Data Mining and Knowledge Discovery, 12, 281–314, 2000
© 2000 Kluwer Academic Publishers. Manufactured in The Netherlands.

Informix under CONTROL: Online Query Processing

JOSEPH M. HELLERSTEIN jmh@cs.berkeley.edu
RON AVNUR ronathan@cs.berkeley.edu
VIJAYSHANKAR RAMAN rshankar@cs.berkeley.edu
Computer Science Division, U.C. Berkeley, USA

Editors: Fayyad, Mannila, Ramakrishnan

Abstract. The goal of the CONTROL project at Berkeley is to develop systems for interactive analysis of large data sets. We focus on systems that provide users with iteratively refining answers to requests and online control of processing, thereby tightening the loop in the data analysis process. This paper presents the database-centric subproject of CONTROL: a complete *online* query processing facility, implemented in a commercial Object-Relational DBMS from Informix. We describe the algorithms at the core of the system, and detail the end-to-end issues required to bring the algorithms together and deliver a complete system.

Keywords: online query processing, interactive, informix, control, data analysis, ripple joins, online reordering

1. Introduction

Of all men's miseries, the bitterest is this: to know so much and have control over nothing.
– Herodotus

Data analysis is a complex task. Many tools can been brought to bear on the problem, from user-driven SQL and OLAP systems, to machine-automated data mining algorithms, with hybrid approaches in between. All the solutions on this spectrum share a basic property: analyzing large amounts of data is a time-consuming task. Decision-support SQL queries often run for hours or days before producing output; so do data mining algorithms (Agrawal, 1997). It has recently been observed that user appetite for online data storage is growing faster than what Moore's Law predicts for the growth in hardware performance (Papadopoulos, 1997; Winter and Auerbach, 1998), suggesting that the inherent sluggishness of data analysis will only worsen over time.

In addition to slow performance, non-trivial data analysis techniques share a second common property: they require thoughtful deployment by skilled users. It is well-known that composing SQL queries requires sophistication, and it is not unusual today to see an SQL query spanning dozens of pages (Walter, 1998). Even for users of graphical front-end tools, generating the correct query for a task is very difficult. Perhaps less well-appreciated is the end-user challenge of deploying the many data mining algorithms that have been developed. While data mining algorithms are typically free of complex input languages, using them effectively depends on a judicious choice of algorithm, and on the careful tuning of various algorithm-specific parameters (Fayyad, 1996).

A third common property of data analysis is that it is a multi-step process. Users are unlikely to be able to issue a single, perfectly chosen query that extracts the "desired information" from a database; indeed the idea behind data analysis is to extract heretofore unknown information. User studies have found that information seekers very naturally work in an iterative fashion, starting by asking broad questions, and continually refining them based on feedback and domain knowledge (O'day and Jeffries, 1993). This iteration of analyses is a natural human mode of interaction, and not clearly an artifact of current software, interfaces, or languages.

Taken together, these three properties result in a near-pessimal human-computer interaction: data analysis today is a complex process involving multiple time-consuming steps. A poor choice or erroneous query at a given step is not caught until the end of the step when results are available. The long delay and absolute lack of control between successive queries disrupts the concentration of the user and hampers the process of data analysis. Therefore many users eschew sophisticated techniques in favor of cookie-cutter reports, significantly limiting the impact of new data analysis technologies. In short, the mode of human-computer interaction during data analysis is fundamentally flawed.

1.1. CONTROL: Interactive data analysis

The CONTROL[1] project attempts to improve the interaction between users and computers during data analysis. Traditional tools present *black box* interfaces: users provide inputs, the system processes silently for a significant period, and returns outputs. Because of the long processing times, this interaction is reminiscent of the *batch* processing of the 1960's and '70's. By contrast, CONTROL systems have an *online* mode of interaction: users can control the system at all times, and the system continuously provides useful output in the form of approximate or partial results. Rather than a black box, online systems are intended to operate like a *crystal ball*: the user "sees into" the online processing, is given a glimpse of the final results, and can use that information to change the results by changing the processing. This significantly tightens the loop for asking multiple questions: users can quickly sense if their question is a useful one, and can either refine or halt processing if the question was not well-formed. We describe a variety of interactive online systems in Section 2.

Though the CONTROL project's charter was to solve interface problems, we quickly realized that the solutions would involve fundamental shifts in system performance goals (Hellerstein, 1997). Traditional algorithms are optimized to complete as quickly as possible. By contrast, online data analysis techniques may never complete; users halt them when answers are "good enough". So instead of optimizing for completion time, CONTROL systems must balance two typically conflicting performance goals: minimizing uneventful "dead time" between updates for the user, while simultaneously maximizing the rate at which partial or approximate answers approach a correct answer. Optimizing only one of these goals is relatively easy: traditional systems optimize the second goal (they quickly achieve a correct answer) by pessimizing the first goal (they provide no interactivity). Achieving both goals simultaneously requires redesigning major portions of a data analysis system, employing a judicious mix of techniques from data delivery, query processing, statistical estimation and user interfaces. As we will see, these techniques can interact in non-trivial ways.

1.2. Online query processing in informix

In this paper we focus on systems issues for implementing online query processing—i.e., CONTROL for SQL queries. Online query processing enables a user to issue an SQL query, see results immediately, and adjust the processing as the query runs. In the case of an online *aggregation* query, the user sees refining estimates of the final aggregation results. In the case of an online *enumeration* query (i.e., a query with no aggregation), the user receives an ever-growing collection of result records, which are available for browsing via user interface tools. In both cases, users should be able to provide feedback to the system while the query is running, to control the flow of estimates or records. A common form of control is to terminate delivery of certain classes of estimates or records; more generally, users might express a preference for certain classes of estimates or records over others.

Online query processing algorithms are designed to produce a steady stream of output records, which typically serve as input to statistical estimators and/or intelligent user interfaces. Our discussion here focuses mostly on the pipelined production of records and the control of that data flow; statistical and interface issues are considered in this paper only to the extent that they drive the performance goals, or interact with implementation issues. The interested reader is referred to Section 1.4 for citations on the statistical and interface aspects of online query processing.

As a concrete point of reference, we describe our experience implementing online query processing in a commercial object-relational database management system (DBMS): Informix's Dynamic Server with Universal Data Option (UDO) (Informix, 1998). The pedigree of UDO is interesting: formerly known as Informix Universal Server, it represents the integration of Informix's original high-performance relational database engine with the object-relational facilities of Illustra (Illustra, 1994), which in turn was the commercialization of the Postgres research system (Stonebraker and Kemnitz, 1991). Informix Corporation made its source code and development environment available to us for this research, enabling us to test our ideas within a complete SQL database engine.

Working with UDO represented a significant challenge and opportunity. UDO is a large and complex system developed (in its various ancestral projects) over more than 15 years. As we describe in Hellerstein et al. (1997), online query processing cannot simply be implemented as a "plug-in" module for an existing system. Most of the work described in this paper involved adding significant new features to the UDO database engine itself. In a few cases—particularly in crafting an API for standard client applications—we were able to leverage the object-relational extensibility of UDO to our advantage, as we describe below.

In addition to adding new algorithms to the system, a significant amount of effort went into architecting a complete end-to-end implementation. Our implementation allows the various algorithms to be pipelined into complex query plans, and interacts effectively with a wide variety of client tools. This paper describes both the core algorithms implemented in UDO, as well as the architectural issues required to provide a usable system.

1.3. Structure of the paper

We discuss related work in Section 1.4. In Section 2 we describe a number of application scenarios for CONTROL-based systems. Section 3 describes the core algorithms used in online query processing including access methods, data delivery algorithms, and join algorithms. Section 4 describes the end-to-end challenges of putting these algorithms together in the context of a commercial object-relational database management system. Section 5 demonstrates the performance of the system, in terms both of interactivity and rate of convergence to accurate answers. In Section 6 we conclude with a discussion of future work.

1.4. Related work

The CONTROL project began by studying online aggregation, which was motivated in (Hellerstein (1997a) and Hellerstein et al. (1997). The idea of online processing has been expanded upon within the project (Hellerstein, 1997b; Hellerstein, 1998a; Hellerstein et al., 1999; Hidber, 1997) a synopsis of these thrusts is given in Section 2. Recently we have presented the details of two of our core query processing algorithms: ripple joins (Haas and Hellerstein, 1999) and online reordering (Raman et al., 1999). Estimation and confidence interval techniques for online aggregation are presented in Haas (1996, 1997) and Haas and Hellerstein (1999). To our knowledge, the earliest work on approximate answers to decision-support queries appears in Morgenstein's dissertation from Berkeley (Morgenstein, 1980), in which he presents motivation quite similar to ours, along with proposed techniques for sampling from relations and from join results.

Our work on online aggregation builds upon earlier work on estimation and confidence intervals in the database context (Hou et al., 1988; Haas et al., 1996; Lipton et al., 1993). The prior work 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. (1989) 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"). The drawback with using sampling to produce approximate answers is that the end-user needs to understand the statistics. Moreover, making the user specify statistical stopping conditions at the beginning reduces the execution time but does not make the execution interactive; for instance there is no way to dynamically control the rate of processing—or the desired accuracy—for individual groups of records.

More recent work has focused on maintaining precomputed summary statistics for approximately answering queries (Gibbons and Matias, 1998; Gibbons et al., 1998); Olken also proposed the construction of sample views (Olken, 1993). In a similar though simpler vein, Informix has included simple precomputed samples for approximate results to ROLAP queries (Informix, 1998). These techniques are to online query processing what materialized views are to *ad hoc* queries: they enhance performance by precomputing results, but are inapplicable when users ask queries that cannot exploit the precomputed results. In the context of approximate query answers, *ad hoc* specification applies both to queries and to the stopping criteria for sampling: a user may specify any query, and want to see answers with differing accuracies. Unlike general materialized views, most precomputed summaries are on single tables, so many of the advantages of precomputed samples can be achieved in an online query processing system via simple buffer management techniques. In short, work on precomputed summaries is complementary to the techniques of this paper; it seems viable to automate the choice and construction of precomputed summaries as an aid to online query processing, much as Hybrid OLAP chooses queries to precompute to aid OLAP processing (Shukla et al., 1998; Harinarayan et al., 1996; Pilot Software, 1998; SQL, 1998).

A related but quite different notion of precomputation for online query processing involves semantically modeling data at multiple resolutions (Silberschatz et al., 1992). A version of this idea was implemented in a system called APPROXIMATE (Vrbsky and Liu, 1993). 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 query processing: it is dependent on carefully designed metadata and does not address aggregation or statistical assessments of precision.

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 (Antoshenkov and Ziauddin, 1996). Bayardo and Miranker propose optimization and execution techniques for fast-first processing using nested-loops joins (Bayardo and Miranker, 1996). Carey and Kossman (1997, 1998), Chaudhuri and Gravano (1996, 1999), and Donjerkovic and Ramakrishnan (1999) discuss techniques for processing ranking and "top-N" queries, which have a "fast-first" flavor as well. Much of this work seems applicable to online query optimization, though integration with online query processing algorithms has yet to be considered. Fagin (1998) proposes an interesting algorithm for the execution of ranking queries over multiple sources that optimizes for early results. This algorithm has a similar flavor to the Ripple Join algorithm we discuss in Section 3.3.

2. Application scenarios and performance requirements

The majority of data analysis solutions are architected to provide black-box, batch behavior for large data sets: this includes software for the back-office (SQL decision-support systems), the desktop (spreadsheets and OLAP tools), and statistical analysis techniques (statistics packages and data mining). The result is that either the application is frustratingly slow (discouraging its use), or the user interface prevents the application from entering batch states (constraining its use.) The applications in this section are being handled by current tools with one or both of these approaches. In this section we describe online processing scenarios, including online aggregation and enumeration, and online visualization. We also briefly mention some ideas in online data mining.

