The Case for Online Aggregation:

New Challenges in User Interfaces, Performance Goals, and DBMS Design

Joseph M. Hellerstein
University of California, Berkeley
EECS Computer Science Division
387 Soda Hall #1776
Berkeley, CA 94720-1776
Phone: 510/643-4011   Fax: 510/642-5615

jmh@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 an accurate answer is returned. Batch mode processing has long been unacceptable to users. In this paper we describe the need for online *aggregation processing, in which aggregation operators provide ongoing feedback, and are controllable during processing. We explore a number of issues, including both user interface needs and database technology required to support those needs. We describe new usability and performance goals for online aggregation processing, and present techniques for enhancing current relational database systems to support online aggregation.*

1. Introduction

Aggregation is an increasingly important operation in today's relational database systems. As data sets grow larger, and users (and their interfaces) become more sophisticated, there is an increasing 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, aggregate processing in today's database systems closely resembles the offline batch processing of the 1960's. When users submit an aggregate query to the system, they are forced to wait without feedback while the system churns through thousands or millions of tuples. Only after a significant period of time does the system respond with the small answer desired. 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 executed with painstaking accuracy, even in situations where an acceptably accurate approximation might be available very quickly.

The time has come to change the interface to aggregate processing. Aggregation must be performed *online*, 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 database system to support online aggregation. This involves not only changes to user interfaces, but also corresponding changes to database query processing, optimization, and statistics, which are required to support the new functionality efficiently. We draw significant distinctions between online aggregation and previous proposals, such as database sampling, for solving this problem. Many new techniques will clearly be required to support online aggregation, but it is our belief that the desired functionality and performance can be supported via an evolutionary approach. As a result our discussion is cast in terms of the framework of relational database systems.

1.1 A Motivating Example

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

```
Query 1:
SELECT AVG(final_grade)
  FROM grades
 WHERE course_name = 'CS101';
```

If there is no index on the "course_name" attribute, this query scans the entire grades table before returning an answer. After an extended period of time, the database produces the correct answer:

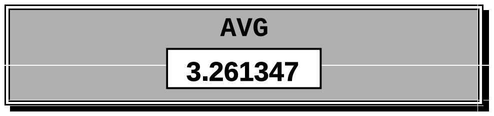


Figure 1: A Traditional Output Interface

As an alternative, consider the following user interface that could appear immediately after the user submits the query:

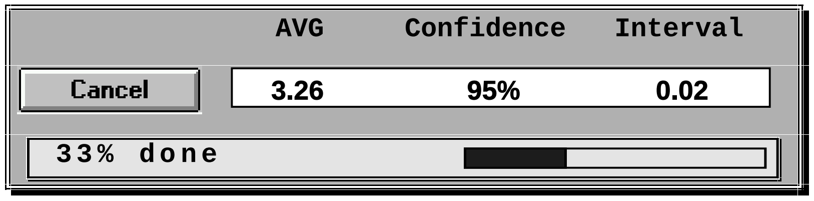
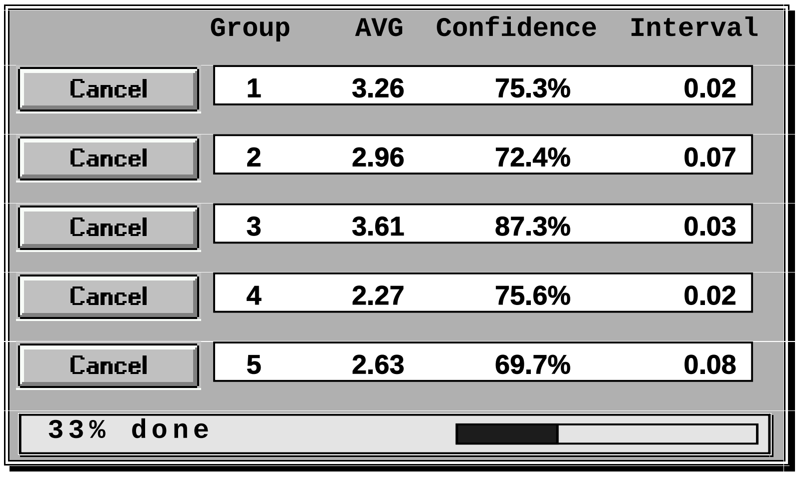


Figure 2: An Online Aggregation Output Interface

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 **AVG** field shows the *running aggregate, i.e.,* the aggregation value that would be returned if no more tuples were found that satisfied the `WHERE` clause. The **Confidence** and **Interval** fields give a statistical estimation of the proximity of the current running aggregate to the final result — in the example above, statistics tells us that with 95% probability, the current average is within .02 of the final result. The **% done** and "growbar" display give an indication of the amount of processing remaining before completion. If the query completes before the "Cancel" button is pressed, the final result can be displayed without any statistical information.

This interface is significantly more useful than the "blinking cursor" or "wristwatch icon" traditionally presented to users during aggregation. It presents information at all times, and more importantly *it gives the user control over the 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, impatience, accuracy needs, and priority of other tasks. Since the user sees the ongoing processing, there is no need to quantify these factors either in advance or in any concrete manner.



