Ripple Joins for Online Aggregation

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

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

Abstract

We present a new family of join algorithms, called ripple joins, for online processing of multi-table aggregation queries in a relational database management system (DBMS). Such queries arise naturally in interactive exploratory decision-support applications.

Traditional offline join algorithms are designed to minimize the time to completion of the query. In contrast, ripple joins are designed to minimize the time until an acceptably precise estimate of the query result is available, as measured by the length of a confidence interval. Ripple joins are adaptive, adjusting their behavior during processing in accordance with the statistical properties of the data. Ripple joins also permit the user to dynamically trade off the two key performance factors of online aggregation: the time between successive updates of the running aggregate, and the amount by which the confidence-interval length decreases at each update. We show how ripple joins can be implemented in an existing DBMS using iterators, and we give an overview of the methods used to compute confidence intervals and to adaptively optimize the ripple join "aspect-ratio" parameters. In experiments with an initial implementation of our algorithms in the POSTGRES DBMS, the time required to produce reasonably precise online estimates was up to two orders of magnitude smaller than the time required for the best offline join algorithms to produce exact answers.

1 Introduction

Current relational database management systems do not handle *ad hoc* decision-support queries well, even though such queries are important in applications. Many decision-support queries consist of a complex sequence of joins and selections over extremely large tables, followed by grouping of the result and computation of aggregates over the groups. Current systems process *ad hoc* queries in what amounts to *batch* mode: users are forced to wait for a long time without any feedback until a precise answer is returned.

Since large-scale aggregation queries typically are used to get a "big picture" of a data set, a more attractive approach is to perform *online aggregation*, in which progressively-refined running estimates of the final aggregate values are continuously displayed to the user. The estimated proximity of a running estimate to the final result is indicated by means of an associated confidence interval. An online aggregation system must be optimized to provide useful information quickly, rather than to minimize the time to query completion. This new performance goal requires fundamental changes to many traditional algorithms for query processing. In prior work [HHW97] we provided initial motivation, statistical techniques and algorithms for supporting online aggregation queries in a relational DBMS. In this paper we extend those results with a new family of join algorithms called *ripple joins*, which are designed to meet the performance needs of an online query processing system.

Ripple joins generalize traditional block nested-loops and hash joins and are non-blocking, thereby permitting the running estimates to be updated in a smooth and continuous fashion. The user can control the rate at which the updates occur; for a given updating rate the ripple join adaptively modifies its behavior based on the data in order to maximize the amount by which the confidence interval shrinks at each update.

Ripple joins appear to be among the first database algorithms to use statistical information about the data not just to estimate selectivities and processing costs, but to estimate the quality of the result currently being displayed to the user and to dynamically adjust algorithm behavior accordingly. We believe that such a synthesis of statistical estimation methods and query processing algorithms will be integral to the online decision support systems of the future.

SIGMOD '99 Philadelphia PA

2 Background

2.1 Online Aggregation

We illustrate online aggregation by means of an example. Consider the following query for determining the grade-point average of various types of honors students (`honors_code NOT NULL`) and non-honors students (`honors_code IS NULL`):

```
SELECT ONLINE student.honors_code,AVG(enroll.grade)
FROM enroll,student
WHERE enroll.sid = student.sid
GROUP BY student.honors_code;
```

A prototype of an online aggregation interface for this query is displayed in Figure 1. There is a row corresponding to each student group, that is, to each distinct value of `honors_code` that appears in the table. The user does not need to specify the groups in advance—they are automatically detected by the system. For each group, the running estimate of the final query result is simply the average of all of the grades for the group found so far. These running

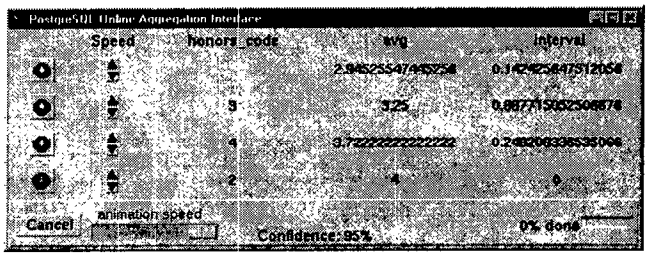

Figure 1: An online aggregation interface.

estimates (after less than 1% of the records in the Cartesian product have been scanned) are displayed in the column entitled **avg**. The "stop-sign" buttons can be used to pause the query processing for one or more groups, while allowing query processing to continue for the remaining groups. The arrow buttons in the column entitled **Speed** permit control of the relative rates at which the running averages for the different groups are updated. Implementation of the **Speed** control is described further in [HHW97, RRH99].

The rows in the table are processed in random order. Such processing is the same as simply scanning the table when it is clustered in random order on disk. Such random clustering (with respect to the attributes involved in the aggregation query) can be verified *a priori* by statistical testing; if the initial clustering is unsatisfactory then the rows can be randomly permuted prior to query processing. Of course, one cannot always cluster a table in random order, e.g., if one desires it to be clustered on a particular column. In such cases, a secondary random index (an index on the `random()` function) can be constructed to support a random ordering; see [HAR99] for a further discussion of issues in physical database design for online aggregation. Alternatively, it may be desirable to either sample during query processing using techniques as in [Olk93] or to materialize/cache a small random sample of each base relation during an initialization step and then subsequently scan the sample base relations during online processing. In this paper we assume that any one of these random-order access methods is available.

Since tuples are processed in random order, we can bring to bear the tools of statistical estimation theory. In particular, we can indicate the proximity of a running estimate to the final query result by means of an associated *running confidence interval*. For example, the running average of about 2.95 for the first group in Figure 1 is within ± 0.15 of the final average with probability 95%. Using the running confidence intervals as a guide, the user can abort the current query as soon as the displayed answer is sufficiently precise. The user can then proceed to the next query and continue the process of data exploration [OJ93]. In general, online algorithms are designed to support this mode of interaction between the user and the DBMS.

2.2 Join Algorithms for Online Processing

Our goal is to provide join algorithms that will support online processing for multi-table queries of the form

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

where $K \geq 2$, *op* is an aggregation operator such as `COUNT`, `SUM`, `AVG`, `VARIANCE`, or `STDEV`, *expression* is an arithmetic expression involving the attributes of the base relations $R_1, R_2, \ldots, R_K$, and *predicate* is a conjunction of join and selection predicates involving these attributes.

In general, there is a tradeoff between the rate at which the running confidence intervals are updated and the degree to which the interval length decreases at each update; this tradeoff gives rise to a spectrum of possible join algorithms. Classical offline join algorithms can be viewed as lying at one end of this spectrum: after processing all of the data, the "running" confidence interval is updated exactly once, at which time the length of the interval decreases to zero. Algorithms that block during processing, such as hash join and sort-merge join, fall into this category. The performance of the classical algorithms often is unacceptable in the online setting, since the time until the "update" occurs can be very long.

Prior to the current work, the only classical algorithm lying elsewhere along the spectrum was the nested-loops join as proposed for use in the setting of online aggregation in [HHW97]. This algorithm (in its simplest form) works roughly as follows for a two-table aggregation query over relations R and S with $|R| < |S|$. At each sampling step, a random tuple s is retrieved from S[1]. Then R is scanned; for each tuple r that joins with s, an argument of the aggregation function is produced from r and s. At the end of the sampling step the running estimate and confidence interval are updated according to formulas as given in [HHW97, Haa97].