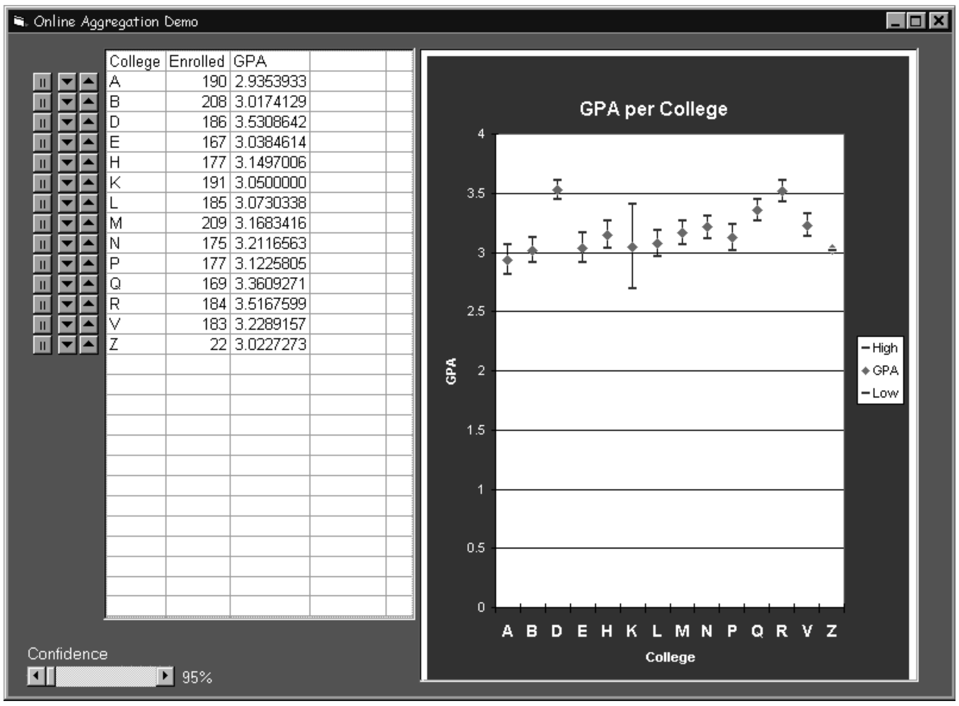
2.1. Online aggregation

Aggregation queries in relational database systems often require scanning and analyzing a significant portion of a database. In current relational systems such query execution has batch behavior, requiring a long wait for the user. Online query processing can make aggregation an interactive process.

Consider the following simple relational query:

```
  SELECT college, AVG(grade)
    FROM enroll
GROUP BY college;
```

This query requests that all records in the `enroll` table be partitioned into groups by college, and then for each college its name and average grade should be returned. The output of this query in an online aggregation system can be a set of interfaces, one per output group, as in figure 1. For each output group, the user is given a current estimate of the final answer. In addition, a graph is drawn showing these estimates along with a description of their accuracy: each estimate is drawn with bars that depict a *confidence interval*, which says that



College	Enrolled	GPA
A	190	2.9353933
B	208	3.0174129
D	186	3.5308642
E	167	3.0384614
H	177	3.1497006
K	191	3.0500000
L	185	3.0730338
M	209	3.1683416
N	175	3.2116563
P	177	3.1225805
Q	169	3.3609271
R	184	3.5167599
V	183	3.2289157
Z	22	3.0227273

Confidence 95%

Figure 1. An online aggregation interface.

with X% probability, the current estimate is within an interval of $\pm\epsilon$ from the final answer (X is set to 95% in the figure). The "Confidence" slider on the lower left allows the user to control the percentage probability, which in turn affects the $2 \cdot \epsilon$ width of the bars. In addition, controls on the upper left of the screen are provided to stop processing on a group, or to speed up or slow down one group relative to others. These controls allow the user to devote more processing to groups of particular interest. These interfaces require the support of significant modifications to a DBMS, which we describe in this paper. We have developed estimators and corresponding implementation techniques for the standard SQL aggregates `AVG`, `COUNT`, and `STDDEV`[2] (Hellerstein et al., 1997; Hass and Hellerstein, 1999).

Online aggregation is particularly useful in "drill-down" scenarios: a user may ask for aggregates over a coarse-grained grouping of records, as in the query above. Based on a quick online estimate of the coarse-grained results, the user may choose to issue another query to "drill down" into a set of particularly anomalous groups. Alternatively the user may quickly find that their first query shows no interesting groups, and they may issue an alternate query, perhaps grouping on different attributes, or requesting a different aggregate computation. The interactivity of online aggregation enables users to explore their data in a relatively painless fashion, encouraging data browsing.

The obvious alternative to online aggregation is to precompute aggregation results before people use the system—this is the solution of choice in the *multidimensional* OLAP (MOLAP) tools (e.g., Hypersion Essbase OLAP Server, 1999). Note that the name OLAP ("OnLine Analytic Processing") is something of a misnomer for these systems. The analytic processing in many OLAP tools is in fact done "off line" in batch mode; the user merely navigates the stored results on line. This solution, while viable in some contexts, is an example of the constrained usage mentioned at the beginning of this section: the only interactive queries are those that have been precomputed. This constraint is often disguised with a graphical interface that allows only precomputed queries to be generated. A concomitant and more severe constraint is that these OLAP systems have trouble scaling beyond a few dozen gigabytes because of both the storage costs of precomputed answers, and the time required to periodically "refresh" those answers. Hybrids of precomputation and online aggregation are clearly possible, in the same way that newer systems provide hybrids of precomputation and batch query processing (e.g., Shukla et al., 1998; Harinarayan et al., 1996; Pilot Software, 1998; Maier and Stein, 1986).

2.2. Online enumeration: Scalable spreadsheets

Database systems are often criticized as being hard to use. Many data analysts are experts in a domain other than computing, and hence prefer simple *direct-manipulation* interfaces like those of spreadsheets (Shneiderman, 1982), in which the data is at least partially visible at all times. Domain-specific data patterns are often more easily seen by "eyeballing" a spreadsheet than by attempting to formulate a query. For example, consider analyzing a table of student grades. By sorting the output by GPA and scrolling to the top, middle, and bottom, an analyst may notice a difference in the ethnic mix of names in different GPA quantiles; this may be evidence of discrimination. By contrast, imagine trying to write an

SQL aggregation query asking for the average grade per apparent ethnicity of the name column—there is no way to specify the ethnicity of a name declaratively. The difficulty is that the (rough) name-to-ethnicity mapping is domain knowledge in the analyst's head, and not captured in the database.

Unfortunately, spreadsheets do not scale gracefully to large datasets. An inherent problem is that many spreadsheet behaviors are painfully slow on large datasets—if the spreadsheet allows large data sets at all. Microsoft Excel, for example, restricts table size to 64K rows or fewer, presumably to ensure interactive behavior. The difficulty of guaranteeing acceptable spreadsheet performance on large datasets arises from the "speed of thought" response time expected of spreadsheet operations such as scrolling, sorting on different columns, pivoting, or jumping to particular cells in the table (by address or cell-content prefix). Thus traditional spreadsheets are not useful for analyzing large amounts of data.

We are building *A-B-C*, a scalable spreadsheet that allows online interaction with individual records (Raman et al., 1999a). As records are enumerated from a large file or a database query returning many rows, A-B-C allows the user to view example rows and perform typical spreadsheet operations (scroll, sort, jump) at any time. A-B-C provides interactive (sub-second) responses to all these operations via the access methods and reordering techniques of Sections 3.1 and 3.2.

Hypotheses formed via online enumeration can be made concrete in A-B-C by grouping records "by example": the user can highlight example rows, and use them to interactively develop a regular expression or other group identity function. Groups are then "rolled up" in a separate panel of the spreadsheet, and users can interactively specify aggregation functions to compute on the groups online. In this case, the online enumeration features of A-B-C are a first step in driving subsequent online aggregation.

2.3. Aggregation + enumeration: Online data visualization

Data visualization is an increasingly active research area, with rather mature prototypes in the research community (e.g. Tioga Datasplash (Aiken et al., 1996), DEVise (Livny et al., 1997), Pad (Perlin and Fox, 1993)), and products emerging from vendors (Ohno, 1998). These systems are interactive data exploration tools, allowing users to "pan" and "zoom" over visual "canvases" representing a data set, and derive and view new visualizations quickly.

An inherent challenge in architecting a data visualization system is that it must present large volumes of information efficiently. This involves scanning, aggregating and rendering large datasets at point-and-click speeds. Typically these visualization systems do not draw a new screen until its image has been fully computed. Once again, this means batch-style performance for large datasets. This is particularly egregious for visualization systems that are expressly intended to support browsing of large datasets.

Related work in the CONTROL project involves the development of online visualization techniques we call CLOUDS (Hellerstein et al., 1999), which can be thought of as *visual* aggregations and enumerations for an online query processing system. CLOUDS performs both enumeration and aggregation simultaneously: it renders records as they are fetched, and also uses those records to generate an overlay of shaded rectangular regions of color

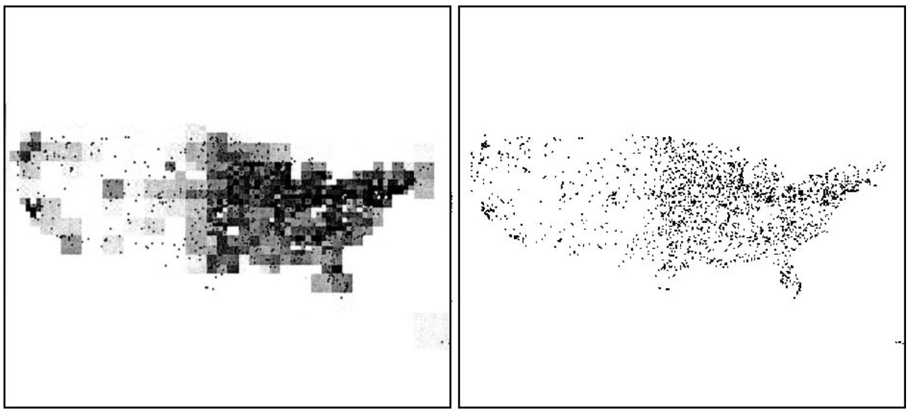


Figure 2. Snapshots of an online visualization of cities in the United States, with and without CLOUDS.

("clouds"), corresponding to nodes in a carefully constructed quad tree. The combination of the clouds and the rendered sample is intended to approximate the final image. This means that the clouds are not themselves an approximation of the image, but rather a *compensatory* shading that accounts for the difference between the rendered records and the projected final outcome. During processing, the user sees the picture improve much the way that images become refined during network transmission. This can be particularly useful when a user pans or zooms over the results of an *ad hoc* query: in such scenarios the accuracy of what is seen is not as important as the rough sense of the moving picture. Figure 2 shows a snapshot of an online visualization of cities in the United States, with and without CLOUDS. Note how the CLOUDS version contains shading that approximates the final density of areas better than the non-CLOUDS version; note also how the CLOUDS visualization renders both data points and shading.

As with the Scalable Spreadsheet, our data visualization techniques tie into data delivery, and benefit directly from the access methods and reordering techniques described in Sections 3.1 and 3.2. For DBMS-centric visualization tools like DEVise and Tioga, the full power of an online query processing system—including joins, aggregations, and so on—is needed in the back end.

2.4. Online data mining

Many data mining algorithms make at least one complete pass over a database before producing answers. In addition, most mining algorithms have a number of parameters to tune, which are not adjustable while the algorithm is running. While we do not focus on data mining algorithms in this paper, we briefly consider them here to highlight analogies to online query processing.