	Group	AVG	Confidence	Interval
Cancel	1	3.26	75.3%	0.02
Cancel	2	2.96	72.4%	0.07
Cancel	3	3.61	87.3%	0.03
Cancel	4	2.27	75.6%	0.02
Cancel	5	2.63	69.7%	0.08

33% done

Figure 3: A Multi-Group Online Aggregation Output Interface

Obviously this example is quite simple; more complex examples will be presented below. However, note that even in this very simple example the user is being given considerably more control over the system than was previously available. The interface — and the underlying processing required to support it effectively — must get more powerful as the queries get more complex. In the rest of the paper we highlight additional ways that a user can control aggregation, and we discuss a number of system issues that need to be addressed in order to best support this sort of control.

1.2 Online Aggregation: More than Sampling

The concept of trading accuracy for efficiency in a database system is not a new one: a large body of work on database sampling has been devoted to this problem. The sampling work closest in spirit to this paper focuses on returning approximate answers to aggregate queries [HOT88, HOT89] and other relational queries [OR86, Olk93, etc.] Online aggregation is different than traditional database sampling in number of ways — particularly in its interface, but also in its architecture and statistical methods. In this section we focus on the interface distinctions between sampling and online aggregation; discussion of internal techniques is deferred until Sections 4 and 5.

Given a user's query, database sampling techniques compute a portion of the query's answer until some "stopping condition" is reached. When this condition is reached, the current running aggregate is passed to the output, along with statistical information as to its probable accuracy. The stopping condition is specified *before* query processing begins, and can be either a statistical constraint (*e.g.* "get within 2% of the actual answer with 95% probability") or a "real-time" constraint (*e.g.* "run for 5 minutes only".)

Online aggregation provides this functionality along with much more. Stopping conditions are easily achieved by a user in an online aggregation system, simply by canceling processing at the appropriate accuracy level or time. Online aggregation systems provide the user more control than sampling systems however, since stopping conditions can be chosen or modified *while the query is running*. Though this may seem a simple point, consider the case of an aggregation query with 5 groups in its output, as in Figure 3. In an online aggregation system, the user can be presented with 5 outputs and 5 "Cancel" buttons. In a sampling system, the user does not know the output groups *a priori*, and hence cannot control the query in a group-by-group fashion. The interface of online aggregation can thus be strictly more powerful than that of sampling.

Another significant advantage of online aggregation interfaces is that users get *ongoing feedback* on a query's progress. This allows intuitive, non-statistical insight into the progress of a query. It also allows for ongoing non-textual, non-statistical representations of a query's output. One common example of this is the appearance of points on a map or graph as they are retrieved from the database. While online aggregation allows the user to observe points being plotted as they are processed, sampling systems are essentially just faster batch systems — they do not produce any answers until they are finished, and thus in a basic sense they do not improve the user's interface.

Perhaps the most significant advantage of online aggregation is that its interface is far more natural and easy to use than that of sampling. Busy end-users are likely to be quite comfortable with the online aggregation "Cancel" buttons, since such interfaces are familiar from popular tools like web browsers, which display images in an incremental fashion [VM92]. 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 and changeable — often another minute or two of processing "suddenly" becomes worthwhile at the last second.

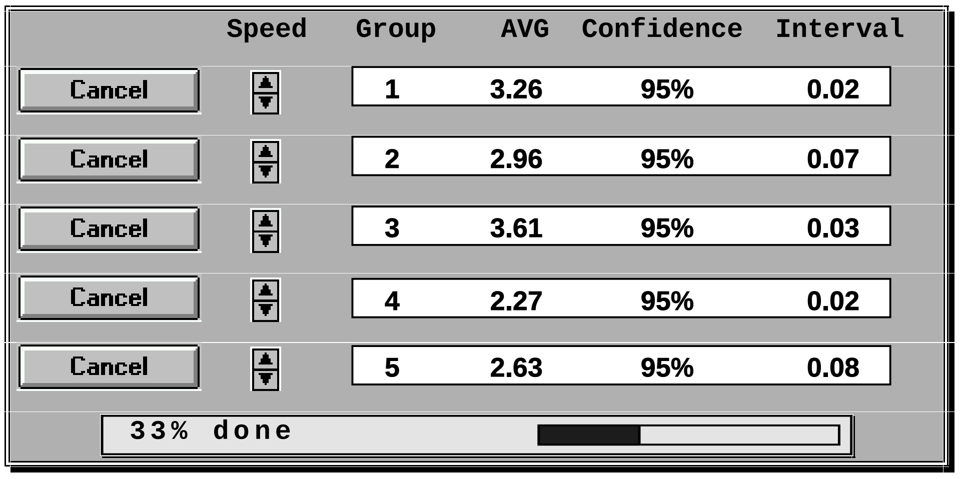
The familiarity and naturalness of the online aggregation interface cannot be overemphasized. It is crucial to remember that user frustration with batch processing is the main motivation for efficiency/accuracy tradeoffs such as sampling and online aggregation. As a result, the interface for these tradeoffs must be as simple and attractive as possible for users. Developers of existing sampling techniques have missed this point, and user-level sampling techniques have not caught on in industrial systems[1].