Using nested-loops join in an online fashion is certainly more attractive than waiting until the nested-loops join has completed before returning an answer to the user. The absolute performance of the online nested-loops join is frequently unacceptable, however, for two reasons. First, a complete scan of R is required at each sampling step; if R is of nontrivial size (as is often the case for decision-support queries), then the amount of time between successive updates to the running estimate and confidence interval can be excessive. Second, depending upon the statistical properties of the data, the length of the confidence interval may not decrease sufficiently at each sampling step. As an extreme example of this latter phenomenon, suppose that the join of R and S is in fact the Cartesian product $R \times S$, and the input to the aggregation function is relatively insensitive to the values in the columns of R, e.g., as in the query `SELECT AVG(S.a + R.b/10000000) FROM R, S`. Also suppose that we have retrieved a random tuple $s \in S$, retrieved the first tuple $r \in R$, and produced an argument of the aggregation function from r and s. Then the rest of the scan of R yields essentially no new information about the value of the aggregation query, statistically speaking, even though a large I/O cost is incurred by performing the scan. While this is an artificial and extreme example, in Section 6 we will see quite reasonable scenarios where nested-loops join does a poor job at shrinking confidence intervals.

Ripple joins are designed to avoid complete relation scans and maximize the flow of statistical information during join processing. The user can explicitly trade off the time between successive updates of the running estimate with the amount by which the confidence-interval length decreases at each update. This tradeoff is effected using the **animation speed** slider shown in Figure 1 and discussed in detail in

[1] For traditional batch processing of this nested-loop join, R would be chosen as the outer relation since it is smaller. For online processing the opposite choice is preferable, since the running estimate is updated after each scan of the inner relation.

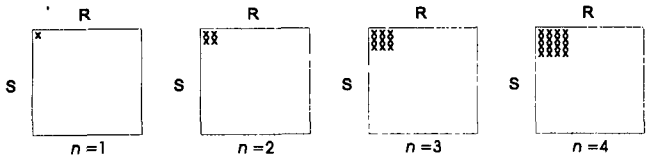

Figure 2: The elements of $R \times S$ that have been seen after n sampling steps of a "square" ripple join.

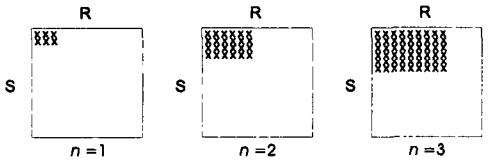

Figure 3: The elements of $R \times S$ that have been seen after n sampling steps of a "rectangular" ripple join ($\beta_1 = 3$, $\beta_2 = 2$.)

Section 5.3. By adjusting the animation setting, we obtain a family of join algorithms that covers the entire spectrum of possibilities.

2.3 Related Work

The idea of sampling from base relations in order to quickly estimate the answer to a COUNT query goes back to the work of Hou, et al. [HOT88, HOT89]; see [HHW97] for further references. Techniques that are applicable to other types of aggregation queries follow from results in [Olk93] and [ODT+91]; the "acceptance/rejection" sampling techniques described in these references do not appear directly applicable to online aggregation.

Algorithmically, ripple join generalizes and extends prior work on pipelining join algorithms. The simplest members of this class are the classical naive-, block-, and index-nested loops joins. Ripple join also bears a resemblance to the *semi-naive evaluation* technique used for recursive query processing (see, e.g., [RSS94]): both algorithms handle newly-arrived tuples in one operand by joining them with all previously-seen tuples of the other operand. Another similar idea is used in the more recent pipelining hash join of [WA91], which was proposed for use in online aggregation previously [HHW97]. None of the prior work considers either the relative rates of the two operands, or the connection to confidence-interval estimation—these issues are critical in the setting of online aggregation.

3 Overview of Ripple Join

In the simplest version of the two-table ripple join, 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 2. 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 retrieval order; this order is assumed to be random.

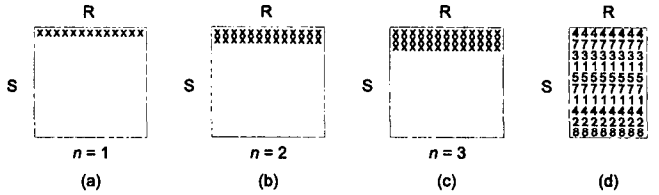

Figure 4: The elements of $R \times S$ that have been seen after n sampling steps of an online nested-loops join ($n = 1, 2, 3$) and a worst-case scenario for online nested-loops join.

The "square" version of the ripple join described above draws samples from R and S at the same rate. As discussed in Section 5.3 below, it is often necessary to sample one relation (the "more variable" one) at a higher rate than another in order to provide the shortest possible confidence intervals for a given animation speed. This requirement leads to the general "rectangular" version of the ripple join[2] depicted in Figure 3. 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 3 corresponds to the special case in which $K = 2$, $\beta_1 = 3$, and $\beta_2 = 2$.) Note the tradeoff between the sampling rate and the confidence-interval length. For example, when $\beta_1 = 1$ 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 is longer; on the other hand, after each sampling step the confidence interval typically is shorter when $\beta_1 = 1$ and $\beta_2 = 2$.

The ripple join reduces to an online nested-loops join when the aspect ratio is defined by $\beta_K = 1$ and $\beta_{K-1} = |R_{K-1}|, \ldots, \beta_1 = |R_1|$; see Figures 4(a)–4(c) for $K = 2$. In Figure 4(d), each point $(r, s) \in R \times S$ is represented by the argument of the aggregation function produced from r and s; the values displayed in this figure correspond to the most extreme form of the problematical case discussed in the Section 2.2—here the input to the aggregation function is completely insensitive to the attribute values in R. In choosing an online nested-loops join, a query optimizer would take S to be the outer relation in this case, since $|S| > |R|$ in Figure 4(d). If R is at all large, this decision is incorrect for the purposes of online aggregation; the optimizer's mistake is in not explicitly taking the statistical characteristics of the data into consideration. We will see how ripple join avoids this error by adapting dynamically to the data's statistical properties.

4 Ripple Join Algorithms

Ripple join can be viewed as a generalization of nested-loops join in which the traditional roles of "inner" and "outer" relation are continually interchanged during processing. In the simple pseudocode for a square two-table ripple join displayed in Figure 5, each full outermost loop corresponds to a sampling step. Within the nth sampling step, the cursor into S is first fixed at the value max $= n$ while the cursor into R loops from 1 to $n-1$. Then, when the cursor into R reaches the value n, the cursor into S loops from 1 to n. Unlike