As a well-known example, consider the oft-cited *apriori* algorithm for finding "association rules" in market-basket data (Agrawal and Srikant, 1994). To use an association rule application, a user specifies values for two variables: one that sets a minimum threshold on the amount of evidence required for a set of items to be produced (*minsupport*) and another which sets a minimum threshold on the correlation between the items in the set (*minconfidence*). These algorithms can run for hours without output, before producing association rules that passed the minimum support and confidence thresholds. Users who set those thresholds incorrectly typically have to start over. Setting thresholds too high means that few rules are returned. Setting them too low means that the system (a) runs even more slowly, and (b) returns an overwhelming amount of information, most of which is useless. Domain experts may also want to explicitly prune irrelevant correlations during processing.

The traditional algorithm for association rules is a sequence of aggregation queries, and can be implemented in an online fashion using techniques for online query processing described in this paper. An alternative association rule algorithm called CARMA was developed in the CONTROL project (Hidber, 1997). While not clearly applicable to SQL query processing, CARMA is worthy of mention here for two reasons. First, it very efficiently provides online interaction and early answers. Second—and somewhat surprisingly—CARMA often produces a final, accurate answer faster than the traditional "batch" algorithms, both because it makes fewer passes of the dataset and because it manages less memory-resident state. So in at least one scenario, inventing an algorithm for online processing resulted in a solution that is also better for batch processing!

Most other data mining algorithms (clustering, classification, pattern-matching) are similarly time-consuming. CONTROL techniques seem worth considering for these algorithms, and the development of such techniques seems to be a tractable research challenge. Note that CONTROL techniques tighten loops in the knowledge-discovery process (Fayyad et al., 1996), bringing mining algorithms closer in spirit to data visualization and browsing. Such synergies between user-driven and automated techniques for data analysis seem like a promising direction for cross-pollenation between research areas.

3. Algorithms for online query processing

Relational systems implement a relatively small set of highly tuned query processing operators. Online query processing is driven by online analogs of the standard relational query processing operators, along with a few new operators. In this section we discuss our implementation in Informix of online query processing operators, including randomized data access, preferential data delivery, relational joins, and grouping of result records. Of these, randomized data access was the simplest to address, and our solution required no additions to the Informix SQL engine. It does, however, have an impact on the *physical design* of a database, i.e., the layout of tables and indexes on disk.

3.1. Randomized data access and physical database design

In most scenarios, it is helpful if the output of a partially-completed online query can be treated as a random sample. In online aggregation queries, the estimators for aggregates like AVG and SUM require random sampling in order to allow for confidence intervals or other statements about accuracy. This requirement is less stringent for online enumeration, but still beneficial: in most scenarios a user would prefer to see a representative sample of the data at any given time. Hence we begin our discussion of query operators by considering techniques for randomized data access.

In order to guarantee random data delivery, we need *access methods*—algorithms for data access—that produce ever-larger random samples of tables. This can be accomplished in a number of ways. Random sampling is the same as simply scanning a table in certain scenarios: particularly, when the rows have been randomly permuted prior to query processing or when, as verified by statistical testing, the storage order of the rows on disk is independent of the values of the attributes involved in the aggregation query. Alternatively, it may be desirable to actually sample a table during query processing, or to materialize a small random sample of each base relation during an initialization step, and then subsequently scan the sample base relations during online processing. Olken (1993) surveys techniques for sampling from databases and for maintaining materialized sample views.

We chose to guarantee random delivery in Informix by storing tables in random order. This approach is already available in any DBMS that supports user-defined functions: we *cluster*[3] tables randomly by clustering records on a user-defined function $f(\ )$ that generates pseudo-random numbers.[4] Scans of the table produce ever-larger random samples at full disk bandwidth. To make this scheme work in the face of updates, new tuples should be inserted to random positions in the table, with the tuples formerly in those positions being appended to the end of the table. While not difficult, we have not added random insertion functionality to our Informix prototype.

There are two potential drawbacks to this approach. The first is that every scan of the table generates the same random sample; over time the random but static properties of the order could be misinterpreted as representative of all possible orderings. This can be alleviated somewhat by starting scans at an arbitrary points in the file (as is done in the "shared scans" of some DBMSs (Red Brick Systems, Inc., 1998)), though of course the properties of the fixed order are still reflected. A better solution is to periodically force some random shuffling in the table; this is analogous to the "reorganization" steps common in database administration, and it easily could be made automatic and incremental.

The second problem with this approach is that a relation stored in random order is by definition not stored in some other order. This has ramifications for database design, since it is typical for a database administrator to cluster a table on an attribute frequently referenced in range queries, rather than on a random function. This can be solved in a manner analogous to that of traditional database design: if one has a clustering on some column, and a secondary random clustering is desired, one can generate a secondary random ordering via an index on a random-valued attribute. This can be done without modification in any object-relational system like Informix that supports *functional* indexes (Maier and Stein, 1986; Lynch and Stonebrakere, 1988). One simply constructs a functional index on $f(R \cdot x)$, where $f(\ )$ is a random-number generator, and x is any column of R. The resulting index on R serves as a secondary random ordering structure. Note that scanning a secondary index requires a random I/O per record; this is a performance drawback of secondary random indexes as well. As with most physical database design decisions, there are no clear rules of thumb here: the choice of which clustering to use depends on whether online queries or traditional range queries are more significant to the workload performance. These kinds of decision can be aided or even automated by workload analysis tools (e.g., Chaudhuri and Narasayya, 1998)).

3.2. Preferential data delivery: Online reordering

It is not sufficient for data delivery in an online query processing system to be random. It also must be user-controllable. This requirement was not present in traditional batch systems, and hence the techniques we discuss in this section do not have direct analogs in traditional systems.

A key aspect of an online query processing system is that users perceive data being processed *over time*. Hence an important performance goal for these systems is to present data of interest early on in the processing, so that users can get satisfactory results quickly, halt processing early, and move on to their next request. The "speed" buttons shown in the Online Aggregation interface in figure 1 are one interface for specifying preferences: they allow users to request preferential delivery for particular groups. The scrollbar of a spreadsheet is another interface: items that can be displayed at the current scrollbar position are of greatest interest, and the likelihood of navigation to other scrollbar positions determines the relative preference of other items.

To support preferential data delivery, we developed an *online reordering* operator that reorders data on the fly based on user preferences—it attempts to ensure that *interesting* items get processed first (Raman et al., 1999b). We allow users to dynamically change their definition of "interesting" during the course of a query; the reordering operator alters the data delivery to try and meet the specification at any time.

In order to provide user control, a data processing system must accept user preferences for different items and use them to guide the processing. These preferences are specified in a value-based, application-specific manner, usually based on values in the data items. The mapping from user preferences to the rates of data delivery depends on the performance goals of the application. This is derived for some typical applications in Raman et al. (1999b). Given a statement of preferences, the reorder operator should permute the data items at the source so as to make an application-specific *quality of feedback* function rise as fast as possible.

Since our goal is interactivity, the reordering must not involve pre-processing or other overheads that will increase runtime. Instead, we do a "best effort" reordering *without* slowing down the processing, by making the reordering concurrent with the processing. Figure 3 illustrates our scheme of inserting a reorder operator into a data flow. We can

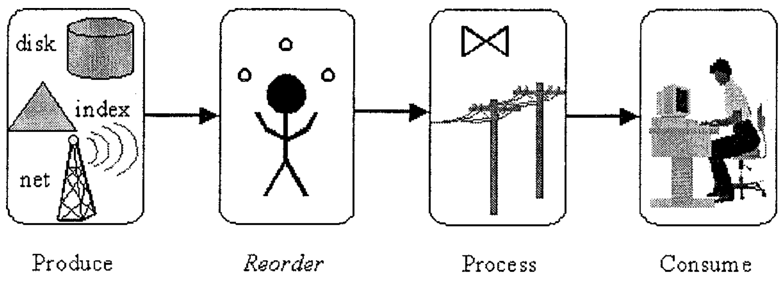


Figure 3. The reordering operator in context.

divide any data flow into four stages: the `produce` stage, the `reorder` stage, the `process` stage, and the `consume` stage. In the context of query processing, `Produce` represents an access method generating records. `Reorder` reorders the items according to the dynamically changing preferences of the consumer. `Process` is the set of operations that are applied "downstream" to the records—this could involve query plan operators like joins, shipping data across a slow network, rendering data onto the screen in data visualization, etc. `Consume` captures the user think-time, if any—this is mainly for interactive interfaces such as spreadsheets or data visualization. Since all these operations can go on concurrently, we exploit the difference in throughput between the `produce` stage and the `process` or `consume` stages to permute the items. For disk-based data sources, `Produce` can run as fast as the sequential read bandwidth, whereas `process` may involve several random I/Os which are much slower (Gray and Graefe, 1997). While the items sent out so far are being processed/consumed, `reorder` can take more items from `produce` and permute them.

The `reorder` operator tries to `put` as many interesting items as possible onto a main-memory buffer, and the `process` operator issues requests to `get` items from the buffer. `Process` decides which item to `get` based on its performance goals. `Reorder` uses the time gap between successive `gets` from the buffer (which may arise due to processing or consumption time) to populate the buffer with more interesting items. It does this either by using an index to fetch interesting items, or by aggressively prefetching from the input, spooling uninteresting items onto an auxiliary disk. Policies for management of the buffer and organization of the auxiliary disk are described in more detail in Raman et al. (1999b); the basic idea is to evict least-interesting items from the buffer, and place them into chunks on disk of records from the same group.

If the reorder operator can `get` records much faster than they can be `process`ed, then the reordering has two phases. In the first phase, reorder continually `gets` data, and tries to keep the buffer full of interesting items in the appropriate ratios, carefully spooling uninteresting items to chunks on the side disk. The second phase occurs when there is no more data to `get`; at this point, reorder simply *enriches* the buffer by fetching chunks of interesting tuples of interest from the side disk.

3.2.1. Index stride and database design issues. The Index Stride access method was first presented in Hellerstein et al. (1997); it works as follows. Given a B-tree index on the grouping columns,[5] 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 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 $[=k_1], \ldots, [=k_n]$ in a round-robin fashion. In order to capture user preference for groups, we do not actually use round-robin scheduling among the groups; rather we use lottery scheduling (Waldspurger and Weihl, 1995), assigning more "tickets" to groups of greater interest.

Index Stride can be used to support online reordering, because the reorder operator can `get` tuples in the appropriate ratios by simply passing its weighting of groups to the Index Stride access method. A drawback of using an index is that it involves many random I/Os. For many scenarios, Index Stride is significantly less efficient than simply running an online reordering operator over a table-scan. However for groups of very low cardinality Index Stride can be extremely beneficial. A hybrid of Index Stride and table-scan can be achieved via *partial indexes* (Seshadri and Swami, 1995; Stonebraker, 1989): an index is built over the small groups, and the reorder operator is run over a union of the small groups (in the index) and the larger groups (in the heap file). In a system without partial indexes like UDO, the hybrid scheme can be effected by explicitly partitioning the table into its rare and common groups, storing the rare groups in a separate table. Performance tradeoffs between Index Stride and online reordering of table-scans are presented in Raman et al. (1999b).

3.3. Ripple join algorithms

Up to this point, our discussion has focused on algorithms for delivering data from individual tables. These techniques are appropriate for SQL queries on single tables; they are also appropriate for simple spreadsheet-like systems built on files, as sketched in Section 2.2. In general, however, SQL queries often combine data from multiple tables—this requires relational join operators.

The fastest classical join algorithms (see, e.g., Graefe (1993)) are inappropriate for online query processing. Sort-merge join is blocking: it generates no output until it has consumed its entire input. Hybrid hash join (DeWitt et al., 1984) does produce output from the beginning of processing, but at a fraction of the rate at which it consumes its input. Moreover most commercial systems use Grace hash join (Fushimi et al., 1986) which is a blocking algorithm.