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 complex super-aggregation ("roll-up"), sub-aggregation ("drill-down") and cross-tabulation. The CUBE operator [GBLP96] has been proposed as an SQL addition to allow standard relational systems to support these kinds of aggregation. It seems fairly clear that computing CUBE queries will often require extremely complex processing, 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; approximations of the various aggregates are likely to suffice in numerous situations. The original motivation for CUBE queries and OLAP systems was to allow decision-makers in companies to browse through large amounts of data, looking for aggregate trends and anomalies in an ad-hoc and interactive fashion. Batch processing is not interactive, and hence inappropriate for browsing. OLAP systems with online aggregation facilities can allow users the luxury of browsing their data with truly continuous feedback, the same way that they can currently browse the world-wide web. This "instant gratification" encourages user interaction, patience, and perseverance, and is an important human factor that should not be overlooked.

Other recent work on aggregation in the relational database research community has focused on new transformations for optimizing queries with aggregation [CH96, GHQ96, YL96, SPL96]. 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.

[1] On the other hand, sampling techniques may prove useful in a less user-intensive aspect of industrial systems: cost estimation and database statistics for query optimization [LNSS93, HNSS95, etc.]



	Speed	Group	AVG	Confidence	Interval
Cancel	▲▼	1	3.26	95%	0.02
Cancel	▲▼	2	2.96	95%	0.07
Cancel	▲▼	3	3.61	95%	0.03
Cancel	▲▼	4	2.27	95%	0.02
Cancel	▲▼	5	2.63	95%	0.08

33% done

Figure 4: A Speed-Controllable Multi-Group Online Aggregation Output Interface

2. Usability and Performance Goals

Traditional metrics of performance are inappropriate for online aggregation systems, since the usability goals in online aggregation are different than those in both traditional and real-time database systems. In online aggregation, the key performance metrics are response time and throughput for *useful estimations*

Query 2:

```
SELECT AVG(final_grades)
FROM grades
GROUP BY course_name;
```

to an answer, rather than response time and throughput of a completely accurate answer. The definition of "useful", of course, depends 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" depends on both kinds of stopping conditions — statistical and real-time — as well as on dynamic and subjective user judgments.

In addition to the time to a useful estimation, an additional performance issue is the *fairness* of the estimation across groups. As an example, consider the following simple query:

The output of this query in an online aggregation system can be a set of interfaces, one per output group, as in the example interface in Figure 3. If each group is equally important, the user would like the estimations in each group to tend toward accuracy at approximately the same rate. 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 each group is updated relative to the others. An example of such an interface appears in Figure 4.

A third performance constraint is that output 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 or boredom for the user.

A number of points come clear from this discussion, both in terms of usability and performance:

Usability:

1. *Interfaces:* Statistical, graphical, and/or other intuitive interfaces should be presented to allow users to observe the processing, and get a sense of the current level of accuracy. The set of interfaces must be extensible, so that an appropriate interface can be presented for each aggregation function, or combination of functions. A good Applications Programming Interface (API) must be provided to facilitate this.
2. *Control over Performance:* Users should be able to control the tradeoffs between accuracy, time and fairness in a natural and powerful manner.
3. *Granularity of Control:* Control should be at the granularity of individual results. For example, for a query with multiple outputs (*e.g.* multiple groups), the user should be able to control each output individually.

Performance Goals:

1. *Response time to accuracy:* The main performance goal should be response time to acceptable accuracy as perceived by user demands.
2. *Response time to completion:* A secondary performance goal is response time to completion.
3. *Fairness:* For queries with multiple outputs, fairness must be considered along with the performance of individual results.
4. *Pacing of results:* Updated output should be available at a reasonably regular rate.

Query 3:

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

3. A First-Cut Implementation

We have developed a very simple prototype of our ideas in the Illustra Object-Relational DBMS. Illustra is convenient for prototyping online aggregation because it supports arbitrary user-defined output functions, which we use to produce running aggregates.

Consider Query 3, requesting the average grade of all students in all courses. In Illustra, we can write a C function `running_avg(integer)` which returns a float by computing the current average after each tuple. In addition to this function, we can also write `running_confidence` and `running_interval`, pseudocode for which is given in Figure 5[2]. Note that the `running_*` functions are *not* registered aggregate functions. As a result, Illustra returns `running_*` values for every tuple that satisfies the `WHERE` clause.

Figure 6 shows a number of outputs from the query, along with elapsed times. This is a trace of running the query upon a table of 1,547,606 records representing all course enrollments in the history of all students enrolled at the University of Wisconsin-Madison during the spring of 1994. The grade field varied between 0 and 4. The query was run on an untuned installation of Illustra on a Pentium PC

[2] Note that the `running_confidence` and `running_interval` functions we chose are merely an example. Hoeffding's inequality is only appropriate for estimating the average values in scanning a base table, and does not naturally extend to join queries, which require alternative statistical estimators [Haa96].

```
running_avg(float current)
{
      /*
      ** count and sum are initialized to 0 at start of processing
      ** and are maintained across invocations until the query
      ** is complete.
      */
        static int count, sum;
        sum += current;
        count++;
        return(sum/count);
}

running_interval(float current, Column c)
{
        /* based on Hoeffding's inequality [Hoe63] */
        static int count; /* initialized to 0, maintained across calls */
        static int upper = highest value in column c, available from db stats;
        static int lower = lowest value in column c, available from db stats;

        count++;
        return ((1.36*(upper - lower)) / sqrt(count));
}

running_confidence(float current)
{
        return(95%);
}
```

