-sign" button soon after the query begins running ("L only" and "S only" in Figures 6 and 7).

Clearly, index stride is faster for clustered indices than for unclustered ones, since the number of heap-file I/Os is reduced by clustering. More interestingly, note that when some groups are stopped during index striding, the groups that remain are computed faster; this is reflected in the steeper decline of "clustered: L only" and "clustered: S only" relative to the corresponding "clustered" curves. Perhaps the most interesting aspect of these graphs is the difference between sequential scanning and index striding. Sequential scanning retrieves tuples faster than index striding, at the cost of lack of control. For the large group (`L`), the superior speed of sequential scanning is reflected in the rate at which the half-width of the confidence interval decreases. However, for the smaller group (`S`), *even the unclustered index stride drops more steeply than sequential scan.* This is

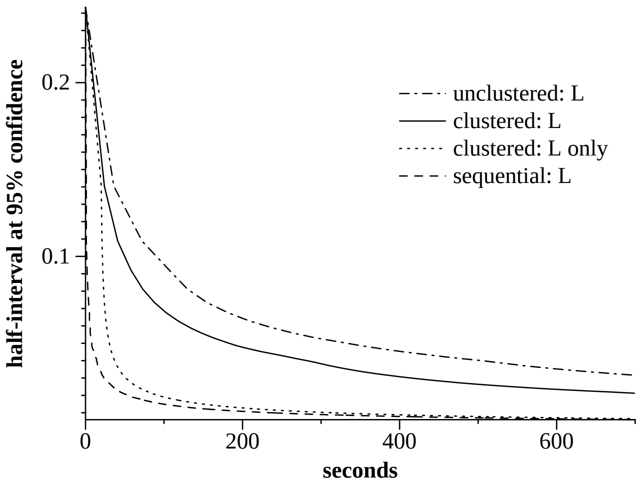

Figure 6: Half-width (ϵ_n) of conservative confidence interval for Query 5, large group.

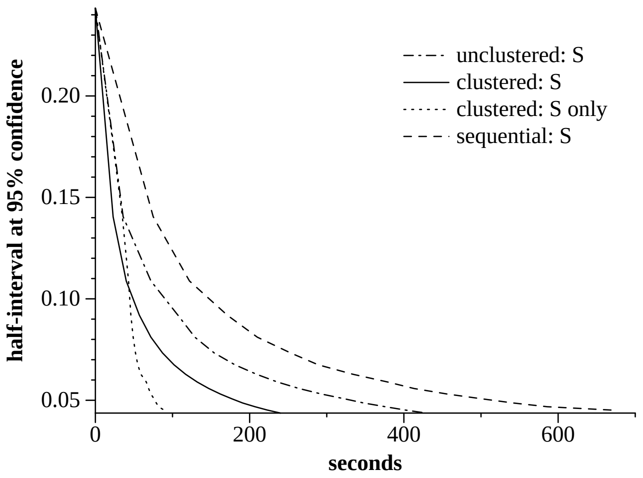

Figure 7: Half-width (ϵ_n) of conservative confidence interval for Query 5, small group.

due to the fact that tuples from group S appear fairly rarely in the relation. Sequential scan provides these tuples only occasionally, while index striding — even when no groups are stopped — fetches tuples from S on a regular basis as part of its round-robin schedule. This highlights an additional advantage of index striding: it provides faster estimates for small groups than access methods that provide random arrivals of tuples.

These experiments provide some initial insights into our techniques for online aggregation, and serve as evidence that our approach to online aggregation provides functionality and performance that would not be available in naive solutions. There is clearly much additional work to be done in measuring the costs and benefits of the various techniques proposed here. We reserve such issues for future study.

6 Conclusion and Future Work

In this paper we demonstrate the need for a new approach to aggregation that is interactive, intuitive, and user-controllable. Supporting this online approach to aggregation requires significant extension to a relational database engine. As a prototype implementation, we extended POSTGRES with aggregates that produce running output, hash-based grouping and duplicate-elimination, index striding, minor optimization changes, new API's and user interfaces. Based on these extensions we developed a relatively attractive system that satisfies many of the performance and usability goals we set out to solve.

An important feature of a user interface for online aggregation is the ability to produce statistical confidence intervals for running aggregates. This paper indicates how such confidence intervals can be implemented in any DBMS that stores rudimentary statistics such as minimum and maximum values per column.