The only classical algorithm that is completely pipelining is nested-loops join, but it is typically quite slow unless an index is present to speed up the inner loop. Using nested-loops join in an online fashion is often more attractive than using a blocking algorithm like sort-merge join. But the absolute performance of the online nested-loops join is often unacceptably slow even for producing partial results—this is particularly true for estimating aggregates.

To see this, consider an online aggregation query over two tables, R and S. When a sample of R is joined with a sample of S, an estimate of the aggregate function can be produced, along with a confidence interval (Haas, 1997). We call this scenario the end of a *sampling step*—at the end of each sampling step in a join, the running estimate can be updated and the confidence interval tightened. In nested loops join, a sampling step completes only at the end of each inner loop. If S is the relation in the outer loop of the join in our example, then a sampling step completes after each full scan of R. But if R is of non-trivial size (as is often the case for decision-support queries), then the amount of time between successive sampling steps—and hence successive updates to the running estimate and confidence interval—can be excessive.

In addition to having large pauses between estimation updates, nested-loops join has an additional problem: it “samples” more quickly from one relation (the inner loop) than from

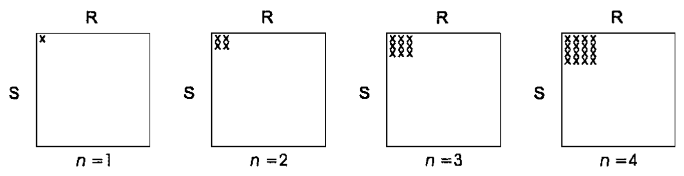


Figure 4. The elements of $R \times S$ that have been seen after n steps of a "square" ripple join.

the other. If the relation in the outer loop contributes significantly to the variance of inputs to the aggregation function, it can be beneficial to more carefully balance the rates of reading from the two relations.

To address these problems, we developed a new join algorithm for online query processing, called the *ripple join* (Haas and Hellerstein, 1999). In the simplest version of the two-table ripple join (figure 6), one previously-unseen random tuple is retrieved from each of R and S at each sampling step; these new tuples are joined with the previously-seen tuples and with each other. Thus, the Cartesian product $R \times S$ is swept out as depicted in the "animation" of figure 4. In each matrix in the figure, the R axis represents tuples of R, the S axis represents tuples of S, each position (r, s) in each matrix represents a corresponding tuple in $R \times S$, and each "x" inside the matrix corresponds to an element of $R \times S$ that has been seen so far. In the figure, the tuples in each of R and S are displayed in the order provided by the access methods; this order is assumed to be random.

The "square" version of the ripple join described above draws samples from R and S at the same rate. As discussed in Haas and Hellerstein (1999), it is often beneficial to sample one relation (the "more variable" one) at a higher rate in order to provide the shortest possible confidence intervals for a given aggregation query. This requirement leads to the general "rectangular" version of the ripple join[6] depicted in figure 5. The general algorithm with K (≥ 2) base relations $R_1, R_2, \ldots, R_K$ retrieves β_k previously-unseen random tuples from R_k at each sampling step for $1 \leq k \leq K$. (figure 5 corresponds to the special case in which $K = 2$, $\beta_1 = 3$, and $\beta_2 = 2$.) Note the tradeoff between interactivity of the display and estimation accuracy. When, for example, $\beta_1 = 1$ and $\beta_2 = 2$, more I/O's are required per sampling step than when $\beta_1 = 1$ and $\beta_2 = 1$, so that the time between updates to the confidence intervals is longer; on the other hand, after each sampling step the confidence interval typically is shorter when $\beta_1 = 1$ and $\beta_2 = 2$.

In Haas and Hellerstein (1999) we provide full details of ripple joins, and their interaction with aggregation estimators. We also include detailed discussion on the statistics behind

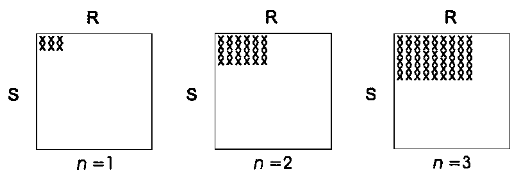


Figure 5. The elements of $R \times S$ that have been seen after n sampling steps of a "rectangular" ripple join with aspect ratio 3×2.

```
for (max = 1 to infinity) {
  for (i = 1 to max-1)
    if (predicate(R[i],S[max]))
      output(R[i],S[max]);
  for (i = 1 to max)
    if (predicate(R[max],S[i]))
      output(R[max],S[i]);
}
```

Figure 6. A simple square ripple join. The tuples within each relation are referred to in array notation.

tuning the aspect ratios to shrink the confidence intervals as fast as possible, while still attempting to meet a user's goal for interactivity. In addition, we demonstrate that ripple join generalizes pipelining nested-loops and hash joins. That leads to descriptions of higher-performance algorithmic variants on figure 6 based on blocking (analogous to blocked nested loops join), hashing (generalizing the pipelined hash join of Wilschut and Apers (1991), and indexes (identical to index nested loops).

The benefits of ripple join for enumeration (non-aggregation) queries are akin to the benefits of sampling for these queries: the running result of a ripple join arguably represents a "representative" subset of the final result, since it is made from sizable random samples from each input relation. In an enumerative query, the ripple join aspect ratio can be tuned to maximize the size of the output, rather than a statistical property.

3.4. Hash-based grouping

SQL aggregation queries can contain a GROUP BY clause, which partitions tuples into groups. This process can be done via sorting or hashing; in either case, the stream of tuples resulting from selections, projections and joins are grouped into partitions such that within a partition all tuples match on the GROUP BY columns. Sorting is a blocking operation and hashing is not (at least not for a reasonable number of groups), so it is important that an online aggregation system implement GROUP BY via hashing. Further discussion of unary hashing and sorting appears in Hellerstein and Naughton (1996).

4. End-to-end issues in online query processing

A collection of algorithms does not make a system; the various building blocks must be made to interact correctly, and consideration must be given to the end-to-end issues required to deliver useful behavior to users. In this section we discuss our experience implementing online query processing in Informix UDO. This implementation is more than a simple "plug-in": the various algorithms must be incorporated inside the server alongside their traditional counterparts, and must be made to work together correctly. In addition, interfaces must be added to the system to handle interactive behaviors that are not offered by a traditional system. In this section we describe interfaces to the system, and implementation details in composing query execution plans. We also discuss some of remaining challenges in realizing a completely online query processing system. There were some issues in implementing the estimators in a DBMS that are not clear from an algorithmic standpoint. We discuss these in Appendix A.

4.1. Client-server interfaces

A traditional database Application Programming Interface (API) accepts SQL queries as input, and returns a *cursor* as output; the cursor is a handle with which the client can request output tuples one at a time. A batch system may process for a long time before the cursor is available for fetching output, while an online system will make the cursor available almost immediately. Beyond this minor change in interaction lie more significant changes in the client-server API: online query processing allows for ongoing interaction between a client application and database server. Two basic issues need to be handled. First, the system needs to provide output beyond relational result records. Second, users can provide many kinds of input to the system while a query is running.

Before actually deciding on functions to add, we had to decide how applications would invoke our functions. We took two approaches to this problem. Our first solution was to extend Informix's Embedded SQL (ESQL) API, which is a library of routines that can be called from C programs. This was problematic, however, because implementing clients in C is an unpleasant low-level task.

We preferred to use a combination of Windows-based tools to build our application: the interface of figure 1 consists of a few hundred lines of Visual Basic code that invoke Informix's MetaCube client tool, and also utilize a Microsoft Excel spreadsheet that provides the table and graph widgets. These Windows-based tools do not work over ESQL; instead they communicate with the DBMS via the standard Open Database Connectivity (ODBC) protocol. Unfortunately ODBC is part of the MS Windows operating systems, and its API is not extensible by third parties. Hence we developed a second architecture that conforms to the traditional ODBC paradigm: all input is done via SQL queries, and output via cursors. Sticking to this restricted interface has a number of advantages: it enables the use of ODBC and similar connectivity protocols (such as Java's JDBC), and it also allows standard client tools to be deployed over an online query processing system.

Given this architecture, we proceed to discuss our CONTROL API for output and input in turn.

4.1.1. Output API. Online enumeration queries provide the same output as traditional database queries; the only distinction is that they make a cursor available for fetching almost instantly. Online aggregation queries provide a form of output not present in traditional systems: running estimates and confidence intervals for results. We chose to have these running outputs appear as regular tuples, with the same attributes as the actual output tuples. This means that a single SQL query is issued, which returns a stream of tuples that includes both estimates and results. For example, the query of Section 2.1 can be issued as:

```
  SELECT ONLINE AVG(grade), CONFIDENCE_AVG(grade, 95)
    FROM enroll
GROUP BY college;
```

The addition of the ONLINE keyword and the CONFIDENCE_AVG function to the query can be done automatically by the client application, or explicitly by a user. In this example, CONFIDENCE_AVG is a user-defined aggregate function (UDA) that returns a confidence interval "half-width" ϵ corresponding to the probability given by the second argument to the aggregate (i.e., the estimate is $\pm\epsilon$ of the correct average with 95% probability). The interface of figure 1 shows the ϵ values displayed as "error bars" in a graph. Immediately after issuing this query, a cursor is available for fetching results. In a traditional system, this query would produce one tuple per `college`. In our system, the query produces multiple tuples per `college`, each representing an estimate of the average `grade` along with a confidence interval half-width. If the query is run to completion, the *last* tuple fetched per `college` represents the accurate final answer.

Any ODBC client application can handle this sequence of tuples. In particular, a standard text-based client works acceptably for online aggregation—estimates stream across the screen, and the final results remain on the last lines of the output. This proved to be our client of choice while developing the system. Our new interfaces (e.g., figure 1) overwrite old estimates with updated ones by assigning groups to positions in a tabular display.

4.1.2. Input API. Users should be able to control a running query by interacting with a client, which passes user input on to the server. As a concrete example from our implementation, the user should be able to speed up, slow down, or stop groups in an online aggregation query. Similarly, the spreadsheet widget should be able to express a preference for records that are in the range currently being shown on screen.

For clients implemented in C over our enhanced ESQL, input to the server is direct: the client simply invokes API functions that we have added to ESQL, such as `pause_group(x)`, `speed_up_group(x)`, etc. Such invocations can be interleaved arbitrarily with requests to fetch data from a cursor. For clients communicating via ODBC rather than ESQL, direct invocation of API functions is not an option. For this scenario, we were able to leverage the Object-Relational features of UDO in a non-standard manner. In an object-relational system, an SQL query can invoke user-defined functions (UDFs) in the SELECT statement. We added UDFs to our database that invoke each of our API functions. Hence the client can issue a query of the form

```
    SELECT PAUSE_GROUP(x);
```

and get the same effect as a client using ESQL. Clients can interleave such "control" queries with requests to fetch tuples from the main query running on the database. Originally we considered this an inelegant design (Hellerstein et al., 1997), because it significantly increases the code path for each gesture a user makes. In practice, however, the performance of this scheme has been acceptable, and the ease of use afforded by ODBC and Windows-based 4GLs has been beneficial.

4.1.3. Pacing. Online aggregation queries can return updated estimates quite frequently—for a single-table aggregation query like that of Section 2.1, a new estimate is available for every tuple scanned from the table. However, many clients use complex graphical interfaces or slow network connections, and are unable to fetch and display updates as fast as the database can generate them. In these scenarios it is preferable for the system to produce a new estimate once for every $k \gg 1$ tuples. This prevents overburdening the client. There is a second reason to avoid producing too many updated estimates. Typically, SQL query output is synchronized between the server and client: after the client fetches a tuple, the server is idle until the next request for a tuple from the client. This architecture is an artifact of batch system design, in which most of the server processing is done before the first tuple can be fetched. In an online aggregation query, time spent handling updated estimates at the client is time lost at the server. Hence it pays to amortize this lost time across a significant amount of processing at the server—i.e., one does not want to incur such a loss too often.