Figure 5: Psuedo-code for Illustra running aggregate functions.

running Windows NT. The elapsed time shown is scaled by an unspecified factor due to privacy constraints from Illustra Information Technologies, and the times were measured roughly using a stopwatch, since we had no tools to measure running results during query execution. Although this presents a rather rough characterization of the performance, the message is clear: online aggregation produces useful output dramatically more quickly than traditional batch-mode aggregation. The running aggregation functions began to produce reasonable approximations in under one second, and were within 0.1 grade points of the correct answer in under 15 seconds. The final accurate answer was not available for over 15 *minutes*. This dramatically demonstrates the advantages of online aggregation over batch-mode aggregation, and shows that an extensible system can provide some of the functionality required to support online aggregation.

3.1 Problems with the Prototype

Illustra's extensibility features make it very convenient for supporting simple running aggregates such as this. Illustra is less useful for more complicated aggregates. A number of problems arise in even the most forward-looking of today's databases, from the fact that they are all based on the traditional performance goal of minimizing time to a complete answer. Some of the most significant problems include:

1. **Grouping:** Since our running aggregate functions are not in fact Illustra aggregates, they can not be used with an SQL `GROUP BY` clause.

2. **Inappropriate Query Processing Algorithms:** Our example above is a simple table-scan. For aggregates over joins, Illustra (or any other traditional DBMS) does not meet the performance goals above, for a variety of reasons which will be described in the next section. The theme behind all these reasons is that the standard relational query processing algorithms for operations like join, duplicate elimination, and grouping were designed to minimize time to completion, rather than the performance goals of Section 2.

3. **Inappropriate Optimization Goals:** A relational query optimizer tries to minimize response time to a complete answer. Traditional relational optimizers will often choose join orders or methods that generate useful approximations relatively slowly.

4. **Unfair Grouping:** Grouping in a traditional DBMS is usually done via sorting or hybrid hashing, both of which can lead to unfairness at the output. This will be discussed in detail in the next section.

5. **Lack of Run-Time Control:** It is possible to cancel an ongoing query in a traditional DBMS, but it is not possible to control the query in any other significant way. Conceivably, groups could be canceled individually at the output, but this does not change the execution strategy, it merely informs the client application to avoid displaying some tuples. As a result there is no way to control the ongoing performance of the query in any way.

6. **Inflexible API:** In our example above, the system returns a refined estimate once per tuple of the `grades` table. In many cases this overburdens the client-level interface program, which does not need to update the screen so regularly. There is no way for the client program to tell the DBMS to "skip" some tuples, or to run asynchronously until the client is ready for new information. Nor is it possible for the DBMS to pass information to the client program only when the output changes significantly. All of these factors are uncontrollable because we have expressed the aggregation as a standard relational query, and are therefore telling the DBMS to generate and ship all running result tuples to the client application.

4. Query Processing Issues

Supporting the performance and usability goals stated above requires making numerous changes in a database system. In this section we sketch a number of the query processing issues that arise in supporting online aggregation. In general these problems require a reevaluation of various pieces of a relational database system, but do not necessarily point to the need for a new DBMS architecture.

AVG	Confidence	Interval	Elapsed Time (Scaled)
2.983193	95%	0.5	0.75 sec
2.916216	95%	0.4	1.30 sec
2.832827	95%	0.3	1.95 sec
2.691892	95%	0.2	4.05 sec
2.596453	95%	0.1	14.30 sec
2.555951	100%	0.0	15 min 28.10 sec

Figure 6: Output and Elapsed Time for Query 3

4.1 Data Layout and Access Methods

In order to ensure that an online aggregation query will quickly converge to a good approximation, it is important that data appear in a controlled order. Order of access is unspecified in a relational query language, and is dependent upon the data layout and access methods at the lower levels of the database system.

4.1.1 Clustering in Heaps

In a traditional relational system, data in an unstructured "heap" storage scheme can be stored in sorted order, or in some useful *clustering* scheme, in order to facilitate a particular order of access to the data during a sequential scan of the heap. In a system supporting online aggregation, it may be beneficial to keep data ordered so that attributes' values are evenly distributed throughout the column. This guarantees that sequentially scanning the relation will produce a statistically meaningful sample of the columns fairly quickly. For example, assume some column c of relation R contains one million instances of the value "0", and one million instances of the value "10". If R is ordered randomly, the running average will approach "5" almost instantly. However if R is stored in order of ascending c, the running average of c will remain "0" for a very long time, and the user will be given no clue that any change in this average should be expected later. Virtually all statistical methods for estimating confidence intervals assume that data appear in a random order [Haa96].

Clustering of heap relations can also take into account fairness issues for common `GROUP BY` queries. For example, in order to guarantee that a sequential scan of the `grades` relation will update all groups at the same rate for the query output displayed in Figure 2, it would be beneficial to order the tuples in `grades` round-robin by `course_name`, and subject to that ordering place the tuples within each course in a random order.