As we have noted, the usability and performance needs of online aggregation are not crisply defined, and there is much latitude in the solution space for the problem. We intend to visit more issues in more detail in our next phase of development, which will be done in the context of a commercial parallel object-relational DBMS. We conclude by listing some directions we are considering for future work:

- **User Interface:** Online aggregation is motivated by the need for better user interfaces, and it is clear that additional work is needed in this area. One direction we plan to pursue is to present running plots of queries on a 2-dimensional canvas, as exemplified by the (batch) visualization system Tioga DataSplash [ACSW96]. In such a system, one can view the screen as a "graphical aggregate" — many data items are aggregated into one progressively refined image. Techniques for storing and presenting progressive refinements of images are well understood [VU92] and exploited by popular web browsers. It would be interesting to try to find common ground between the techniques presented here, and the image compression techniques used for progressive network delivery.

  Another interface problem is to present "just enough" information on screen. In current OLAP systems, this is typically handled by presenting the input data in a small number of default aggregate groups, and then allowing "drill-down" and "roll-up" facilities. We hope to combine this interface with online processing so that drill-down is available instantly, with super-aggregates being continuously computed in the background while users drill into *ad hoc* sub-aggregates that are computed more quickly.

- **Nested Queries:** An open question is how to provide online execution of queries containing aggregations in both subqueries and outer-level queries: the running results at the top level depend on the running results at lower levels. Traditional block-at-a-time processing requires the lower query blocks to be processed before the higher ones, but this is a blocking execution model,[2] and hence violates our performance goals. Any non-blocking approach would lead to significant statistical problems in terms of confidence intervals, in addition to complicating other performance and usability issues. An additional question is how processing is best time-sliced across the various query blocks, in both uniprocessor and parallel configurations.

- **Control Without Indices:** As of now, we can provide maximal user control only when we have an appropriate index to support index striding. We are considering techniques for providing this control in other

[2] No pun intended.

scenarios as well. For example, in order to provide partiality in aggregates over joins, it may be beneficial to effectively scan base relations multiple times, each time providing a different subset of the relation for join processing; the early subsets can contain a preponderance of tuples from the preferred groups. The functionality of multiple scans can be efficiently achieved via "piggy-back" schemes which allow more than one cursor to share a single physical scan of the data. Another possibility is to recluster heaps on the fly to support more desirable access orders on subsequent rescans.

- **Checkpointing and Continuation:** Aggregation queries that benefit from online techniques will typically be long-running operations. As a result they should be checkpointed, so that computation can be saved across system crashes, power failures, and operator errors. This is particularly natural for online aggregation queries: users should be allowed to "continue" queries (or pieces of queries) that they have previously stopped. Checkpoints of partially computed queries can also be used as materialized sample views [Olk93].
- **Tracking Online Queries:** Although users may often stop aggregation processing early, they may also want to make use of the actual tuples used to compute the partial aggregate. This is a common request in the context of, for example, financial auditing or statistical quality control: an unusual value of an online aggregate produced from a sample population may indicate the need to study that population in more detail. In order to support such query tracking, one must generate a relation, RID-list, or view while processing the aggregation online. Techniques for doing this efficiently will depend on the query.
- **Extensions of Statistical Results:** We are actively working on confidence intervals for additional aggregate functions. In addition, we are developing techniques to provide "simultaneous" confidence intervals, which can describe the statistical accuracy of the estimations for all groups at once, complementing the confidence interval per group. Finally, the statistical techniques in this paper assume accurate statistical information in the system catalogs, particularly regarding the cardinalities of relations. An extension we are pursuing is to provide confidence intervals that can tolerate a certain amount of error in these stored statistics.

7 Acknowledgments

Index Striding was inspired by a comment made by Mike Stonebraker. Andrew MacBride implemented the Postgres95 Online Aggregation Interface. Thanks are due to Jeff Naughton, Praveen Seshadri, Donald Kossman, and Margo Seltzer for interesting discussion of this work. Thanks also to the following for their editorial suggestions: Alex Aiken, Mike Carey, Alice Ford, Wei Hong, Navin Kabra, Marcel Kornacker, Bruce Lindsay, Adam Sah, Sunita Sarawagi, our anonymous reviewers, and the students of CS286, UC Berkeley, Spring 1996. The Wisconsin course dataset was graciously provided by Bob Nolan of UW-Madison's Department of Information Technology (DoIT). Hellerstein and Wang were partially funded by a grant from Informix Corporation.