A solution to this problem would be to reimplement the threading model for online aggregation so that the query processing could proceed in parallel with the delivery of estimates. To achieve this, the aggregation query could run as one thread, and a separate request-handling thread could process fetch requests by "peeking" at the state of the query thread and returning an estimate. Rather than rearchitecting Informix's structure in this way, we chose to maintain Informix's single-thread architecture, and simply return an output tuple for every k tuples that are aggregated. We refer to the number k as the *skip factor*. The skip factor is set by the user in terms of time units (seconds), and we translate those time units into a skip factor via a dynamic feedback technique that automatically calibrates the throughput of client fetch-handling.

4.2. Implementing online query operators

The algorithms of Section 3 were designed in isolation, but a number of subtleties arise when integrating them into a complete system. In this section, we discuss issues in implementing online query processing operators in Informix.

4.2.1. Online reordering. Like the early System R DBMS prototype (Astrahan et al., 1976), Informix UDO is divided into two parts: a storage manager called RSAM, and a query optimization/execution engine implemented on top of RSAM. In our prototype, we decided to implement all our algorithms entirely above RSAM. This decision was motivated by considerations of programming complexity, and by the fact that RSAM has a well-defined, published API that we did not want to modify (Informix, 1998).

Implementing Index Stride above RSAM resulted in some compromises in performance. RSAM presents interfaces for scanning relations, and for fetching tuples from indexes. The index-based interface to RSAM takes as arguments a relation R, an index I, and a predicate p, and returns tuples from R that match p. To amortize I/O costs, an efficient implementation of Index Stride would fetch tuples a disk block at a time: one block of tuples from the first group, then one from the second, and so on. Since the RSAM interface is tuple-at-a-time, there is no way to explicitly request a block of tuples. In principle the buffer manager could solve this problem by allocating a buffer per group: then when fetching the first tuple from a block it would put the block in the buffer pool, and it would be available on the next fetch. Unfortunately this buffering strategy does not correspond naturally to the replacement policies in use in a traditional DBMS. The performance of Index Stride could be tuned up by either implementing a page-based Index Stride in RSAM, or enhancing the buffer manager to recognize and optimize Index Stride-style access.

4.2.2. Ripple join. Performance is not the only issue that requires attention in a complete implementation; we also encountered correctness problems resulting from the interaction of ripple join and our data delivery algorithms. As described in figure 6, ripple join rescans each relation multiple times. Upon each scan, it expects tuples to be fetched in the same order. Traditional scans provide this behavior, delivering tuples in the same order every time. However, sampling-based access methods may not return the same order when restarted. Moreover, the online reordering operator does not guarantee that its output is the same when it is restarted, particularly because users can change its behavior dynamically.

To address this problem, we introduce a cache above non-deterministic operators like the reorder operator, which keeps a copy of incoming tuples so that downstream ripple join operators can "replay" their inputs accurately. This cache also prevents ripple join from actually repeating I/Os when it rescans a table—in this sense the cache is much like the "building" hash table of a hash join.

In ordinary usage, we expect online queries to be terminated quickly. However, if a query is allowed to run for a long period of time, the ripple caches can consume more than the available memory. At that point two options are available. The first alternative is to allow the cache to become larger than available memory, spilling some tuples to disk, and recovering them as necessary—much as is done in hash join algorithms. The second alternative is to stop caching at some point, but ensure that the data delivery algorithms output the same tuples each time they are restarted. For randomized access methods this can be achieved by storing the seed of the random number generator. For user-controllable reordering, this requires remembering previous user interactions and "replaying" them internally when restarting the delivery. We implemented the latter technique in Informix.

An additional issue arises when trying to provide aggregation estimates over ripple joins. As noted in Section 3.3, estimates and confidence intervals over ripple joins can be refreshed when a sample of R has been joined with sample of S—this corresponds to a "corner" in the pictures of figure 4. It therefore makes sense for the estimates in a join query to be updated at every k'th corner for skip-factor k. However it is possible that the tuple corresponding to a corner may not satisfy the WHERE clause; in this case, the system does not pass a tuple up the plan tree, and the estimators are not given the opportunity to produce a refined output tuple. To avoid this loss of interactivity, when a ripple join operator reaches a "corner", it returns a dummy, "null tuple" to the aggregation code. These null tuples are handled specially: they are not counted as part of the aggregation result, but serve as a trigger for the aggregation code to update estimates and confidence intervals. Null tuples were easy to implement in UDO as generalizations of tuples in *outer join* operators: an outer join tuple has null columns from a proper subset of its input relations, whereas a null tuple has null columns from all of its input relations.

4.3. Constructing online query plans

Like any relational database system, an online query processing system must map declarative SQL queries to explicit query plans: this process is typically called *query optimization*. Online processing changes the optimization task in two ways: the performance goals of an online system are non-traditional, and there are interactive aspects to online query processing

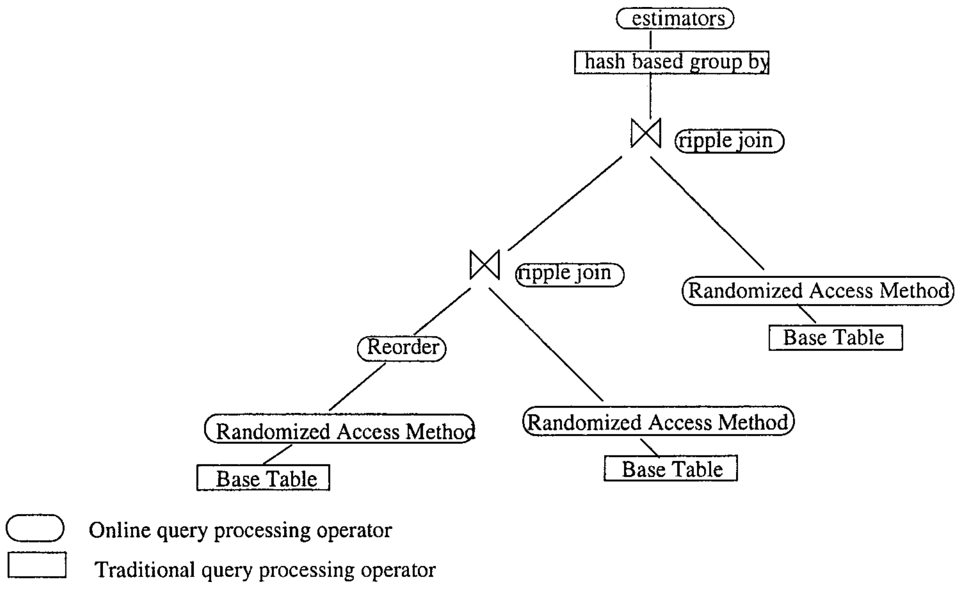


Figure 7. An online query plan tree.

that are not present in traditional databases. For example, for any given query we want to minimize not the time to completion but rather the time to "reasonable" accuracy. Online query optimization is an open research area we have only begun to address in the CONTROL project. In our first prototype we modified Informix UDO's optimizer and query executor to produce reasonable online query plans, without worrying about optimality. In this section we describe our implementation, and raise issues for future research. As motivation, figure 7 shows a complete online query plan, including a number of online query processing operators.

4.3.1. Access method selection. The choice of access methods dictates the extent to which a user can control data delivery. The choices of access methods lie on a spectrum of controllability. At one extreme, sequential scan provides no control, but it is fast and does not require indexes. At the other extreme, Index Stride provides complete control over the processing rates from different groups in a GROUP BY query, but it is slow because it performs many random I/Os. Online reordering lies between these two extremes. It runs almost as fast as a sequential scan (except for a small CPU overhead), but only does a best-effort reordering: if the user has a high preference for extremely rare groups, online reordering can deliver tuples only as fast as it scans them. In our implementation, we modified the UDO optimizer to handle SELECT ONLINE queries with a simple heuristic: if there is a clustered index on the GROUP BY column we use an Index Stride access method, and otherwise we use a sequential scan access method and follow it with online reordering.

An open issue is to allow for reordering when there are GROUP BY clauses over columns from multiple tables: in this case the reordering operators must work in concert with ripple join to produce groups in the appropriate ratio. Currently, we add a reordering operator to only the leftmost table in the query plan that has a GROUP BY. We intend to address this issue in future work.

4.3.2. Join ordering. Unlike most traditional join algorithms, ripple joins are essentially symmetric: there is no distinction between the "outer" and "inner" relations. Hence for a tree of ripple joins, the join ordering problem is quite simple: all that is required is to tune the aspect ratios described in Section 3.3; this is done dynamically as described in Haas and Hellerstein (1999). This view of the problem is overly simplistic, since there is a choice of ripple join variants (block, index or hash), and the join order can affect this choice. For example, in a query over relations R, S and T, there may be equality join clauses connecting R to S, and S to T, but no join clause connecting R to T. In this case the join order $(R \bowtie T) \bowtie S$ cannot use a hash- or index-ripple join since we need to form the cross-product of R and T; by contrast, the join order $(R \bowtie S) \bowtie T$ can use two hash joins.

In our current prototype, we let UDO decide on the join ordering and join algorithms in its usual fashion, optimizing for batch performance. We then post-process the plan, inserting reorder operators above access methods, converting hash joins to hash-ripple joins, index nested-loops joins to index-ripple joins, and nested-loops joins to block-ripple joins.

In ongoing related work, we are studying query optimization in the online context. We have developed a prototype of a continuous optimization scheme based on the notion of an *eddy* (Avnur and Hellerstein, 2000). An eddy is an n-ary query operator interposed between n input tables and $n - 1$ joins, which adaptively routes tuples through the joins. The output of each join is sent back to the eddy for further processing by other joins. By interposing an eddy in the data flow, we change the join order of the query for every tuple as it passes through the query plan. Eddies hold promise not only for online query processing, but for any pipelined set of operations that operate in an uncertain environment—any scenario where user preferences, selectivities or costs are unpredictable and dynamic.

4.4. Beyond select-project-join

The majority of our discussion up to this point has concentrated on simple SQL query blocks—consisting of selection, projection and join—followed by grouping and aggregation. A complete online SQL system must handle other variations of SQL queries as well. We have not implemented all of SQL in an online fashion: some of it is best done by the user interface, and some is future research. We detail these issues here.

4.4.1. Order by, Having. The ORDER BY clause in SQL sorts output tuples based on the values in some column(s). Implementing an online ORDER BY at the server is somewhat pointless: at any time the server could construct an ordered version of the output so far, but re-shipping it to the client for each update would be wasteful. Instead, we believe that ORDER BY is best implemented at the client, using an interface like those described in Section 2.2. Note that the ORDER BY clause is often used to get a "ranking", where a few representative rows near the top will suffice (Carey and Kossmann, 1997; Chaudhuri and Gravano, 1996). In this case an online spreadsheet interface may be more appropriate to get a typical smattering of the top few rows; the strict total ordering of SQL's ORDER BY may over-constrain the request at the expense of performance.

SQL's HAVING clause filters out groups in a GROUP BY query, based on per-group values—either aggregation functions or grouping columns. Consider a version of the query of Section 2.1 that contains an additional clause at the end, `HAVING AVG(grade) > 3.0`; this is an example of a HAVING clause with an aggregate in it. Over time, some college's estimated average grade may go from a value greater than 3.0 to one less than 3.0. In that case, the client *must* handle the deletion of the group from the display; the server does not control how the client displays previously-delivered tuples. Since the HAVING clause has to be handled appropriately by clients, we did not implement a HAVING-clause update scheme at the server.