4.1.2 Indices

A relation can only be clustered one way, and this clustering clearly cannot be optimized for all possible queries. Secondary orderings can be achieved by accessing the relation through secondary indices. Given an unordered relation, a functional index [MS86, LS88] or linked-list can be built over it to maintain an arbitrary useful ordering, such as a random ordering, a round-robin ordering, etc. [3] Such indices will be useful only for particular online aggregation queries, however, and may not be worth the storage and maintenance overheads that they require. Traditional B+-trees [Com79] can also be used to aid online aggregation.

B+-trees can be used to guarantee fairness in scanning a relation. For example, if a B+-tree index exists on the `course_name` attribute, fairness for Query 2 can be guaranteed by opening multiple scans on the index, one per value of the `course_name` column. Tuples can be chosen from the different scans in a round-robin fashion. As users choose to speed up or slow down various groups, the scheme for choosing the next tuple can favor some scan cursors over other. This technique enhances fairness and user control over fairness, while utilizing indices that can also be used for traditional database processing like selections, sorts, or nested-loop joins.

Ranked B+-trees are B+-trees in which each subtree is labeled with the number of leaf nodes contained in that subtree [Knu73]. Many authors in the database sampling community have noted that such trees can be used to rapidly determine the selectivity of a range predicate over such an index. These trees can also be used to improve estimates in running aggregation queries. When traversing a ranked B+-tree for a range query, one knows the size of subranges before those ranges are retrieved. This

[3] One way to produce a random ordering on a table is to have an extra column in the table containing a random number per tuple, and order on that column. Another technique is to take a good hash function f, and build a functional index over *f({keys})*, where *{keys}* is a candidate key of the table. If no candidate key is available then a system-provided tuple or object identifier can be used instead.

information can be used to help compute approximations or answers for aggregates. For example, COUNT aggregates can be quickly estimated with such structures: at each level of the tree, the sum of the number of entries in all subranges containing the query range is an upper bound on the COUNT. A similar technique can be used to estimate the AVG. A generalization of ranked B+-trees is to include additional statistical information in the keys of the B+-tree. For example, the keys could contain a histogram of the frequencies of data values to be found in the subtree of their corresponding pointer. This can increase the accuracy of the estimations as the tree is descended. Of course B+-trees can be used to quickly evaluate MIN and MAX as well.

As a general observation, it is important to recognize that almost all database search tree structures are very similar, presenting a labeled hierarchy of partitions of a column [HNP95]. In essence, the labels (or "keys") in the search tree are aggregate descriptions of the data contained in the leaves below them. An entire level of a database search tree is thus an abstraction of the set of values indexed by the structure. As a result, a horizontal scan of an internal level of a search tree should give a "rough picture" of the data contained at the leaves. This intuition should allow arbitrary search trees (e.g. B+-trees, R-trees [Gut84], hB-trees [LS90], GiSTs [HNP95], etc.) to be used for refining estimations during online aggregation.

4.2 Complex Query Execution

In the previous subsections, we showed how heaps and indices can be structured and used to improve online aggregation processing on base tables. For more complex queries, a number of query execution techniques must be enhanced to support our performance and usability goals.

4.2.1 Joins

Join processing is an obvious arena that requires further investigation for supporting online aggregation. Popular join algorithms such as sort-merge and hybrid-hash join are *blocking* operations. Sort-merge does not produce any tuples until both its input relations are placed into sorted runs. Hybrid-hash does not produce any tuples until one of its input relations is hashed. Blocking operations are unacceptable for online aggregation, because they sacrifice interactive behavior in order to minimize the time to a complete answer. This does not match the needs of an online aggregation interface as sketched in Section 2.

Nested-loops join is not blocking, and hence seems to be the most attractive of the standard join algorithms for our purposes. It can also be adjusted so that the two loops proceed in an order that is likely to produce statistically meaningful estimations; *i.e.* the scans on the two relations are done using indices or heaps ordered randomly and/or by grouping of the output. Nested-loops join can be painfully inefficient, however, if the inner relation is large and unindexed. An alternative non-blocking join algorithm is the pipelined hash join [WA91], which has the disadvantage of requiring a significant amount of main memory to avoid paging. Some new join techniques (or very likely additional adaptations of known techniques) may be helpful in making queries in this situation meet our performance and usability goals.

4.2.2 Grouping

Another important operation for aggregation queries is grouping. In order to compute an aggregate function once per group of an input relation, one of two techniques is typically used: sorting or hashing. The first technique is to sort the relation on the grouping attribute, and then compute the aggregate per group using the resulting ordered stream of tuples. This technique has two drawbacks for online aggregation. First, sorting is a blocking operation. Second, sorting by group destroys fairness, since no results for a group are computed until a complete aggregate is provided for the preceding group.

The second technique used is to build a hash table, with one entry per group to hold the state variables of the aggregate function. The state variables are filled in as tuples stream by in a random order. This technique works much better for online aggregation. However, in some systems hybrid hashing is used if there are too many groups for the hash tables to fit in memory. As noted above, hybrid hashing is a blocking operator which may be unacceptable for online aggregation. Naïve hashing, which allocates as big a hashtable as necessary in virtual memory, may be preferable because it is non-blocking, even though it may result in virtual memory paging. However, note that if there are too many groups to fit in a main-memory hash table, there may also be too many groups to display simultaneously to the user. This is related to the super-aggregation issues which are discussed in Section 5.1.2.