References

[AAD+96] A. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. 22nd Intl. Conf. Very Large Data Bases*, Mumbai(Bombay), September 1996.

[ACSW96] A. Aiken, J. Chen, M. Stonebraker, and A. Woodruff. Tioga-2: A direct manipulation database visualization environment. In *Proc. of the 12th Intl. Conf. Data Engineering*, pages 208–217, New Orleans, February 1996.

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

[Bil86] P. Billingsley. *Probability and Measure*. Wiley, New York, second edition, 1986.

[BM96] R. J. Bayardo, Jr. and D. P. Miranker. Processing queries for first-few answers. In *Fifth Intl. Conf. Information and Knowledge Management*, pages 45–52, Rockville, Maryland, 1996.

[Bra84] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proc. 10th Intl. Conf. Very Large Data Bases*, pages 323–333, Singapore, August 1984.

[CCS93] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP (on-line analytical processing) to user-analysts: an IT mandate. URL http://www.arborsoft.com/papers/coddTOC.html., 1993.

[CGL83] T. F. Chan, G. H. Golub, and R. J. LeVeque. Algorithms for computing the sample variance: Analysis and recommendation. *Amer. Statist.*, 37:242–247, 1983.

[CS96] S. Chaudhuri and K. Shim. Optimizing queries with aggregate views. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in Database Technology– EDBT'96 5th Intl. Conf. on Extending Database Technology*, volume 1057 of *Lecture Notes in Computer Science*, pages 167–182. Springer-Verlag, New York, 1996.

[DKO+84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. ACM-SIGMOD Intl. Conf. Management of Data*, pages 1–8, Boston, June 1984.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing Group-By, Cross-Tab, and Sub-Totals. In *Proc. of the 12th Intl. Conf. Data Engineering*, pages 152–159, 1996.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. 21st Intl. Conf. Very Large Data Bases*, Zurich, September 1995, pages 358–369.

[Haa96a] P. J. Haas. Hoeffding inequalities for join-selectivity estimation and online aggregation. IBM Research Report RJ 10040, IBM Almaden Research Center, San Jose, CA, 1996.

[Haa96b] P. J. Haas. Large-sample and deterministic confidence intervals for online aggregation. IBM Research Report RJ 10050, IBM Almaden Research Center, San Jose, CA, 1996.

[HN96] J. M. Hellerstein and J. F. Naughton. Query execution techniques for caching expensive methods. In *Proc. ACM-SIGMOD Intl. Conf. Management of Data*, Montreal, June 1996, pages 423–424.

[HNSS96] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Selectivity and cost estimation for joins based on random sampling. *J. Comput. System Sci.*, 52:550–569, 1996.

[HOD91] W.-C. Hou, G. Ozsoyoglu, and E. Dogdu. Error-constrained count query evaluation in relational databases. In *Proceedings, 1991 ACM-SIGMOD Intl. Conf. Managment of Data*, pages 278–287. ACM Press, 1991.

[Hoe63] W. Hoeffding. Probability inequalities for sums of bounded random variables. *J. Amer. Statist. Assoc.*, 58:13–30, 1963.

[HOT88] W.-C. Hou, G. Ozsoyoglu, and B. K. Taneja. Statistical estimators for relational algebra expressions. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 276–287, Austin, March 1988.

[HOT89] W.-C. Hou, G. Ozsoyoglu, and B. K. Taneja. Processing aggregate relational queries with hard time constraints. In *Proc. ACM-SIGMOD Intl. Conf. Management of Data*, pages 68–77, Portland, May-June 1989.

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

[Mye85] B. A. Myers. The importance of percent-done progress indicators for computer-human interfaces. In *Proc. SIGCHI '85: Human Factors in Computing Systems*, pages 11–17, April 1985.

[Olk93] F. Olken. *Random Sampling from Databases.* PhD thesis, University of California, Berkeley, 1993.

[Pos95] Postgres95 home page, 1995. URL http://www.ki.net/postgres95.

[SAC+79] P. G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. ACM-SIGMOD Intl. Conf. Management of Data*, pages 22–34, Boston, June 1979.