4.4.2. Subqueries and other expensive predicates. SQL allows queries to be nested in the FROM, WHERE and HAVING clauses. Queries containing such *subqueries* can sometimes be rewritten into single-level queries (see Leung et al. (1998) or Cherniack (1998) for an overview), but there are cases where such subqueries are unavoidable. The problem with such subqueries is that they force batch-style processing: the outer query cannot produce output until the subquery is fully processed. For *correlated* subqueries, the problem is exacerbated: the outer query must complete processing (at least!) one subquery before each tuple of output. A similar problem arises with *expensive* user-defined functions in an object-relational system (Hellerstein, 1998b; Chaudhuri and Shim, 1996): at least one expensive computation must be completed before each tuple is passed to the output.

To date, we have not addressed the online processing of SQL queries with subqueries. One perspective on subqueries in an online system is to view them as the composition of two online systems: the subquery Q_0 is an input to the outer query Q, and the goal is to produce running estimates for $Q(Q_0(\ ))$. The optimization of such compositions has been studied in the AI community as *anytime algorithms* (Zilberstein and Russell, 1996), but it is not immediately clear whether that body of work is applicable to multiset-based operators like SQL queries. Recently (Tan et al., 1999) have suggested that queries with subqueries be executed as two threads, one each for the outer and inner query blocks, with the outer block executing based on estimated results from the inner block. The user controls the rate of execution of these two threads, and thereby the accuracy of the answers. While this approach is promising, it works only when the subquery simply computes an aggregate, and the predicate linking the outer block to the inner block is a comparison predicate; it will be interesting to see if this approach can be extended to other predicates. Various techniques are available for efficiently executing correlated subqueries and expensive functions during processing (Rao and Ross, 1998; Hellerstein and Naughton, 1996); these ameliorate but do not solve the batch-style performance.

5. A Study of CONTROL in action

In this section, we illustrate the performance of interactive data analysis in Informix UDO via a sample usage scenario. Our goal is to demonstrate the benefits of online query processing: continually improving feedback on the final result, and interactive control of the processing. We also give an example which combines a ripple join and online reordering to show the completeness of our system against all "select-project-join" queries. Note that this is not intended as an robust analytic study of performance; comparisons of online algorithms

```
Query 1:  SELECT ONLINE avg(o_totalprice), confidence_avg(o_totalprice)
          FROM order;

Query 2:  SELECT ONLINE avg(o_totalprice), o_orderpriority, confidence_avg(o_totalprice)
          FROM order
          WHERE NOT EXISTS (SELECT * FROM lineitem WHERE o_orderkey = l_orderkey
                        AND l_shipmode = 'AIR')
          GROUP BY o_orderpriority;

Query 3:  SELECT ONLINE avg(l_extendedprice), l_shipmode, confidence_avg(o_totalprice)
          FROM order, lineitem
          WHERE o_orderkey = l_orderkey
          GROUP BY o_orderpriority;
```

Figure 8. Queries used in our data analysis session.

under different distributions and parameter settings are given in Hellerstein et al. (1997), Haas and Hellerstein (1999) and Raman et al. (1999).

We scale up all numbers by an undisclosed factor to honor privacy commitments to Informix Corporation while still allowing comparative analysis (hence time is expressed in abstract "chronons"). We give rough figures for the actual wall-clock times in order to show the interactivity of CONTROL.[7] We run each of our experiments only until reaonable accuracy is reached and the analyst switches to the next query.

Our data analysis session uses three queries (see figure 8) against the TPC-D database with a scale factor of 1 (about 1 Gb of data) (Transaction Processing Council). Since online reordering is more difficult for skewed distributions of group values, we use a Zipfian distribution ($1{:}\frac{1}{2}{:}\frac{1}{3}{:}\frac{1}{4}{:}\frac{1}{5}$) of tuples across different order priorities in the `Order` table. In Queries 2 and 3 we group by `o_orderpriority`, to demonstrate the effectiveness of online reordering even in the presence of the skewed distributions commonly found in real data.

Our scenario begins with an analyst issuing Query 1, to find the average price of various orders. Although the base table takes a while to scan (it is about 276MB), the analyst immediately starts getting estimates of the final average. Figure 9 shows the decrease in the

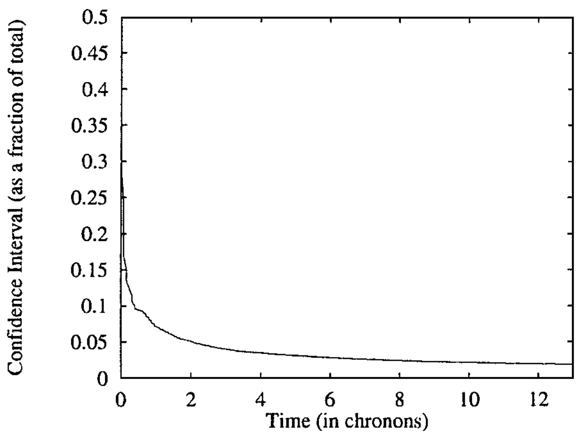


Figure 9. Decrease in the confidence interval for Query 1.

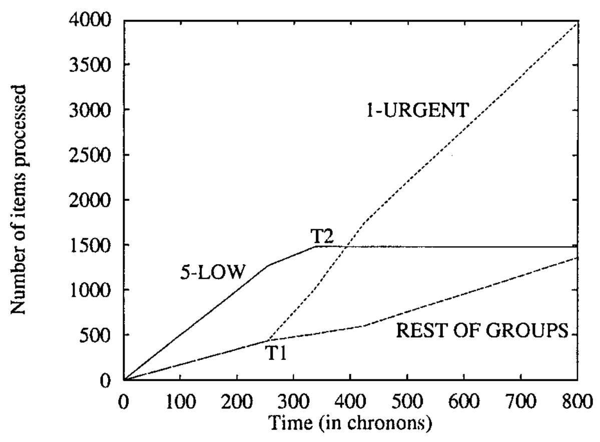


Figure 10. Number of tuples processed for different groups for Query 2.

confidence interval for the average as a function of time. We see that very accurate results are obtained within 12 chronons. On the other hand, a non-online version of the same query took 1490 chronons to complete, or over two orders of magnitude higher! To give an idea of the interactivity, the online version took about a second to reach reasonable (confidence interval <2%) accuracy whereas the non-online version took on the order of a few minutes to complete.

After 12 chronons, the analyst looks at the estimates and feels (based on domain knowledge) that they "are not quite right". Wondering what is happening, the analyst hypothesizes that the prices may be skewed by the presence of some expensive air shipments. Hence the next query issued is Query 2, which finds the average price of orders that do not have any item shipped by air. Notice that with a traditional DBMS, the analyst would not be able to try out this alternative until the first query had completed and returned a result.

Since Query 2 involves a non-flattenable subquery, Informix cannot pick a ripple join. It instead scans `Order`, using the index on `Lineitem` to evaluate the subquery condition. It chooses online reordering to permit user control over the processing from different groups of `Order`. Almost immediately after Query 2 starts running, the analyst decides, from the estimates, that the group with `o_orderpriority` as 5-LOW is more interesting than others, and presses the "speed up" button (see figure 1) to give 5-LOW a preference of 5, compared to 1 for the rest of the groups. Figure 10 shows the number of tuples processed for different groups as a function of time. After some time (point T1 in the figure), 5-LOW's confidence interval has narrowed sufficiently, and the analyst shifts interest to group 1-URGENT, giving it a preference of 5, and reducing that of all other groups to 1.[8] We see that online reordering is able to meet user preferences quite well, despite the interesting groups being rare. In contrast, a sequential scan will actually process the interesting group 5-LOW more slowly than others, because it is uncommon.

Figure 11 shows the rate at which the confidence intervals for the estimates decrease for the same query. We see that the reorder operator automatically adjusts the rates of processing so as to decrease confidence intervals based on user preferences. Also note that at T1 when

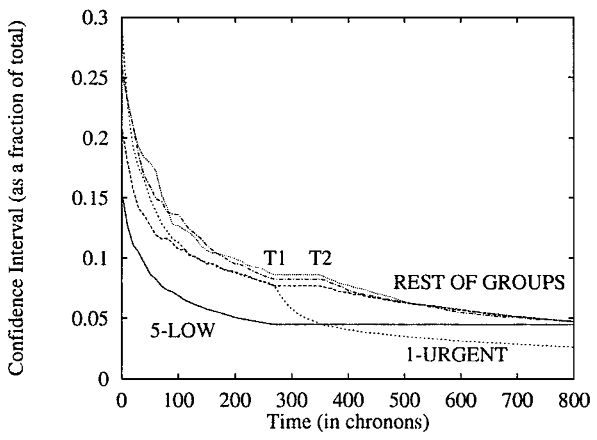


Figure 11. Decrease in confidence interval for different groups for Query 2 with online reordering. Note that the confidence interval for 5-LOW decreases sharply at the beginning, and that the preference for 1-URGENT drops sharply at T1.

the preference for 1-URGENT is increased, the other groups stop for a while (until T2) (in fact 5-LOW plateaus for so long after T2 that the analyst sees no further updates for this group before quitting at 800 chronons; this same behavior is also seen in Figure 10). Likewise, there is a sudden spurt in the processing of tuples from 1-URGENT from T1 until T2. This happens because, reorder realizes that it has processed very few tuples from 1-URGENT with respect to the new preferences, and starts to compensate for it. After T2, the different groups get processed according to their preferences. This compensation arises because of the performance goal that we use: we view the preferences as a weight on the importance of a group, and try to make the weighted average confidence interval across all groups decrease as fast as possible. The motivation for this goal, as well as performance for several other goals, is given in greater detail in Raman et al. (1999).

The mechanism for doing the reordering to decrease the confidence intervals in an optimal manner is explained in Raman et al. (1999). In that paper we also present results that show that online reordering performs well against a variety of data distributions, processing costs, and user preference change scenarios. We also show its efficacy in reordering for an online enumeration application, and for speeding up traditional batch query processing.

After seeing the estimates for 800 chronons, the query does not seem to exhibit any interesting trends. Hence the analyst decides to proceed in a different direction, and submits a new query (Query 3), which drills down on the average lineitem prices, grouping by the `o_order_priority`. Online reordering has helped the analyst to narrow down the estimates for the group of interest and drill down much before Query 2 completes; the entire query takes 58730 chronons to run.

Query 3 needs to access the `Order` *and* `Lineitem` tuples, and since the join predicate is equality it can be evaluated with a hash ripple join. Again, we use online reordering on `Order` to allow user control over the processing of tuples from different order priorities. This query involves running a hash-ripple join over a reorder operator.

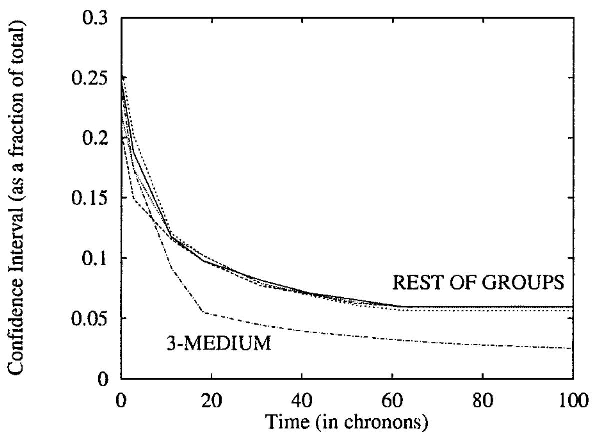


Figure 12. Decrease in confidence interval for different groups for Query 3 with online reordering. Note that the confidence interval for 3-MEDIUM decreases much faster than the others.