[2]The name "ripple join" has two sources. One is shown in the pictures in Figures 2 and 3—the algorithm sweeps out the plane like ripples in a pond. The other source is the rectangular version of the algorithm, which produces "Rectangles of Increasing Perimeter Length".

```
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 5: A simple square ripple join. The tuples within each relation are referred to in array notation.

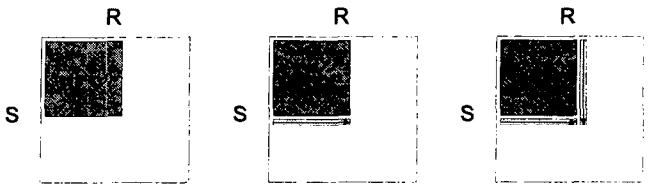

Figure 6: Three phases of a sampling step, square ripple join.

nested-loops join, square ripple join is essentially symmetric: during a sampling step, each input relation participates in a tight "innermost" loop from position 1 to either position n or position $n-1$. The manner in which a sampling step sweeps out $R \times S$ is depicted graphically in Figure 6.

In this simple form, ripple join is quite easy to express. We have, however, ignored a number of issues, which we proceed to address in the remainder of this section. First, since most DBMS's use a "pull" or *iterator* model for relational operators [Gra93], we show how to represent ripple joins in iterator form, starting with the simplest binary square ripple join. Then we show how to augment this simple iterator to handle non-unitary aspect ratios and permit incorporation into a pipeline of multiple ripple joins. Finally, we describe variants of the basic algorithm that exploit blocking, indexing, and hashing techniques to enhance performance.

4.1 A Square Binary Ripple Join Iterator

An iterator-based DBMS invokes an iterator's `next()` method each time an output tuple is needed. A ripple join iterator must maintain enough internal state variables to allow production of tuples in the same sequence as would be produced by the algorithm of Figure 5. A simplified ripple join iterator "object class" is shown in Figure 7, in a C++ or Java-style pseudocode. The iterator needs to store the next position to be fetched from each of its inputs R and S (`R.pos, S.pos`), along with the current sampling step being produced (`curstep`), and the relation currently acting as the "inner" (`currel`). The code in Figure 7 does not handle the case in which the last tuple of R or S has been retrieved, and it assumes that the query plan consists of a single square ripple join; these assumptions will be relaxed below.

A slight asymmetry arises in this iterator: only the cursor into S loops all the way to `curstep`, and `curstep` is advanced after completing a loop through S but not after completing a loop through R. This asymmetry corresponds to the asymmetry in Figure 5, in which only the cursor into S loops to `max`, and `max` is advanced only after completing a loop through S. The same asymmetry also appears in the way that the tuple "layers" are mitred together in the lower right corner in Figure 6. When the situation is as depicted in Figure 6, we call S the "starter" relation since each sampling step starts with the retrieval of a new tuple from S.

```
class simple_RIPL {
  int curstep;     // sampling step
  relation R, S;   // operands
  relation currel; // the current inner
  bool ilooping;   // in midst of inner loop?
  init() {
    R.pos = 1; // cursor positions in R and S
    S.pos = 0;
    curstep = 1;
    currel = S;
    ilooping = true;
  }
  next() {
    do { // loop until return() is called
      if (ilooping) { // scanning side of a rectangle
        while(currel.pos < curstep) {
          if (currel.pos < curstep-1 || currel==S) {
            currel.pos++;
            if (predicate(R[R.pos],S[S.pos]))
                return(R[R.pos], S[S.pos])
            } }
        ilooping = false; // finished a side
      }
      else { // done with one side of a rectangle
        if (currel == S)
           curstep++; // finished a step
        currel.pos++; // sets currel to new curstep
        toggle(currel);
        currel.pos = 0;
        ilooping = true;
      } } } }
```

Figure 7: A simple iterator for square ripple join.

4.2 An Enhanced Ripple Join Iterator

For clarity of exposition, the previous section ignored complications arising from non-unitary aspect ratios and integration of a ripple join iterator into a query plan tree. In this section we address these remaining issues. Full pseudocode for the resulting ripple join iterator is presented in [HH98, Appendix A].

4.2.1 Non-Unitary Aspect Ratios

As mentioned previously, it is often beneficial to retrieve tuples from the two inputs of a ripple join at uneven rates, resulting in "ripples" of non-unit aspect ratio. This requires three details to be handled by the iterator. First, the aspect ratio must be stored as a local variable `beta` for each relation. Second, the iterator loops through R until it reaches a limit of `curstep*R.beta-1`, and loops through S until it reaches a limit of `curstep*S.beta`.

The third detail requires some care: `R.beta` and `S.beta` may not equal 1, and may not be equal to each other, so simply "wrapping" the entire old rectangle with a fixed number of new layers will not expand the sides of the next ripple by `R.beta` and `S.beta` respectively. In a single sampling step we must join `S.beta` "new" (previously-unseen) S tuples with all "old" (previously-seen) R tuples, join `R.beta` new R tuples with all old S tuples, and join all new R and S tuples. To do this, we enhance the iterator so that the first time it sees a tuple from a given relation, it considers it to be a "new" tuple, and combines it with all tuples seen so far from the previous relation. The resulting traversal of $R \times S$ is illustrated in Figure 8.

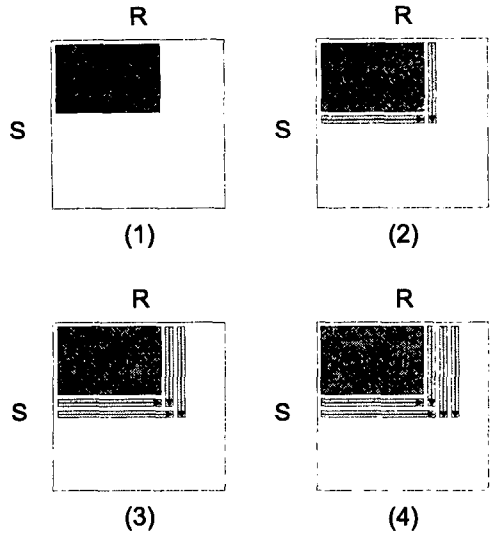

Figure 8: Four phases of a sampling step for a rectangular ripple join with $\beta_1 = 3$ and $\beta_2 = 2$.

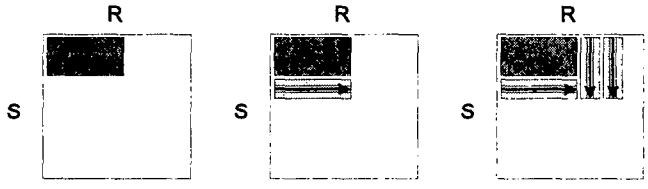

Figure 9: Three phases of a sampling step for a rectangular block ripple join with $\beta_1 = 2$, $\beta_2 = 1$, and block size = 3 tuples.

4.2.2 Pipelining Multiple Ripple Joins

The ripple join algorithm requires each of its input iterators to be restartable, and to deliver the same set of tuples each time it is restarted; beyond that it has no special prerequisites. Multiple binary ripple joins can therefore be pipelined in a join tree, or even intermingled with other query processing operators, including special online iterators like index stride [HHW97] and online reordering [RRH99].