[SHP+96] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *Proc. ACM-SIGMOD Intl. Conf. Management of Data*, Montreal, June 1996, pages 435–446.

[SPL96] P. Seshadri, H. Pirahesh, and T. C. Leung. Complex query decorrelation. In *Proc. 12th IEEE Intl. Conf. Data Engineering*, New Orleans, February 1996.

[VU92] M. Vetterli and K. M. Uz. Multiresolution coding techniques for digital video: A review. *Multidimensional Systems and Signal Processing*, 3:161–187, 1992.

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

[WA91] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. First Intl. Conf. Parallel and Distributed Information Systems*, pages 68–77, Dec 1991.

[YL95] W. P. Yan and P.-A. Larson. Eager aggregation and lazy aggregation. In *Proc. 21st Intl. Conf. Very Large Data Bases*, Zurich, September 1995, pages 345–357.

Appendix: Formulas for Running Confidence Intervals

In this appendix we provide formulas that can be used to compute conservative and large-sample confidence intervals for a variety of aggregation queries encountered in practice. Throughout, we fix the confidence parameter p and give formulas for the precision parameter ϵ_n.

We first consider queries of the form

SELECT *op*(*expression*) **FROM** R **WHERE** *predicate*;

where *op* is one of **COUNT**, **SUM**, **AVG**, **VARIANCE**, or **STD DEV**, *expression* is an arithmetic expression as before, and *predicate* is an arbitrary predicate involving the attributes of R. When *op* is equal to **COUNT**, we assume for simplicity that *expression* is equal to *, that is, the "value" of the expression is equal to 1 for all tuples. (Null values can be handled by modifying *predicate*, and counts of distinct values can be handled as described below.) As in Section 4, relation R consists of tuples $t_1, t_2, \ldots, t_m$.

If *op* is equal to **COUNT** or **SUM**, the formulas in (1) and (3) apply, provided we take $v(i)$ equal to m times the value of *expression* when applied to tuple t_i if tuple t_i satisfies *predicate* and $v(i) = 0$ otherwise.

To handle the remaining operators, namely **AVG**, **VARIANCE**, and **STD DEV**, we proceed as follows. As in Section 4, let $v(i)$ be the value of *expression* when applied to tuple t_i, L_i be the random index of the ith tuple retrieved from R, and a, b be *a priori* bounds on the function v. Denote by S ($\subseteq R$) the set of tuples that satisfy *predicate*, and set $u(i) = 1$ if $t_i \in S$ and $u(i) = 0$ otherwise. After n tuples have been retrieved, the running aggregates for an AVG, VARIANCE, and STD DEV query are given by

$$\overline{Y}_n(S) = \frac{1}{I_n} \sum_{i=1}^{n} uv(L_i),$$

$$Z_n(S) = \frac{1}{I_n - 1} \sum_{i=1}^{n} u(L_i) \left(v(L_i) - \overline{Y}_n(S)\right)^2,$$

and $\sqrt{Z_n(S)}$, respectively, where $I_n = \sum_{i=1}^n u(L_i)$ and the function uv is defined by $uv(i) = u(i)v(i)$. We assume throughout that $I_n > 1$.

Set $\theta_0(a,b) = (|a| \vee b)(|a| + b) - 0.25(|a| \vee b)^2$,

$$\theta(a,b) = \begin{cases} \dfrac{8}{(b-a)^4} & \text{if } 0 \le a < b \text{ or } a < b \le 0; \\ \max\left(\dfrac{8}{(b-a)^4}, \dfrac{2}{\theta_0^2(a,b)}\right) & \text{if } a < 0 < b, \end{cases}$$

and

$$B_n = \frac{1}{\theta(a,b)\lfloor I_n/2 \rfloor} \ln\left(\frac{2}{1-p}\right),$$

where $x \vee y = \max(x,y)$ and $\lfloor x \rfloor$ is the greatest integer $\le x$. Also set $T_n(f) = (1/n)\sum_{i=1}^n f(L_i)$,

$$T_{n,q}(f) = \frac{1}{n-1} \sum_{i=1}^{n} \left(f(L_i) - T_n(f)\right)^q$$

and

$$T_{n,q,r}(f,g) = \frac{1}{n-1} \sum_{i=1}^{n} \left(f(L_i) - T_n(f)\right)^q \left(g(L_i) - T_n(g)\right)^r,$$