4.3 Query Optimization

Perhaps the most daunting task in developing a database system supporting online aggregation is the modification of a query optimizer to maximize the performance and usability goals described in Section 2. This requires quantifying those goals, and then developing cost equations for the various query processing algorithms, so that the optimizer can choose the plan which best fits the goals. For example, while nested-loops join is the most natural non-blocking join operator, for some queries it may take so much time to complete that a blocking join method may be appropriate. This decision involves a tradeoff between regular pacing and response time to completion. Many other such tradeoffs can occur among the performance goals, and it is difficult to know how the various goals should be weighted, how such weighting depends on user needs, and how users can express those need to the system. A good optimizer for online aggregation must be a concrete description of how online aggregation is best performed. The task of developing such an optimizer is best left until the performance and user interface issues are more crisply defined, and the various execution algorithms are developed.

4.4 Statistical Issues

Statistical confidence measurements provide users of online aggregation with a quantitative sense of the progress of an ongoing query. They are clearly a desirable component of an online aggregation system.

Computation of running confidence intervals for various common aggregates presents a non-trivial challenge, akin to the body of work that has developed for database sampling [HOT88, HOT89, LNSS93, HNSS95, etc.] Estimation of confidence factors can take into account the statistics stored by the database for base relations, but it also needs to intelligently combine those statistics in the face of intervening relational operators such as selection, join, and duplicate elimination. In addition to the work on sampling, recent work on building optimal histograms [IP95] and frequent-value statistics [HS95] is applicable here. An open question is whether new kinds of simple database statistics could be maintained to aid in confidence computations for online aggregation.

One major advantage of online aggregation over sampling is that the stat istical methods for online aggregation form an added benefit in the user interface, but are not an intrinsic part of the system. That is, online aggregation queries can run even when there are no known statistical methods for estimating their ongoing accuracy; in such scenarios users must use qualitative factors (*e.g.* the rate of change of the running aggregate) to decide whether it is safe to stop a query early. This is in stark contrast to the work on database sampling, which requires accurate statistical techniques to begin operation.

This power of an online aggregation system can be misused, however. Unless sufficient warning is presented, statistically insignificant running results can lull a user into a false sense of knowledge about their data. One key issue that remains to be explored is the level of statistical confidence that users will demand during online aggregation. On one extreme are users who demand complete accuracy; these will be satisfied only with complete answers, such as those returned by a traditional batch system. On the other extreme are users who demand no accuracy at all; these will be satisfied by "wild guesses", and do not need a database system. Database sampling techniques provide a middle ground, quantitatively estimating the trustworthiness of their output. Online aggregation systems allow any point in this spectrum to be achieved, and let users choose the level of certainty that they desire on the fly. Although in many scenarios no quantitative estimate of accuracy will be available during online aggregation, humans regularly operate under such uncertain knowledge, and our belief is that an online aggregation system will be useful even in scenarios where no statistical estimation techniques are available.

5. Future Issues

5.1 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.

5.1.1 API and Pacing

Online aggregation functions can produce a new data value as often as once per tuple of input to the function. In standard textual interfaces like those of Figure 2-4, the current running aggregation value should not be updated once per tuple, or the user will be unable to read the values as they rapidly change on the screen, and the client application will waste time displaying values that have not changed significantly. Analogous pacing issues will exist for non-textual user interfaces, but the rate of pacing will depend on the interface. For example, in an interface that plots points onto a map, updates can probably proceed as quickly as possible without overwhelming the user's ability to perceive transitory states. Thus pacing seems to be an interface-specific issue, and needs to be controlled differently for different interfaces.

An API must be set up so that the client application can negotiate with the DBMS about the pacing rate. It may be appropriate for a client to periodically poll the DBMS for updated estimates. Alternatively, it may be better for the DBMS to provide data to the client asynchronously, when estimation values change significantly or reach important confidence threshholds. The settings for such a "wakeup call" from the DBMS should be controllable by the client. It will often be appropriate to allow users to adjust pacing during execution, so client input on pacing must be passed to the DBMS and handled on the fly.

An important consideration in developing an API for open aggregation is that it be backwards-compatible with existing APIs for query processing (e.g. SQL cursors), in order to make a system with online aggregation subsume existing approaches. This will present some complications, since online aggregation is not really cursor-based: in online aggregation, all output results are presented as soon as possible, and then re-presented multiple times as they are refined. Our Illustra prototype avoided this problem by folding aggregation into traditional cursor-based processing, but this simplification came at the cost of other problems, including lack of grouping and lack of pacing control.

5.1.2 Manageable Information Loads

An orthogonal problem to pacing a given output is the problem of presenting multiple outputs in a manageable fashion. Queries with too many outputs (*e.g.* thousands of groups) can be hard to visualize effectively, and even harder to control in a meaningful way.