Although in principle one could combine ripple joins with other join techniques, the typical K-table query plan in an online scenario will consist of a left-deep tree of binary ripple joins, which in combination are supposed to correctly sweep out a sequence of K-dimensional hyper-rectangles with the appropriate aspect ratios. To accomplish this, the operating parameters for each iterator in the tree must take into account the position of the iterator in the tree. A three-table cubic ripple join, for example, cannot simply be treated as a pipeline of two (binary) square ripple joins, each operating in isolation. In order to get a full n-dimensional "wrapper" of the hyper-rectangle from the previous step, the following modifications must be made:

- **Aspect ratios must be set and maintained correctly.** For an iterator with two query subtrees, where subtree R contains relations $R_1, \ldots, R_j$ and subtree S contains relations $S_1, \ldots, S_k$, the aspect-ratio parameters must be set to the values $\texttt{R.beta} = \beta_{R_1}\beta_{R_2}\cdots\beta_{R_j}$ and $\texttt{S.beta} = \beta_{S_1}\beta_{S_2}\cdots\beta_{S_k}$.
- **The appropriate number of retrievals must be made from each operand.** In particular, an iterator should retrieve $\texttt{R.beta} \times n^j$ tuples from R in step n, where j is the number of leaves in subtree R.
- **Only one relation in the plan can be the starter.** At the beginning of processing, the right (i.e., the base) relation of the highest join node in the plan tree is designated as the starter relation. When the cursor of a non-starter relation R exceeds $n\beta_R$ for the nth sampling step, it returns a signal as if it had reached end-of-file. When the cursor of the starter relation, say S, exceeds $n\beta_S$, it increments the sampling step to $n+1$.

These three modifications ensure that ever-larger hyper-rectangles are swept out correctly.

4.3 Ripple Join Variants

It is well known that nested-loops join can be improved by *blocking* I/Os from the outer relation. The idea is to read this relation not merely a tuple at a time or even a disk page at a time, but rather in large "blocks" of pages. A *block ripple join* can be derived along the same lines. When a new block of one relation (say R) is read from disk, each tuple in that block is compared with all old tuples of the other relation, S. Then the block of R is evicted from memory and a new block of pages from S is read in, followed by a scan of the old tuples of R. The graphical representation of the way in which block ripple join sweeps out $R \times S$ is similar to that of standard ripple join in Figure 8, but with "thick" arrows consisting of multiple tuples at once; see Figure 9. Blocking amortizes the cost of rescanning one relation (e.g. S) across multiple tuples of the other (R), resulting in an I/O savings factor proportional to the block size.

The performance of ripple join also can be improved by the use of indexes. When there are two input relations R and S and there is an index on the join attributes of R, the *index ripple join* uses the index to identify the tuples in R that join with a given random tuple $s \in S$ selected during a sampling step. The relevant tuples from R can then be retrieved using fewer I/O's than would be required by a full scan of R as in nested-loops join. Note that the roles of outer and inner do not alternate in an index ripple join, and there is no choice of aspect ratio—each sampling step corresponds to a complete probe of the index on R, which sweeps out an entire row of $S \times R$. Thus while naive and block ripple join generalize their nested-loops counterparts, the index-enhanced ripple join is identical to an index-enhanced nested-loops join.

Finally, it is natural to consider a *hash ripple join* variant that can be used for equijoin queries. For such queries, use of hashing can drastically reduce I/O costs by avoiding the inefficient reading and re-reading of each tuple many times from disk that occurs during a simple ripple join. (This re-reading problem is worse even than for nested-loops join!) The basic idea is as follows. When a new tuple is fetched from one relation (say, R) in a ripple join, it must be combined with all old tuples from the other relation (S). Only some of these combinations will satisfy the join predicates. If the old tuples of S are kept in memory, and hashed on the join column, it is then possible to find the old matches for the new tuple very efficiently. Since ripple join is symmetric, an analogous situation arises with new tuples from S and old tuples from R. Thus, it is beneficial to materialize two hash tables in memory—one for R and one for S. Each contains the tuples seen so far. When a new tuple of R (S)

is fetched from disk, it is joined with all matches in the hash table for S (R), then inserted into the hash table for R (S). In the case of a square aspect ratio, this scheme reduces to the pipelining hash join of [WA91]. The hashing scheme breaks down, of course, when the hash tables no longer fit in memory. At that point, the hash ripple algorithm can gracefully fall back to block ripple join[3]. This memory-overflow scenario should not cause much concern in practice—very tight confidence intervals often can be achieved long before memory is filled (see, e.g., Section 6).

5 Statistical Considerations for Algorithm Performance

The performance goal of ripple join is to provide efficient, accurate, interactive estimation. It should deliver join results in such a way that estimates of the aggregates are updated regularly and the corresponding confidence intervals shrink rapidly. Performance in this online regime depends critically on the statistical methods used to compute confidence intervals and on the way in which these methods interact with retrieval of tuples from the join's input relations.

To highlight the key issues, we give a brief overview of estimators for some common multi-table aggregation queries. We then present confidence-interval formulas that characterize the precision of these estimators. To keep the presentation simple, we focus on the simplest types of aggregation queries. Complete details of currently available formulas and their derivations are given in [HH98]. We conclude the section by discussing our approach to the dynamic optimization of aspect-ratio parameters.

5.1 Estimators for SUM, COUNT and AVG

Our running estimators for standard SQL aggregates **SUM**, **COUNT** and **AVG** are little more than running sums, counts and averages, scaled as appropriate. Specifically, consider a simple two-table query of the form