where f and g are arbitrary real-valued functions defined on $\{1, 2, \ldots, m\}$. Finally, set

$$G_n = T_{n,2}(uv) - 2R_{n,2}T_{n,1,1}(uv,u) + R_{n,2}^2 T_{n,2}(u)$$

and

$$\begin{aligned} G_n' &= T_{n,2}(uv^2) - 4R_{n,2}T_{n,1,1}(uv^2, uv) \\ &+ (4R_{n,2}^2 - 2R_{n,1})T_{n,1,1}(uv^2, u) + 4R_{n,2}^2 T_{n,2}(uv) \\ &+ (4R_{n,1}R_{n,2} - 8R_{n,2}^3)T_{n,1,1}(uv, u) + (2R_{n,2}^2 - R_{n,1})^2 T_{n,2}(u), \end{aligned}$$

where $R_{n,1} = T_n(uv^2)/T_n(u)$, $R_{n,2} = T_n(uv)/T_n(u)$, and $uv^2(i) = u(i)\left(v(i)\right)^2$.

Using this notation, Table 1 gives formulas for the precision constant ϵ_n in conservative and large-sample confidence intervals; these formulas are derived in [Haa96a, Haa96b]. The quantity s that appears in the formulas is a lower bound for the (unknown)

type	AVG	VARIANCE	STD DEV
conserv.	$(b-a)\left(\frac{1}{2I_n}\ln(\frac{2}{1-p})\right)^{1/2}$	$\frac{s(b-a)^2}{4(s-1)^2}+\frac{sB_n^{1/2}}{s-1}$	$\frac{s(b-a)^2}{4(s-1)^2}+\frac{sB_n^{1/4}}{s-1}$
lg-sample	$\left(\frac{z_p^2 G_n}{nT_n^2(u)}\right)^{1/2}$	$\left(\frac{z_p^2 G'_n}{nT_n^2(u)}\right)^{1/2}$	$\left(\frac{z_p^2 G'_n}{4nZ_n(S)T_n^2(u)}\right)^{1/2}$

Table 1: Formulas for the precision parameters ϵ_n: one table.

quantity $|S|$; the larger the lower bound s, the narrower the resulting conservative confidence interval. Note that we can set $s = I_n$ if I_n is sufficiently large. When *predicate* is empty, so that $u(i) = 1$ for $1 \le i \le M$, then s can be taken as n in the second and third entries in the first row of the table. Moreover, in each of these entries the first term in the sum can be discarded.

The formulas in Table 1 also apply to queries of the form

```
SELECT op(expression) FROM R WHERE predicate
GROUP BY attributes
```

For the group with *attribute* value equal to x, use these formulas with $u(i) = 1$ if tuple t_i satisfies *predicate* and t_i.*attribute* $= x$, and $u(i) = 0$ otherwise. The formulas also can easily be modified to handle queries of the form

```
SELECT op(DISTINCT expression) FROM R WHERE predicate;
```

The idea is to set $U'_i = 1$ if $t_{L_i} \in S$ and $v(L_i) \neq v(L_j)$ for $1 \le j \le i-1$; otherwise, set $U'_i = 0$. The formulas in Table 1 then hold with $u(L_i)$ replaced by U'_i, $uv(L_i)$ replaced by $U'_i v(L_i)$, and I_n replaced by $I'_n = \sum_{i=1}^n U'_i$.

We next consider queries of the form

```
SELECT op(expression) FROM R1, R2, ..., RK
 WHERE predicate;
```

where $K > 1$, *op* is one of `COUNT`, `SUM`, or `AVG`, *expression* is an arithmetic expression, and *predicate* is an arbitrary predicate involving the attributes of input relations R_1 through R_K. As before, *expression* is always equal to `*` when *op* is equal to `COUNT`. Usually, *predicate* is a conjunction of join and selection predicates. A typical instance of such a query might look like

```
SELECT SUM(supplier.price * inventory.quantity)
  FROM supplier, inventory
 WHERE supplier.part_number = inventory.part_number
       AND inventory.location = 'San Jose';
```