In current OLAP systems, this problem is typically handled by initially grouping the input data into a small number of large aggregate groups. Users can then "drill down" into individual groups to see more detail on sub-groups. As the display gets cluttered, the low-level groups can be re-aggregated or "rolled up" into their higher-level groups. "Drilling down" into a group requires the aggregates to be computed for each of the subgroups. "Rolling up" a number of groups requires the subaggregates to be combined into a superaggregate. In their paper on the CUBE operator, Gray, *et al.* propose that these operations be supported over an extended SQL query interface [GPLB96]. What this means is that the database engine may need to compute a complex aggregation query every time a user "double-clicks" on a group to effect a drill-down operation. The results of such screen gestures are supposed to be relatively quick, but it is clear that a batch aggregation system may compute for a very long time in many such cases. An online aggregation interface would clearly be beneficial for these operations. A fair bit of work will be required to develop efficient online algorithms that take advantage of already-displayed results, both in order to avoid recomputing results, and in order to use previous results to generate accurate estimation of future results.

5.2 Unifying Other Online Display Techniques

Users are familiar with online display techniques from web-browsing software like Netscape Navigator, which display images that have been compressed using techniques such as interlacing and wavelets. These images can be very large, and are displayed as they are shipped over the network. As more and more of the image data arrives at the application, the user perceives the image getting sharper and sharper. The display of a compressed image is essentially the output of an online aggregation over the bits of the image. It may be interesting to try to unify the notions of online aggregation with interlacing, wavelets and other compression techniques that present incrementally improving results.

5.3 Nested queries

SQL supports nesting of queries in various ways. Queries can be nested in the `FROM` clause in the form of views. Queries can be nested in the `WHERE` clause in the form of subqueries. Aggregation queries with `GROUP BY` clauses can be nested in either location. For purposes of illustration, we consider an example from the first category, which generates per department the percentage of total sales contributed by that department:

```
Query 4:
SELECT SUM(s.sale_price)/v.total_sales
  FROM sales s, (SELECT SUM (sales.sale_price) AS total_sales
                   FROM sales) v,
GROUP BY s.dept;
```

An open question in this work is how one can provide an online execution of queries that contain aggregation both at the topmost level and at lower levels: the running results at the top level will depend on the running results at lower levels. Traditional block-at-a-time processing requires the lower blocks to be processed before the higher blocks, but this is a blocking processing model (no pun intended), and hence violates the performance goals mentioned in Section 2. Any non-blocking approach would clearly present significant statistical problems in terms of confidence intervals and other estimations, in addition to complicating other issues of performance and usability mentioned in Section 2.

5.4 Checkpointing and Reuse

Aggregation queries that benefit from online techniques will typically be long-running operations. It is well known that long-running operations should be *checkpointed,* so that computation can be saved across system crashes, power failures, and operator errors. This is particularly true for online aggregation queries: users should be allowed to “continue” queries (or pieces of queries) that they have previously canceled. Techniques for checkpointing the state of online aggregation queries should therefore prove very useful. Checkpoints of partially computed queries can also be used as materialized sample views [Olk93] or partial indexes [Sto89], depending on their makeup.

5.5 Concurrency Control

Online aggregation queries are likely to be very time-consuming. Such queries will typically hold read-locks for long periods of time, and may prevent updates from occurring. A typical solution to this problem is to avoid setting long-term read locks, and guarantee a lower level of consistency for long-running aggregation queries. This approach is known as *cursor stability* [Gra79], and has the disadvantage that while the values read are individually correct, they are not from one transaction-consistent state of the database. An alternative *compensation-based* technique was proposed by Srinivasan and Carey [SC92]. Their solution guarantees transaction semantics and high concurrency by posting update actions to the query processing engine, which generates compensatory actions for affected queries. Since online aggregation queries generate output before processing all their input, compensation-based techniques in an online aggregation system would guarantee transaction semantics for the *final* result of the aggregate, but not necessarily for the running results. This is because the running output may be affected by concurrent updates, and only "corrected" later on by the compensatory actions. An interesting spectrum of semantic options is presented here – in order of ascending strictness we have cursor stability, followed by compensation-based transactions, followed by strict two-phase locking. Compensation-based processing seems most appropriate for online aggregation, since it provides both high performance and eventually correct results. However in some scenarios strict transaction semantics will be desirable, especially to guarantee the accuracy of statistical estimates.

6. Conclusion

In this paper we demonstrate the need for a new approach to aggregation which is interactive, intuitive, and user-controllable. Based on a preliminary implementation in Illustra we find that significant new technology will be required to provide the functionality and performance needed for online aggregation. We define the parameters of success for such a system, and discuss a number of database system implementation issues that need to be addressed to make online aggregation work well.

It seems clear that a good online aggregation system will require significant new research. However it seems equally clear to us that relational systems can and should be extended to support both online aggregation as well as traditional relational processing. We are in the process of designing such a system at UC-Berkeley, which will address a number of interesting challenges in query processing and optimization, as well as in data layout, indexing, concurrency control and visualization.

7. Acknowledgments

The author benefited greatly from discussions about "sloppy databases" with Jeff Naughton and Praveen Seshadri. Naughton and Seshadri also provided useful feedback on a draft of this paper, as did Peter Haas, Marcel Kornacker, Sunita Sarawagi, Mike Carey, Alex Aiken, Adam Sah, Navin Kabra, and the students of CS286, Spring 1996. The Wisconsin course dataset was graciously provided by Bob Nolan of UW-Madison's Department of Information Technology (DoIT).