```
SELECT op(expression) FROM R, S
WHERE predicate;
```

where *op* is one of **SUM**, **COUNT** or **AVG**. (All of our formulas extend naturally to the case of multiple tables. When *op* is equal to **COUNT**, we assume that *expression* reduces to the SQL "*" identifier.) The predicate in the query can in general consist of conjunctions and/or disjunctions of boolean expressions involving multiple attributes from both R and S; we make no simplifying assumptions about the joint distributions of the attributes in either of these relations. At the end of the nth sampling step, a natural estimator for **SUM**(*expression*) is

$$\frac{|R| \cdot |S|}{|R_n| \cdot |S_n|} \sum_{(r,s) \in R_n \times S_n} expression_p(r,s), \qquad (5.1)$$

where R_n and S_n are the sets of tuples that have been read from R and S by the end of the nth sampling step, and $expression_p(r,s)$ equals $expression(r,s)$ if (r,s) satisfies the **WHERE** clause, and 0 otherwise. This estimator is simply the running sum scaled up by a ratio of the total input size to the current input size. The estimator is *unbiased*: if the sampling and estimation process were repeated over and over, then the estimator would be equal on average to the true query result. The estimator is also *consistent* in that it converges to the correct result as the number of sampling steps increases. Similarly, an unbiased and consistent estimator for **COUNT**(*) is given by (5.1), but with $expression_p(r,s)$ replaced by $one_p(r,s)$, where $one_p(r,s)$ equals 1 if (r,s) satisfies the **WHERE** clause, and equals 0 otherwise. Finally, an estimator for **AVG**(*expression*) is found by dividing the sum estimator by the count estimator. This ratio—after factoring—is simply the running average. Like all ratio estimators, the estimator for **AVG**(*expression*) is biased, but the bias converges to 0 as the number of sampling steps increases. Moreover, the estimator is consistent. Although each of the **SUM**, **COUNT**, and **AVG** estimators is a running aggregate (suitably scaled), running estimators of more complicated aggregates need not be exactly of this form; see, for example, the discussion of the **VARIANCE** and **STDEV** aggregates in [Haa97], in which the running aggregate is multiplied by a correction factor to remove bias.

5.2 Confidence Intervals

We need to develop tight confidence intervals in order to characterize the accuracy of the estimators in Section 5.1; this is a nontrivial task. In this section we give an overview of our methodology for obtaining "large-sample" confidence intervals based on central limit theorems (CLT's)[4].

5.2.1 The CLT and Confidence Intervals

To motivate our approach, we briefly review the classical CLT for averages of independent and identically distributed (iid) random variables and the way in which this CLT is used to develop confidence intervals for estimators of the population average. Consider an arbitrary but fixed set of distinct numbers $A = \{v_1, v_2, \ldots, v_{|A|}\}$. (The assumption that the numbers are distinct is convenient but not essential.) Let μ and σ^2 be the average and variance of these values:

$$\mu = \frac{1}{|A|}\sum_{i=1}^{|A|} v_i \qquad \text{and} \qquad \sigma^2 = \frac{1}{|A|}\sum_{i=1}^{|A|}(v_i - \mu)^2. \qquad (5.2)$$

Suppose that we wish to estimate the average μ, and that a sample $B = \{X_1, X_2, \ldots, X_n\}$ of size $n > 1$ is drawn randomly and uniformly (with replacement) from A. Under this sampling scheme, each X_i is equal to v_1 with probability $1/|A|$, to v_2 with probability $1/|A|$, and so forth, and knowledge of the value of X_i yields no information about the value of X_j for $j \neq i$. Thus the random observations $X_1, X_2, \ldots, X_n$ are iid.

The natural estimator of μ is the average of the n values in the sample, denoted by $\widehat{\mu}_n$. Of course, $\widehat{\mu}_n$ is a random quantity since the sample is random. The CLT for iid random variables asserts that *for large n the random variable $\widehat{\mu}_n$ has approximately a normal distribution with mean μ and variance σ^2/n.* "Large" can mean as few as 20 to 40 samples when σ^2 is small relative to μ. The normal approximation is accurate even when samples are obtained without replacement, as long as $n \ll |A|$.

To obtain a confidence interval for μ, we consider a "standardized" random variable Z that is obtained by shifting

[3] It is tempting in this context to utilize the ideas of hybrid hash join [DKO+84] as extended in [HN96] and spool tuples to disk after memory fills. Unfortunately, the resulting statistical properties of the running estimator are unsuitable for confidence-interval estimation; see [HH98]. Such a "symmetric hybrid hash" join algorithm could, however, be used for traditional query processing.

[4] See [Haa97, HHW97] for a discussion of other possible types of confidence intervals, as well as methods for dealing with **GROUP BY** and **DISTINCT** clauses.

and scaling $\widehat{\mu}_n$: $Z = (\widehat{\mu}_n - \mu)/(\sigma/\sqrt{n})$. It follows from an elementary property of the normal distribution that Z has approximately a standard (mean 0 and variance 1) normal distribution. For $p \in (0,1)$, denote by z_p the unique number such that the area under the standard normal curve between $-z_p$ and z_p is equal to p; see [AS72, Sec. 26] for a discussion of methods for computing z_p. It follows from the foregoing discussion that $P\{-z_p \le Z \le z_p\} \approx p$, and straightforward calculations then show that $P\{\widehat{\mu}_n - \epsilon_n \le \mu \le \widehat{\mu}_n + \epsilon_n\} \approx p$, where $\epsilon_n = z_p\sigma/\sqrt{n}$. Thus the true value μ lies within $\pm\epsilon_n$ of the estimator $\widehat{\mu}_n$ with probability approximately p. Equivalently, the random interval $I_n = [\widehat{\mu}_n - \epsilon_n, \widehat{\mu}_n + \epsilon_n]$ contains μ with probability $\approx p$ and hence is an approximate $100p\%$ confidence interval for μ. Since σ, like μ, is unknown, we replace σ with an estimator $\widehat{\sigma}_n$ in the final formula for ϵ_n. A natural choice for $\widehat{\sigma}_n$ is the standard deviation of the n numbers in the sample; $\widehat{\sigma}_n$ is close to σ when n is large, and the confidence interval remains valid.

5.2.2 Confidence Intervals for the Aggregates

In this section we derive confidence intervals for the SUM, COUNT, and AVG estimators of Section 5.1. One might hope to do this by directly applying the results in Section 5.2.1. Indeed, each of the SUM and COUNT aggregates is actually an average like μ in (5.2), but in disguise: SUM is the average value of $|R| \cdot |S| \cdot \mathit{expression}_p(r,s)$ over $(r,s) \in R \times S$ and COUNT is the average value of $|R| \cdot |S| \cdot \mathit{one}_p(r,s)$.

Unfortunately, several complicating factors preclude application of the classical CLT for averages of iid random variables. One obvious difficulty is that the AVG aggregate is not a simple average but rather a ratio of two averages. A perhaps more subtle but even more serious complication faces all three estimators: the random observations $\{\mathit{expression}_p(r,s)\colon (r,s) \in R_n \times S_n\}$ are identically distributed but *not* independent, and similarly for the random observations $\{\mathit{one}_p(r,s)\colon (r,s) \in R_n \times S_n\}$. For example, suppose that $r \in R_n$ and $s, s' \in S_n$. Then $\mathit{expression}_p(r,s)$ and $\mathit{expression}_p(r,s')$ are in general dependent, because both observations involve the same tuple r. Thus we need an extension of the classical CLT to the case of "cross-product averages" and (in order to handle AVG queries) ratios of such averages.

The desired extensions of the CLT can be derived using arguments very similar to those in [HNSS96, Haa97]. The basic idea for an individual cross-product average is to use induction on the number of input relations together with results from the theory of "convergence in distribution" [Bil86]; ratios of cross-product averages are handled using a "delta-method" argument as in [Bil86]. The new CLT's assert that after a sufficiently large number of sampling steps, the SUM, COUNT, and AVG aggregate estimators of Section 5.1 are each approximately distributed according to a normal distribution with mean μ equal to the final query result and variance σ^2/n, where the formula for the variance constant σ^2 depends on the type of aggregate. Given such results, we can then proceed exactly as in Section 5.2.1 and obtain a $100p\%$ confidence interval for the running estimate after n sampling steps as $I_n = [\widehat{\mu}_n - \epsilon_n, \widehat{\mu}_n + \epsilon_n]$. Here $\widehat{\mu}_n$ is the running estimate and

$$\epsilon_n = \frac{z_p \widehat{\sigma}_n}{\sqrt{n}}, \tag{5.3}$$

where $\widehat{\sigma}_n^2$ is a consistent estimator of σ^2. This final half-width ϵ_n of the confidence interval is precisely the quantity displayed in the interval column of the interface in Figure 1.

In the remainder of this section we describe the specific form of the variance constant σ^2 and its estimator $\widehat{\sigma}_n^2$ in the context of SUM, COUNT, and AVG queries. For simplicity we focus primarily on two-way joins; see [HH98] for a detailed discussion of K-way joins.

SUM and COUNT Queries