Figure 12 shows the decrease in confidence intervals for different groups over time. The analyst increases the preference for group 3-MEDIUM to 20 at the beginning of the processing, and this is reflected in the faster decrease in its confidence interval. We see that within 100 chronons, a reasonably accurate picture emerges, especially for 3-MEDIUM. In contrast, a regular hash join took 68210 chronons to execute this query, which is about three orders of magnitude slower! Notice that this query involves running a hash ripple join over a reorder operator. Hence the join replays tuples from `Lineitem` using a cache of the previous results from reorder as explained in Section 4.2.2. Further comparisons of ripple joins are presented in Haas and Hellerstein (1999) including a comparison of hash, index, and block ripple joins with batch algorithms, and a study of the importance of tuning join aspect ratios for shrinking confidence intervals.

6. Conclusions and future work

We have implemented a prototype online query processing system in Informix UDO, which supports *ad-hoc* select-project-join queries with grouping and aggregation. Our prototype cannot be considered “shippable” software, but it is fairly complete. The set of algorithms we developed provides interactive processing for a large family of SQL queries, and we were able to address most of the details required to implement a usable general-purpose system. We have been able to interface our system with 4GL-based client tools with little difficulty, resulting in quite serviceable online data analysis applications.

A number of lessons emerged in our work. Most significantly, we found that extending an SQL engine to provide online query processing is neither trivial nor overly daunting. A full implementation does require extending the database engine from within, and extensions are required from the access methods through the query operators, optimizer, and client API. But most of the changes are analogous to facilities already present in a complete SQL engine, and are therefore relatively simple to add.

We believe that many of our lessons in extending the system are pertinent outside the specific domain of online query processing. For example, any extension of a database system to handle new interactions will require some of the same API considerations mentioned in this paper. Similarly, additions of new access methods entail new considerations in database design. In addition, the issues raised by *composing* algorithms into large query plans are generally important, and perhaps too often neglected outside the context of relational database systems. Because these kinds of issues are often ignored in initial publications, new data analysis techniques are often passed over by practitioners. On occasion these pragmatic issues can also become rather interesting research problems, which would not naturally arise outside the context of a full implementation.

The most significant lesson for us resulted from the leverage we got from cross-disciplinary synergy: in our case, the mixture of statistics, computer systems and user interfaces. Our work was motivated by user interface goals; these mapped to system performance requirements, which were in turn clarified by optimizing for statistical metrics. Ripple join is perhaps the clearest example of this synergy: in order to achieve a balance between interactivity and output quality for multi-table queries, we developed a new query processing algorithm, which led to a very non-standard dynamic optimization paradigm (adjusting aspect ratios) based on variance of input to statistical estimators. We believe that this kind of cross-disciplinary synthesis is a fertile approach for developing future computer systems.

A number of issues remain as future work in online query processing. We have barely begun to study the user interface issues involved in online data analysis; much work remains to be done in understanding and constructing usable applications, especially for online enumeration and visualization. In the systems arena, we are extending our results to work in a parallel context, which entails algorithmic work as well as statistical work—e.g., parallel ripple joins are akin to stratified sampling techniques, which affects the estimators and confidence intervals for online aggregation. It would also be helpful to find general techniques for handling subqueries—many typical decision-support queries contain subqueries (e.g. the TPC-D benchmark queries (Transaction Processing Council)), and some of those cannot be rewritten away.

More generally, we are interested in the way that interactive techniques change the process of data analysis and knowledge discovery: when long-running operations become more interactive, the distinction between so-called "automated" techniques (e.g., data mining) and user-driven techniques (data visualization, SQL) are significantly blurred. The implications are likely to go beyond the obvious issues of faster interaction, to suggest new and more natural human-computer interactions for data analysis.

Appendix: A. Issues in implementing aggregate estimation

Online aggregation queries raise additional complications, since query processing must be integrated with estimation techniques. While this paper does not dwell on the statistics involved in our estimators, we do describe some of the system implementation issues that they raise.

A.1. Choice of confidence intervals

Three different classes of confidence intervals for online aggregation are detailed in (Haas, 1997). *Deterministic* confidence intervals give strict bounds on the error of an estimator, using bounds on the cardinalities of the tables and on the values of the input to the aggregation function. *Conservative* estimators are based on Hoeffding's inequality (Hoeffding, 1963), and can be used for samples of any size, but appear inapplicable to certain aggregation functions like STDDEV. *Large sample* estimators are based on Central Limit Theorems, and can be used on large samples (where large can be as small as 20–30 tuples) to provide quite tight confidence intervals. All three of these estimators can be extended to work on the cross-product samples provided by ripple joins (Haas and Hellerstein, 1999). We prefer to use large-sample confidence intervals because they are typically much tighter than deterministic or conservative estimates (Haas, 1997) and apply to a wider range of aggregate functions. However, there are two scenarios in which large-sample confidence intervals are inappropriate.

First, large sample confidence intervals fluctuate wildly at the beginning of a query, until a "large enough" sample has been processed. In order to avoid misleading the user, conservative confidence intervals should be used at first, until a large sample has been obtained. A reasonable choice might be to switch to large-sample confidence intervals only after 40 or 50 tuples have passed the query's WHERE clause. For those aggregates that cannot use conservative confidence intervals, it makes sense to simply postpone producing confidence intervals for that period. In general it is difficult to robustly decide when the Central Limit Theorem "kicks in" and one can switch estimators; in practice the sample sizes resulting from "big picture" database queries become large enough very quickly. Second, towards the end of processing a query large-sample confidence intervals are too conservative. Even after all tuples have been processed, a large-sample confidence interval will have a finite, although small, width. Thus towards the end of a query it is appropriate to switch to using deterministic confidence intervals. This switch can be done as soon as the deterministic interval is tighter than the large-sample interval.

In most situations, we expect user to stop processing online queries well before they complete, so the issue of deterministic confidence intervals may seem irrelevant. However deterministic confidence intervals can get used quite often because of reordering in GROUP BY queries. If a user is especially interested in a particular group, they will speed up that group's delivery to the point where it all tuples from the group will get fetched quickly; this is analogous to the "end-of-query" scenario for that group. If large-sample confidence intervals were used, that group could have a wide confidence interval even though it was finished.

A.2. Interaction of reordering and aggregate estimation

Reordering the data delivery biases the order in which tuples are read from different groups; in this sense it does not produce unbiased random samples. However, a key property of our reordering methods is that, *within* a group, reordering does not affect the order of tuples produced, and hence unbiased random samples are guaranteed per group. Since the aggregate function for each group in a GROUP BY query is estimated independently, no bias is involved in the estimation.

A.3. Calculation of quantities for estimation

A number of quantities are required as inputs to our estimators and confidence-interval computations. We briefly describe how we derive these in our implementation.

First, the estimators we use require knowledge of the actual number of tuples read from each table *before* any selections are applied. Obtaining this information required us to modify the RSAM layer slightly, since RSAM only returns tuples that satisfy relevant selections in the query's WHERE clause. This was the only situation where we added code in the RSAM layer.

Second, a few of our estimators and confidence interval computations require knowledge of the cardinalities of individual groups in a table (Haas, 1997; Hass and Hellerstein, 1999). To obtain this information for an Index Stride, we require a histogram on the GROUP BY column; a more natural solution would be to use a *ranked* index (Knuth, 1973; Antoshenkov, 1992; Aoki, 1998), but this is not available in most commercial databases. In the first phase of the reordering operator described at the end of 3.2, we can estimate a group's cardinality as the fraction of tuples from that group in the currently scanned sample. The statistics in the confidence intervals need to take the error of this estimation into account (Haas, 1997; Hass and Hellerstein, 1999); the statistical details remain as future work. In the second phase of the reordering operator, the exact cardinalities of all groups are known, so our usual estimation techniques apply.

Third, deterministic confidence intervals also require upper and lower bounds on the inputs to the aggregation functions. If the input to the function is a column (e.g. `AVG(enroll.grade)`), these bounds are typically available from the system catalogs. If the input is more complex (e.g. `AVG(f(enroll.grade)` for some UDF `f`) the only bounds available are those of the data type of the input (e.g. MAXINT is an upper bound for integers).

Acknowledgments

Bruce Lo implemented our user interface for online aggregation, and was involved in early development of our Informix extensions. Chris Olston and Andy Chou developed the CLOUDS system we describe in Section 2.3, and Christian Hidber developed the CARMA data mining algorithm described in Section 2.4. We are indebted to the CONTROL group at Berkeley for many interesting discussions: Andy Chou, Christian Hidber, Bruce Lo, Chris Olston, Tali Roth and Kirk Wylie. We are also grateful to Peter Haas of IBM Almaden Research for his many contributions to this work. Our relationship with Informix was made possible by Mike Stonebraker and Cristi Garvey; this kind of close academic/industrial collaboration is all too rare. At Informix, we received great help from Chih-Po Wen, Robyn Chan, Paul Friedman, Kathey Marsden, and Satheesh Bandaram. Helen Wang and Andrew MacBride contributed to an earlier Postgres-based prototype of online query processing. This work was supported by a grant from Informix Corporation, a California MICRO grant, NSF grant IIS-9802051, and a Sloan Foundation Fellowship. Computing and network resources for this research were provided through NSF RI grant CDA-9401156.

Notes

1. Continuous Output and Navigation Technology with Refinement On Line.
2. The other SQL aggregates, `MIN` and `MAX`, are "needle-in-a-haystack" scenarios that cannot be satisfied via sampling schemes like online aggregation: the minimum or maximum value may not appear until the last tuple is fetched. However, related user-defined-aggregates are amenable to online aggregation: e.g., `99-percentile`, which displays a tuple that can be said to be within ϵ of the 99'th percentile with some confidence.
3. The term *clustering* in relational databases is distinct from the statistical notion of clustering. A table in a database is said to be clustered on a column (or list of columns) if it is stored on disk in ascending order of that column (or ascending lexicographic order of the list of columns).
4. In order for this scheme to work, one must declare the function $f(\ )$ to SQL as being "NOT VARIANT". This informs the system that the value of $f(\ )$ for a tuple will not change over time, and the index can serve as a cache of the $f(\ )$ value for each tuple. This might seem counterintuitive since $f(\ )$ generates random numbers, but the point is that the *use* of the random numbers is static: once they define an initial random ordering of the table, that ordering is static until the table is reclustered.
5. Index Stride is naturally applicable to other types of indices as well, but we omit discussion here.
6. The name "ripple join" has two sources. One is shown in the pictures in figures 4 and 5—the algorithm sweeps out the plane like ripples spreading in a pond. The other source is the rectangular version of the algorithm, which produces "Rectangles of Increasing Perimeter Length".
7. We ran all experiments on a lightly-loaded dual-processor 200 MHz UltraSPARC machine running SunOS 5.5.1 with 256MB RAM. We used the INFORMIX Dynamic Server with Universal Data Option version 9.14 which we enhanced with online aggregation and reordering features. We used a separate disk for the side-disk for online reordering. We used a statistical confidence parameter (Haas, 1997) of 95% for our large sample confidence intervals. Note that we did not bother to tune our Informix installation carefully, since our online results were already performing sufficiently well. The performance of online and batch results presented here are not necessarily indicative of the peak performance available in a well-tuned installation, and readers should not use this study to extrapolate about the performance of Informix UDO.
8. To ensure that preferences are changed at a fixed, repeatable point in our experiments, we modified the *reorder* operator to read in the preference change points from a configuration file instead of from the GUI of figure 1.

References

Agrawal, R. 1997. Personal communication.

Agrawal, R. and Srikant, R. 1994. Fast algorithms for mining association rules. In Proc. 20th International Conference on Very Large Data Bases, Santiago de Chile, September 1994.

Aiken, A., Chen, J., Stonebraker, M., and Woodruff, A. 1996. Tioga-2: A direct-manipulation database visualization environment. In Proc. 12th IEEE International Conference on Data Engineering, New Orleans, February 1996.