8. Bibliography

[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.`

[Com79] Douglas Comer. The Ubiquitous B-Tree. *Computing Surveys*, 11(2):121-137, June 1979.

[CS96] Surajit Chaudhuri and Kyuseok Shim. Optimizing Queries with Aggregate Views. In *Proc. EDBT96*, to appear.

[GBLP96] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Microsoft Research Technical Report MSR-TR-95-22, 1996.

[GHQ95] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-Query Processing in Data Warehousing Environments. In *Proc. 21st International Conference on Very Large Data Bases,* Zurich, September 1995, pages 358-369.

[Gra79] Jim Gray, Notes on Database Operating Systems. In Rudy Bayer, Robert Graham, and G. Seegmuller, editors. *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science, Volume 60. Springer-Verlag, 1979. Also available as IBM Research Report RJ2188, IBM Almaden Research Center, February 1978.

[Gut84] Antonin Guttman. R-Trees: A Dynamic Index Structure For Spatial Searching. In *Proc. ACM-SIGMOD International Conference on Management of Data,* pages 47-57, Boston, June 1984.

[Haa96] Peter Haas. Personal communication, February 1996,

[Hoe63] Wassily Hoeffding. Probability Inequalities for Sums of Bounded Random Variables. *Journal of the American Statistics Society,* 58:13-30, 1963.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In *Proc. 21st International Conference on Very Large Data Bases,* Zurich, September 1995, pages 562-573.

[HNSS95] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri and Lynne Stokes. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *Proc. 21st International Conference on Very Large Data Bases,* Zurich, September 1995, pages 311-322.

[HOT88] Wen-Chi Hou, Gultekin Ozsoyoglu, and Baldeao K. Taneja. Statistical Estimators for Relational Algebra Expressions. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Austin, March 1988, pages 276-287.

[HOT89] Wen-Chi Hou, Gultekin Ozsoyoglu, and Baldeo K. Taneja. Processing Aggregate Relational Queries with Hard Time Constraints. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Portland, May-June 1989, pages 68-77.

[HS95] Peter J. Haas and Arun N. Swami. Sampling-Based Selectivity Estimation for Joins Using Augmented Frequent Value Statistics. In *Proc. 11th International Conference on Data Engineering*, Taipei, Taiwan, March, 1995, pages 522-531.

[IP95] Yannis Ioannidis and Viswanath Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 233-244, San Jose, May 1995.

[Knu81] Donald Ervin Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Co., 2nd edition, 1981.

[LNSS93]Richard J. Lipton, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Efficient Sampling Strategies for Relational Database Operations. *Theoretical Computer Science*, (116):195-226, 1993.

[LS88] C. Lynch and M. Stonebraker. Extended User-Defined Indexing with Application to Textual Databases. In *Proc. 14th International Conference on Very Large Data Bases*, pages 306-317, Los Angeles, August-September 1988.

[LS90] David B. Lomet and Betty Salzberg. The hB-Tree: A Multiattribute Indexing Method. *ACM Transactions on Database Systems*, 15(4), December 1990.

[MS86] D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. In Klaus R. Dittrich and Umeshwar Dayal, editors, *Proc. Workshop on Object-Oriented Database Systems*, pages 171-182, Asilomar, September 1986.

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

[OR86] Frank Olken and Doron Rotem. Simple Random Sampling from Relational Databases. In *Proc. 12th International Conference on Very Large Data Bases,* pages 160-169.

[OR90] Frank Olken and Doron Rotem. Random Sampling from Database Files: A Survey. In Z. Michalewicz, editor, *Statistical and Scientific Database Management, Proceedings of the Fifth Conference,* pages 92-111. Springer-Verlag, April 1990.

[SC92] V. Srinivasan and Michael J. Carey. Compensation-Based On-Line Query Processing. In *Proc. ACM-SIGMOD International Conference on Management of Data*, San Diego, June 1992, pages 331-340.

[SPL99] Praveen Seshadri, Hamid Pirahesh, and T.Y. Cliff Leung. Complex Query Decorrelation. In *Proceedings 12th IEEE Conference on Data Engineering*, New Orleans, February, 1996.

[Sto89] Michael Stonebraker. The Case for Partial Indexes. *SIGMOD Record*, 18(4):4-11, 1989.

[VM92] M. Vetterli and U. K. Metin. Multiresolution Coding Techniques for Digital Television: A Review. In *Multidimensional Systems and Signal Processing*, 3:161-187.

[WA91] A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proc. First International Conference on Parallel and Distributed Information Systems,* Miami Beach, December, 1991, pages 68-77.

[WE80] C. K. Wong and M. C. Easton. An Efficient Method for Weighted Sampling Without Replacement. *SIAM Journal on Computing*, 9(1):111-113, February 1980.

[YL95] Weipeng P. Yan and Per-Åke Larson. *Eager Aggregation and Lazy Aggregation*. In *Proc. 21st International Conference on Very Large Data Bases,* Zurich, September 1995, pages 345-357.