First consider a SUM query. For $r \in R$, let $\mu(r;R)$ be the average of $|R| \cdot |S| \cdot \mathit{expression}_p(r,s)$ over all $s \in S$. It is not hard to see that the average of $\mu(r;R)$ over $r \in R$ is simply the final query result μ. Let $\sigma^2(R)$ be the variance of the numbers $\{\mu(r;R)\colon r \in R\}$: $\sigma^2(R) = (1/|R|)\sum_{r \in R}(\mu(r;R) - \mu)^2$. Similarly define $\sigma^2(S)$ for relation S. Suppose that at each sampling step of the ripple join we retrieve β_R blocks of tuples from R and β_S blocks of tuples from S, where there are α tuples per block. Then the variance constant σ^2 is given by $\sigma^2 = \sigma^2(R)/(\alpha\beta_R) + \sigma^2(S)/(\alpha\beta_S)$.

As in the classical iid case, the parameter σ^2 is unknown and must be estimated from the tuples seen so far. A natural estimator $\widehat{\sigma}_n^2(R)$ of $\sigma^2(R)$ after n sampling steps is the variance of the numbers $\{\widehat{\mu}_n(r;R)\colon r \in R_n\}$, where each $\widehat{\mu}_n(r;R)$ estimates $\mu(r;R)$ and is simply the average of $|R| \cdot |S| \cdot \mathit{expression}_p(r,s)$ over all $s \in S_n$ (that is, over all tuples from S seen so far). We can similarly define an estimator $\widehat{\sigma}_n^2(S)$ of $\sigma^2(S)$ and estimate σ^2 by $\widehat{\sigma}_n^2 = \widehat{\sigma}_n^2(R)/(\alpha\beta_R) + \widehat{\sigma}_n^2(S)/(\alpha\beta_S)$. In the case of a COUNT query, the formulas for σ^2 and $\widehat{\sigma}_n^2$ are almost identical to those for a SUM query, except that $\mathit{one}_p(r,s)$ plays the role of $\mathit{expression}_p(r,s)$.

AVG Queries

Because confidence intervals for an AVG query are based on a CLT for ratios of cross-product averages, the formulas for σ^2 and $\widehat{\sigma}_n^2$ are correspondingly more complicated than for a SUM or COUNT query. Recall that the AVG estimator can be expressed as a SUM estimator divided by a COUNT estimator. Denote by σ_s^2 and σ_c^2 the variance constants for these two estimators, defined as above. Also let μ_c and μ_s be the value of the COUNT and SUM aggregates based on all of the tuples in R and S. For $r \in R$, define $\mu(r;R)$ as in the case of a SUM query and define $\mu'(r;R)$ as the average of $|R| \cdot |S| \cdot \mathit{one}_p(r,s)$ over all $s \in S$. Next define $\rho(R)$ to be the covariance[5] of the pairs $\{(\mu(r;R), \mu'(r;R))\colon r \in R\}$. Similarly define $\mu'(s;S)$ and $\rho(S)$, and set $\rho = \rho(R)/(\alpha\beta_R) + \rho(S)/(\alpha\beta_S)$. Then the variance constant σ^2 is given by $\sigma^2 = (\sigma_s^2 - 2\mu\rho + \mu^2\sigma_c^2)/\mu_c^2$, where $\mu = \mu_s/\mu_c$ is the final AVG query result. Each of the parameters ρ, μ, σ_s^2, σ_c^2, and μ_c in the formula for σ^2 is computed from all of the tuples in R and S. If instead we compute each parameter from the tuples in R_n and S_n, we obtain natural estimators $\widehat{\rho}_n$, $\widehat{\mu}_n$, $\widehat{\sigma}_{s,n}^2$, $\widehat{\sigma}_{c,n}^2$, and $\widehat{\mu}_{c,n}$ of the parameters. Substituting these estimators into the formula for σ^2 leads to a consistent estimator $\widehat{\sigma}_n^2$.

General Aggregation Queries

For aggregation queries with $K \ge 2$ input relations R_1, $R_2, \ldots, R_K$ and corresponding aspect ratios $\beta_1, \beta_2, \ldots, \beta_K$, the computations are almost the same as the case of two input relations. Quantities such as $\mu(r;R_k)$, for example, are computed by fixing tuple $r \in R_k$ and averaging $\mathit{expression}_p$ over the cross-product of the remaining input relations; see

[5] Recall that for pairs $(x_1,y_1),(x_2,y_2),\ldots,(x_k,y_k)$, the covariance is defined as $(1/k)\sum_{i=1}^{k}(x_i - \overline{x})(y_i - \overline{y})$, where $\overline{x}$ and $\overline{y}$ are the averages of the x_i's and y_i's.)

[HH98] for details. With respect to choosing the aspect ratios (as discussed in the next section), the most important overall observation is that, for **SUM**, **COUNT**, **AVG**, **VARIANCE**, and **STDEV** queries, the variance constant σ^2 can be written in the form

$$\sigma^2 = \sum_{k=1}^{K} \frac{d(k)}{\alpha\beta_k}. \tag{5.4}$$

In the above representation, each $d(k)$ is a constant that is computed from all of the tuples in the input relations according to a formula that depends upon the type of aggregate. Moreover, there always exists a consistent estimator $\widehat{d}_n(k)$ of each $d(k)$, which is obtained by applying the formula for $d(k)$ to the samples from the input relations rather than to the relations themselves.

5.3 Ripple Optimization: Choosing Aspect Ratios

For any aggregation query, the two key goals of ripple join are (1) to maximize the rate of updates to the estimates, and (2) to maximize the shrinkage of the confidence intervals at each update. These goals typically conflict: increasing the updating speed decreases the shrinkage rate for the confidence intervals, since rapid updates allow for only a few samples per update. To handle this tradeoff, we allow the user to set a lower bound on the updating speed; this lower bound is inversely proportional to the maximum time that a user is willing to look at a frozen display. In our example interface of Figure 1, the bound is controlled via the **animation speed** slider. Given a specified animation-speed setting, we try to minimize the length of the running confidence intervals. This is done by carefully selecting values of the aspect-ratio parameters[6] and, in the case of block ripple join, the blocking factor. In the following we consider the case of K-way block ripple and hash ripple joins with a single aggregate, and then briefly discuss the case of multiple aggregates.

5.3.1 Block Ripple Joins

Consider a block ripple join with blocking factor α and aspect-ratio parameters $\beta_1, \beta_2, \ldots, \beta_K$. It can be shown [HH98] that the cumulative I/O cost for n sampling steps of a block ripple join is proportional to[7] $\beta_1\beta_2\cdots\beta_K\alpha^{K-1}n^K + o(n^K)$. Roughly speaking, the quantity $\beta_1\beta_2\cdots\beta_K\alpha^{K-1}$ determines the rate at which the confidence-interval length is updated; the smaller the β_k's, the faster the updating rate. At one extreme, the animation speed is maximized when $\beta_1 = \beta_2 = \cdots = \beta_K = \alpha = 1$; that is, when we have a non-blocked square ripple join. Conversely, it follows from (5.4) that the larger the β_k's and α, the smaller the confidence-interval length after each update. So at the other extreme, the confidence-interval length is minimized when $\alpha\beta_k$ is equal to the cardinality of the kth relation for $1 \le k \le K$; that is, when the entire join is completed in one "sampling step."

Suppose for simplicity that the blocking factor α is prespecified, so that we need only optimize the aspect-ratio parameters. This is often the case in practice; see [GG97] for rules of thumb when choosing a blocking factor. Our goal is to minimize the confidence interval subject to an upper bound on the product of the β_k's that corresponds to the animation-speed setting. More precisely, we wish to choose $\beta_1, \beta_2, \ldots, \beta_K$ to solve the optimization problem

$$\begin{aligned} &\text{minimize } \sum_{k=1}^{K} \frac{d(k)}{\alpha\beta_k} \\ &\text{such that} \\ &\quad \beta_1\beta_2\cdots\beta_K\alpha^{K-1} \le c, \\ &\quad 1 \le \beta_k \le m_k/\alpha \text{ for } 1 \le k \le K, \\ &\quad \beta_1, \beta_2, \ldots, \beta_K \text{ integer}, \end{aligned} \tag{5.5}$$

where m_k is the cardinality of the kth input relation and the value of the constant c is determined by the position of the **animation speed** slider. (The constant c is permitted to lie anywhere between α^{K-1} and $m_1m_2\cdots m_K/\alpha$.) An exact solution method for the nonlinear integer-programming problem in (5.5) is expensive and complicated to code. For our prototype we use a simple approximate solution algorithm: first solve a relaxed version of (5.5) in which all the constraints but the first are dropped, and then adjust the solution so that the remaining constraints are satisfied. For simplicity, suppose that each $d(k)$ is positive. Then it can be shown [HH98] that the solution $\beta_1^*, \beta_2^*, \ldots, \beta_K^*$ to the relaxed minimization problem is given by

$$\beta_k^* = \left(\frac{c}{d(1)d(2)\cdots d(K)}\right)^{1/K} \alpha^{(1-K)/K} d(k)$$

for $1 \le k \le K$. To adjust this solution, set $\beta_k = \beta_k^*$ for $1 \le k \le K$ and then execute the algorithm in Figure 10. For ease of exposition, we present the algorithm for the special case in which, initially, $\beta_1 \le \beta_2 \le \cdots \le \beta_K$; in general, we sort $\beta_1, \beta_2, \ldots, \beta_K$ in ascending order and then execute the algorithm. The first step of the algorithm is to determine whether at least one β_k is less than 1. If so, then during the first time through the **repeat** loop the algorithm scales up $\beta_1, \beta_2, \ldots, \beta_K$ proportionately so that each β_k is greater than or equal to 1, as required by the second constraint in (5.5). Observe, however, that this scaleup will cause the first constraint in (5.5) to be violated. To handle this problem, the algorithm then executes one or more scaling-down steps (the remaining iterations of the **repeat** loop): at each such step, those β_k's that exceed 1 are scaled down proportionately until either the first constraint in (5.5) is satisfied (in which case the scaling-down phase terminates) or the smallest of these β_k's is scaled down to 1. This procedure is repeated until the first constraint in (5.5) is satisfied. Finally,