Antoshenkov, G. 1992. Random sampling from pseudo-ranked B+ trees. In Proc. 18th International Conference on Very Large Data Bases, Vancouver, August 1992.

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

Aoki, P.M. 1998. Generalizing "search" in generalized search trees. In IEEE International Conference on Data Engineering, Orlando, February 1998.

Astrahan, M., Blasgen, M., Chamberlin, D., Eswaran, K., Gray, J., Griffiths, P., King, W., Lorie, R., McJones, P., Mehl, J., Putzolu, G., Traiger, I., Wade, B., and Watson, V. 1976. System R: Relational approach to database management. ACM Transactions on Database Systems, 1(2):97–137.

Avnur, R. and Hellerstein, J.M. 2000. Eddies: Continuously adaptive query processing. In Proc. ACM-SIGMOD International Conference on Management of Data, Dallas, May 2000.

Bayardo Jr., R.J. and Miranker, D.P. 1996. Processing queries for first-few answers. In Fifth Intl. Conf. Information and Knowledge Management, Rockville, MD.

Carey, M.J. and Kossmann, D. 1997. On saying “Enough Already!” in SQL. In Proc. ACM-SIGMOD International Conference on Management of Data, Tucson, May 1997.

Carey, M.J. and Kossmann, D. 1998. Reducing the braking distance of an SQL query engine. In Proc. 24th International Conference on Very Large Data Bases, New York City.

Chaudhuri, S. and Gravano, L. 1996. Optimizing queries over multimedia repositories. In Proc. ACM-SIGMOD International Conference on Management of Data, Montreal, June 1996.

Chaudhuri, S. and Gravano, L. 1999. Evaluating top-k selection queries. In Proc. International Conference on Very Large Data Bases, Edinburgh.

Chaudhuri, S. and Narasayya, V. 1998. AutoAdmin “What-If” index analysis utility. In Proc. ACM-SIGMOD International Conference on Management of Data, Seattle, June 1998.

Chaudhuri, S. and Shim, K. 1996. Optimization of queries with user-defined predicates. In Proc. 24th International Conference on Very Large Data Bases, Bombay (Mumbai), September 1996.

Cherniack, M. 1998. Building query optimizers with combinators. PhD Thesis, Brown University.

DeWitt, D.J., Katz, R.H., Olken, Frank, Shapiro, L.D., Stonebraker, R.M., and Wood, D. 1984. Implementation techniques for main memory database systems. In Proc. ACM-SIGMOD International Conference on Management of Data, Boston, June 1984.

Donjerkovic, D. and Ramakrishnan, R. 1999. Probabilistic optimization of Top N queries. In Proc. International Conference on Very Large Data Bases, Edinburgh.

Fagin, R. 1998. Fuzzy queries in multimedia database systems. In Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Seattle, June 1998.

Fayyad, U., Piatetsky-Shapiro, G., and Smyth, P. 1996. The kdd process for extracting useful knowledge from volumes of data. Communications of the ACM, 39(11).

Fushimi, S., Kitsuregawa, M., and Tanaka, H. 1986. An overview of the system software of a parallel relational database machine GRACE. In Proc. 24th International Conference on Very Large Data Bases, Kyoto, August 1986.

Gibbons, P.B. and Matias, Y. 1998. New sampling-based summary statistics for improving approximate query answers. In Proc. ACM-SIGMOD International Conference on Management of Data, Seattle.

Gibbons, P.B., Poosala, V., Acharya, S., Bartal, Y., Matias, Y., Muthukrishnan, S., Ramaswamy, S., and Suel, T. 1998. Aqua: System and techniques for approximate query answering. Technical Report, Bell Laboratories.

Graefe, G. 1993. Query evaluation techniques for large databases. ACM Computing Surveys, 25(2):73–170.

Gray, J. and Graefe, G. 1997. The five-minute rule ten years later, and other computer storage rules of thumb. SIGMOD Record, 26(4).

Haas, P.J. 1996. Hoeffding inequalities for join-selectivity estimation and online aggregation. IBM Research Report RJ 10040, IBM Almaden Research Center.

Haas, P.J. 1997. Large-sample and deterministic confidence intervals for online aggregation. In Proc. 9th International Conference on Scientific and Statistical Database Management, Olympia, WA, August 1997.

Haas, P.J. and Hellerstein, J.M. 1999. Ripple algorithms for online aggregation. In Proc. ACM-SIGMOD International Conference on Management of Data, Philadelphia, May 1999.

Haas, P.J., Naughton, J.F., Seshadri, S., and Swami, A.N. 1996. Selectivity and cost estimation for joins based on random sampling. Journal of Computer System Science, 52:550–569.

Harinarayan, V., Rajaraman, A., and Ullman, J.D. 1996. Implementing data cubes efficiently. In Proc. ACM-SIGMOD International Conference on Management of Data, Montreal, June 1996.

Hellerstein, J.M. 1997a. The case for online aggregation. Computer Science Technical Report CSD-97-958, University of California, Berkeley.

Hellerstein, J.M. 1997b. Online processing redux. IEEE Data Engineering Bulletin, 20(3).

Hellerstein, J.M. 1998a. Looking forward to interactive queries. Database Programming and Design, 11(8):28–33.

Hellerstein, J.M. 1998b. Optimization techniques for queries with expensive predicates. ACM Transactions on Database Systems, 23(2).

Hellerstein, J.M., Avnur, R., Chou, A., Hidber, C., Olston, C., Raman, V., and Roth, T. 1999. Interactive Data Analysis with CONTROL. IEEE Computer 32(9):51–59.

Hellerstein, J.M., Haas, P.J., and Wang, H.J. 1997. Online aggregation. In Proc. ACM-SIGMOD International Conference on Management of Data, Tucson, May 1997.

Hellerstein, J.M. and Naughton, J.F. 1996. Query execution techniques for caching expensive methods. In Proc. ACM-SIGMOD International Conference on Management of Data, Montreal, June 1996.

Hidber, C. 1997. Online association rule mining. In Proc. ACM-SIGMOD International Conference on Management of Data, Tucson, May 1997.

Hoeffding, W. 1963. Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association, 58.

Hou, W.C., Ozsoyoglu, G., and Taneja, B.K. 1988. Statistical estimators for relational algebra expressions. In Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Austin, March 1998.

Hou, W.C., Ozsoyoglu, G., and Taneja, B.K. 1989. Processing aggregate relational queries with hard time constraints. In Proc. ACM-SIGMOD International Conference on Management of Data, Portland, May–June 1989.

Hyperion Essbase OLAP Server, 1998. URL http://www.hyperion.com/essbaseolap.cfm.

Illustra Information Technologies, Inc. 1994. Illustra User's Guide, Illustra Server Release 2.1.

Informix Corp. 1998a. Sampling: The latest breakthrough in decision support technology. Informix White Paper 000-21681-70.

Informix Corp. 1998b. C-ISAM Version 7.24 for the UNIX Operating System.

Informix Corp. 1998c. Informix Dynamic Server with Universal Data Option 9.1x.

Knuth, D.E. 1973. The Art of Computer Programming: Vol. 3, Sorting and Searching. Addison-Wesley.

Leung, T.Y.C., Pirahesh, H., Seshadri, P., and Hellerstein, J.M. 1998. Query rewrite optimization rules in IBM DB/2 universal database. In Readings in Database Systems, 3rd ed., M. Stonebraker and J.M. Hellerstein (Eds.). San Francisco: Morgan-Kaufmann.

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

Livny, M., Ramakrishnan, R., Beyer, K.S., Chen, G., Donjerkovic, D., Lawande, S., and Myllymaki, J. 1997. DEVise: Integrated querying and visualization of large datasets. In Proc. ACM-SIGMOD International Conference on Management of Data, Tucson, May 1997.

Lynch, C. and Stonebraker, M. 1988. Extended user-defined indexing with application to textual databases. In Proc. 14th International Conference on Very Large Data Bases, Los Angeles, August–Septeber 1998.

Maier, D. and Stein, J. 1986. Indexing in an object-oriented DBMS. In Proc. 1st Workshop on Object-Oriented Database Systems, Asilomar, September 1986.

Morgenstein, J.P. 1980. Computer based management information systems embodying answer accuracy as a user parameter. PhD Thesis, U.C. Berkeley.

O'day, V. and Jeffries, R. 1993. Orienteering in an information landscape: How information seekers get from here to there. In INTERCHI.

Ohno, P. 1998. Visionary. Informix Magazine.

Olken, F. 1993. Random sampling from databases. PhD Thesis, University of California, Berkeley.

Papadopoulos, G. Chief Technology Officer. 1997. Sun Microsystems. Untitled talk. Berkeley NOW Retreat, July 1997.

Perlin, K. and Fox, D. 1993. Pad: An alternative approach to the computer interface. In Proc. ACM SIGGRAPH, Anaheim, pp. 57–64.

Pilot Software 1998. Announces release of PDSS 6.0. URL http://www.pilotsw.com/about/pressrel/pr72998.htm.

Raman, V., Chou, A., and Hellerstein, J.M. 1999a. Scalable spreadsheets for interactive data analysis. In DMKD Workshop.

Raman, V., Raman, B., and Hellerstein, J.M. 1999b. Online dynamic reordering for interactive data processing. In Proc. International Conference on Very Large Data Bases, Edinburgh.

Rao, J. and Ross, K.A. 1998. Reusing invariants: A new strategy for correlated queries. In Proc. ACM-SIGMOD International Conference on Management of Data, Seattle, June 1998.

Red Brick Systems, Inc. 1998. Red brick warehouse. URL http://www.redbrick.com/products/rbw/rbw.html.

Seshadri, P. and Swami, A. 1995. Generalized partial indexes. In Proc. 11th IEEE International Conference on Data Engineering, Taipei, March 1995.

Shneiderman, B. 1982. The future of interactive systems and the emergence of direct manipulation. Behavior and Information Technology, 1(3):237–256.

Shukla, A., Deshpande, P., and Naughton, J.F. 1998. Materialized view selection for multidimensional datasets. In Proc. 24th International Conference on Very Large Data Bases, New York City.

Silberschatz, A., Read, R.L., and Fussell, D.S. 1992. A multi-resolution relational data model. In Proc. 18th International Conference on Very Large Data Bases, Vancouver, August 1992.

QL 1998. Server 7.0 OLAP services. URL http://www.microsoft.com/backoffice/sql/70/whpprs/olapoverview.htm.

Stonebraker, M. 1989. The case for partial indexes. SIGMOD Record, 18(4):4–11.

Stonebraker, M. and Kemnitz, G. 1991. The POSTGRES Next-Generation database management system. Communications of the ACM, 34(10):78–92.

Tan, K., Goh, C.H., and Ooi, B.C. 1999. Online feedback for nested aggregate queries with multi-threading. In Prov. International Conference on Very Large Data Bases, Edinburgh.

Transaction Processing Council. TPC-D Rev. 1.2.3 Benchmark Specification. URL http://www.tpc.org/dspec.html.

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

Waldspurger, C.A. and Weihl, W.E. 1995. Lottery scheduling: Flexible proportional-share resource management. In First Symposium on Operating Systems Design and Implementation (OSDI).

Walter, T., Chief Technical Officer. 1998. NCR parallel systems. Complex queries. NSF Database Systems Industrial/Academic Workshop, October 1998.

Wilschut, A.N. and Apers, P.M.G. 1991. Dataflow query execution in a parallel main-memory environment. In Proc. First Intl. Conf. Parallel and Distributed Info. Sys. (PDIS), pages 68–77, Miami Beach, December 1991.

Winter, R. and Auerbach, K. 1998. The big time: 1998 winter VLDB survey. Database Programming and Design.

Zilberstein, S. and Russell, S.J. 1996. Optimal composition of real-time systems. Artificial Intelligence, 82(1/2):181–213.