For $1 \le k \le K$, denote the tuples in R_k by $t_{k,1}, t_{k,2}, \ldots, t_{k,m_k}$, where m_k is the number of tuples in R_k. Set $v(i_1, i_2, \ldots, i_K)$ equal to α times the value of *expression* when applied to tuples $t_{1,i_1}, t_{2,i_2}, \ldots, t_{K,i_K}$, where $\alpha = 1$ if *op* is equal to `AVG` and $\alpha = m_1 m_2 \cdots m_K$ if *op* is equal to `COUNT` or `SUM`. Denote by S the subset of $R_1 \times R_2 \times \cdots \times R_K$ such that $(t_{1,i_1}, t_{2,i_2}, \ldots, t_{K,i_K}) \in S$ if and only if these tuples jointly satisfy *predicate*. Set $u(i_1, i_2, \ldots, i_K) = 1$ if $(t_{1,i_1}, t_{2,i_2}, \ldots, t_{K,i_K}) \in S$ and $u(i_1, i_2, \ldots, i_K) = 0$ otherwise. As before, let a, b be *a priori* bounds on the function v.

For each relation R_k, we assume that tuples are retrieved in random order, independently of the retrieval order for the other relations. Denote by $L_{k,i}$ the random index of the ith tuple retrieved from relation R_k. Suppose that n tuples have been retrieved from relation R_k for $1 \le k \le K$, where $1 \le n \le \min_{1 \le k \le K} m_k$. (See [Haa96a, Haa96b] for extensions to the case in which n_k tuples are retrieved from R_k for $1 \le k \le K$ and $n_k \neq n_{k'}$ for some k, k'.) The running aggregate for a `COUNT` or `SUM` query is given by $\tilde{Y}_n = \tilde{T}_n(uv)$, where

$$\tilde{T}_n(f) = \frac{1}{n^K} \sum_{i_1=1}^n \sum_{i_2=1}^n \cdots \sum_{i_K=1}^n f(L_{1,i_1}, L_{2,i_2}, \ldots, L_{K,i_K})$$

type	SUM/COUNT	AVG
conserv.	$(b-a)\left(\frac{1}{2n}\ln(\frac{2}{1-p})\right)^{1/2}$	–
lg-sample	$\left(\frac{z_p^2 \tilde{T}_{n,2}(uv)}{n}\right)^{1/2}$	$\left(\frac{z_p^2 \tilde{G}_n}{n\tilde{T}_n^2(u)}\right)^{1/2}$

Table 2: Formulas for the precision parameter ϵ_n: K tables.

and the definition of the function uv is analogous to the definition for the single-table case. The running aggregate for an AVG query is given by $\tilde{Y}_n(S) = \tilde{T}_n(uv)/\tilde{T}_n(u)$.

Set

$$\tilde{T}_n(f;k,j) = \frac{1}{n^{K-1}} \sum_{i_1=1}^n \cdots \sum_{i_{k-1}=1}^n \sum_{i_{k+1}=1}^n \cdots \sum_{i_K=1}^n L_{1,i_1}, \ldots, L_{k-1,i_{k-1}}, L_{k,j}, L_{k+1,i_{k+1}}, \ldots, L_{K,i_K}),$$

$$\tilde{T}_{n,q}(f) = \sum_{k=1}^K \left(\frac{1}{(n-1)} \sum_{j=1}^n \left(\tilde{T}_n(f;k,j) - \tilde{T}_n(f) \right)^q \right),$$

and

$$\tilde{T}_{n,q,r}(f,g) = \sum_{k=1}^K \left(\frac{1}{(n-1)} \sum_{j=1}^n \left(\tilde{T}_n(f;k,j) - \tilde{T}_n(f) \right)^q \left(\tilde{T}_n(g;k,j) - \tilde{T}_n(g) \right)^r \right).$$

Also set $\tilde{R}_n = \tilde{T}_n(uv)/\tilde{T}_n(u)$ and

$$\tilde{G}_n = \tilde{T}_{n,2}(uv) - 2\tilde{R}_n \tilde{T}_{n,1,1}(uv,u) + \tilde{R}_n^2 \tilde{T}_{n,2}(u).$$

Using this notation, Table 2 gives formulas for the precision constant ϵ_n in conservative and large-sample confidence intervals. As above, the formulas are derived in [Haa96a, Haa96b]. As can be seen, there is currently no formula available for conservative confidence intervals corresponding to AVG queries with selection predicates. (Actually, when there are no predicates, so that the average is being taken over a cross-product of the input relations, the formula in Table 2 for COUNT and SUM queries applies. This case is uncommon in practice, however.)