```
if (β_1 < 1) {
  j := 0;
  repeat {
    j := j + 1;
    a := min((β_j β_{j+1} ··· β_K α^{K-1}/c)^{1/(K-j+1)}, β_j)
    for k = j, j+1, ..., K {
      β_k := β_k / a;
    }
  }
  until (β_1 β_2 ··· β_K α^{K-1} ≤ c or j = K);
}
for k = 1, 2, ..., K {
  β_k := min(⌊β_k⌋, ⌊m_k/α⌋);
}
```

Figure 10: Algorithm for modification of $\beta_1, \beta_2, \ldots, \beta_K$ in block ripple join.

[6] It might be tempting to try and select an aspect ratio such that the confidence-interval length is minimized at each time point. Unfortunately, such an aspect ratio does not exist in general.

[7] For simplicity, we assume that a constant I/O cost is incurred per tuple scanned; our basic approach can be extended to more complicated I/O models.

each β_k is decreased further if necessary to ensure that the remaining constraints in (5.5) are satisfied. If $\beta_1, \beta_2, \ldots, \beta_K$ have a greatest common divisor (GCD) that is greater than 1, we can divide each β_k by this GCD—this modification increases interactivity without affecting statistical efficiency.

5.3.2 Hash Ripple Joins

For hash ripple join, the approach to choosing the aspect-ratio parameters is similar; see [HH98] for details. The main difference from block ripple join is that the cumulative I/O cost for n sampling steps of a hash ripple join is proportional to $(\beta_1 + \beta_2 + \cdots + \beta_K)n$, since each input tuple is read from disk exactly once. Thus the appropriate optimization problem is of the form (5.5) with the first constraint changed to: $\beta_1 + \beta_2 + \cdots + \beta_K \leq c$. It can be shown that the solution $\beta_1^*, \beta_2^*, \ldots, \beta_K^*$ to the corresponding relaxed problem is given by $\beta_k^* = c\sqrt{d(k)} / \sum_{j=1}^{K} \sqrt{d(j)}$ for $1 \leq k \leq K$. This solution can then be adjusted to satisfy the remaining constraints using an algorithm almost identical to that in Figure 10; see [HH98].

5.3.3 Multiple Aggregates

Many aggregation queries encountered in practice, such as queries with a `GROUP BY` clause, require computation of several aggregates simultaneously. One approach to setting the aspect-ratio parameters when there are multiple aggregates is to minimize a weighted average of the squared confidence interval lengths, that is, to minimize $w_1\epsilon_{n,1}^2 + w_2\epsilon_{n,2}^2 + \cdots + w_m\epsilon_{n,m}^2$, where m is the number of aggregates, $\epsilon_{n,j}$ $(1 \leq j \leq m)$ is the length of the confidence interval for the jth running estimate after n sampling steps, and $w_1, w_2, \ldots, w_m$ are weights chosen by the user. Since, by our previous discussion, each $\epsilon_{n,j}^2$ can be written in the form

$$\epsilon_{n,j}^2 = \frac{z_p^2}{n} \sum_{k=1}^{K} \frac{d(k;j)}{\alpha\beta_k},$$

it follows that the appropriate minimization problem is of the form (5.5) with $d(k) = \sum_{j=1}^{m} w_j d(k;j)$. This approach is easy to implement; more sophisticated approaches are possible, but they require solution of a more complex minimization problem than the one in (5.5).

5.4 Implementation Issues

Given the foregoing framework, ripple join can be designed to adaptively set its aspect ratio by estimating the optimal β_k's at the end of each sampling step. The idea is to replace each $d(k)$ with its estimator $\widehat{d}_n(k)$ before solving the optimization problem in (5.5). Lacking any initial information about the optimal values of the β_k's, we start the join with each β_k equal to 1 in order to get a high initial tuple delivery rate. The β_k estimates can fluctuate significantly at first, but typically stabilize quickly. A large, poorly chosen aspect ratio can result in a long period of ineffective processing. Thus it is best to postpone updating the initial aspect ratio until at least 20 to 30 tuples have passed the `WHERE` clause, and then use a stepwise approach for adjustment: when the estimated optimal β_k is far from its current value, the new value can be set to a fractional point (e.g. halfway) between the current value and its newly estimated optimum.

When the aspect ratio is changed at the end of a sampling step, the new sampling step must "wrap" the current hyper-rectangle as described in Section 4 so that the length of the kth side $(1 \leq k \leq K)$ becomes an appropriate multiple of the updated value of β_k. For example, consider our two-table query at the end of step $n = 2$, with blocking factor $\alpha = 1$ and aspect ratio specified by $\beta_1 = 2$ and $\beta_2 = 3$. At this point, the ripple join has swept out a 4×6 rectangle. Suppose that it is beneficial to change the aspect-ratio parameters to $\beta_1 = \beta_2 = 1$. Then at the end of the next sampling step the ripple join should have swept out a 7×7 rectangle. Note that at the end of the step we have jumped from $n = 2$ to $n = 7$; such jumps do not present difficulties to our estimation methods.

In the remainder of this section we outline algorithms for computing the variance-constant estimator $\widehat{\sigma}_n^2$ that determines the half-width ϵ_n of the confidence intervals as in (5.3). Consider, for example, a `SUM` query as in Section 5.2.2. In order to update $\widehat{\sigma}_n^2$ at the end of a sampling step, we first need to update the quantities $\widehat{\sigma}_n^2(R)$ and $\widehat{\sigma}_n^2(S)$. In the following we focus on updating methods for $\widehat{\sigma}_n^2(R)$; these methods apply to $\widehat{\sigma}_n^2(S)$ virtually unchanged. Recall that $\widehat{\sigma}_n^2(R)$ is the variance of the numbers in the set $\mathcal{I} = \{\, \widehat{\mu}_n(r;R) \colon\ r \in R_n \,\}$, and that each $\widehat{\mu}_n(r;R)$ is the average of $|R| \cdot |S| \cdot \mathit{expression}_p(r,s)$ over all $s \in S_n$. Observe that, at each sampling step, we add new elements to $\mathcal{I}$ (which correspond to new tuples from R) and also possibly modify some of the existing elements of $\mathcal{I}$ (which correspond to old tuples from R that join with new tuples from S). Our goal when updating $\widehat{\sigma}_n^2(R)$ is to minimize the amount of recomputation required at each sampling step. To this end, we use the fact [CGL83] that if we augment a set of n numbers with average A_1 and variance V_1 by adjoining a set of m numbers with average A_2 and variance V_2, then the variance V of the augmented set of $n + m$ numbers is

$$V = \frac{n}{m+n}V_1 + \frac{m}{m+n}V_2 + \frac{mn}{(m+n)^2}(A_1 - A_2)^2. \quad (5.6)$$

Using this composition formula, we proceed as follows. At the beginning of each sampling step, we update $\widehat{\sigma}_n^2(R)$ under the "optimistic" assumption that all new observations $\mathit{expression}_p(r,s)$ obtained during the sampling step will be identically zero. The reason for this approach is that in practice many observations $\mathit{expression}_p(r,s)$ are in fact equal to 0 because r doesn't join with s. The initial update is easy to apply: it can be shown that the effect of changes in the existing entries can be incorporated into $\widehat{\sigma}_n^2(R)$ simply by multiplying[8] the old value of $\widehat{\sigma}_n^2(R)$ by $\big((n-1)/n\big)^3$, and then all of the new (zero) entries can be incorporated via a single computation based on (5.6). Each nonzero observation actually encountered during the sampling step results in changes to one or more elements of $\mathcal{I}$. For each changed element, we run the composition formula in (5.6) "backwards" to remove the element from the $\widehat{\sigma}_n^2(R)$ computation, update the element, and then run the formula forwards to incorporate the modified element back into the $\widehat{\sigma}_n^2(R)$ computation.

The above updating approach works for any `SUM` or `COUNT` query; see [HH98] for the complete algorithm. For `AVG` queries we also need to update the covariance statistic $\widehat{\rho}_n$ introduced in Section 5.2.2. The updating method is almost identical to that for $\widehat{\sigma}_n^2$; see [HH98]. In practice, the computation cost for the updating algorithms is minimal. Memory consumption, however, is proportional to the number of tuples that have passed the `WHERE` clause so far. This increasing storage requirement can become burdensome after a significant period of processing. Typically, we expect

[8] For the general case of $K > 2$ input relations, the multiplicative factor is $\big((n-1)/n\big)^{2K-1}$.

the user to abort the query before the storage problem becomes severe—tight estimates are usually obtained quickly when the output relation is of non-trivial size. If the user does not abort the query, then several approaches are available for handling the later stages of query processing. One approach is to switch to a "conservative" or "deterministic" confidence interval as described in [HHW97]; such intervals typically are longer than large-sample intervals but have no additional storage requirements. Another approach is to also process the query using standard "batch" techniques; this batch processing can be performed in parallel with the online execution, and the user can switch over to batch mode as desired. Alternatively, all of the statistics except the current running estimate and confidence-interval length can be discarded and a new running aggregation computation initiated; the new running estimate and confidence-interval length can be combined with the previous running estimate(s) and confidence-interval length(s) to yield final running results. We hope to explore this last approach in future work.

6 Performance

In this section, we present results from an implementation of the ripple join algorithm in POSTGRES[9]; these results illustrate the functionality of the algorithm and expose tradeoffs in online performance. We used data from the University of Wisconsin that comprise the course history of students enrolled over a three-year period. Our experiments focus on two tables: **student**, which contains information about students at the university, and **enroll**, which records information about students' enrollment in particular classes. The **student** table has 60,300 rows, and in POSTGRES occupies about 22.3 Mb on disk; the **enroll** table has 1,547,606 rows, and occupies about 327.0 Mb on disk. Records are clustered in random order on disk, so a scan yields a random sample. Our version of POSTGRES does not support histograms, and hence makes radically incorrect selectivity estimates. We augmented POSTGRES by providing the equivalent of a 20-bucket equi-width histogram per column from the command line as needed; a standard DBMS would typically provide at least this much accuracy [IBM97, Inf97, Ora97].

The two relevant performance metrics for online aggregation are the rate at which the length of the confidence interval decreases (i.e., the precision of the display over time) and the rate at which the user receives new updates (i.e., the animation speed).

In our first experiment we ran the following query:

```
SELECT ONLINE AVG(enroll.grade) FROM enroll,student
WHERE enroll.sid = student.sid
  AND student.honors_code IS NULL;
```

We ran the query for 60 seconds, using block-, hash-, and index-ripple join (i.e., index nested-loops join), along with classical block nested-loops join. In order to avoid exaggerating the effectiveness of ripple join, we attempted to make nested-loops as competitive as possible. We therefore built an index over the far bigger **enroll** table, which is used by the indexed join. We also forced the POSTGRES optimizer to choose the smaller **student** relation as the "inner" of the block nested-loops join, since this is more effective for online aggregation. Finally, we set the animation speed to 100%, which makes block- and hash-ripple join very interactive, but hampers their ability to shrink confidence intervals as discussed in Section 5.3.1.

[9]Our implementation is based on the publicly available PostgreSQL distribution [Pos98], Version 6.3. Our measurements were performed on a PC with an Intel Pentium Pro 200 Mhz processor and 256 Kb cache, 128 Mb RAM, running the RedHat Linux 5.1 distribution (kernel version 2.0.34). One 6.4 Gb Quantum Fireball ST6.4A EIDE disk was used to hold the database, and another 2.1 Gb Seagate ST32151N SCSI disk held the operating system and core applications, home directories, swap space, and POSTGRES binaries. POSTGRES was configured with 10,000 8-Kb buffers in its buffer pool.

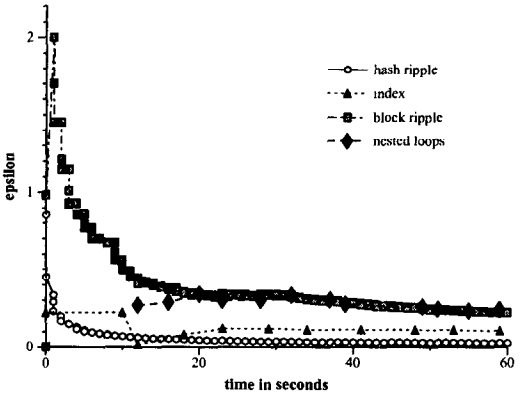

Figure 11: Confidence-interval half-width (ϵ) over time.

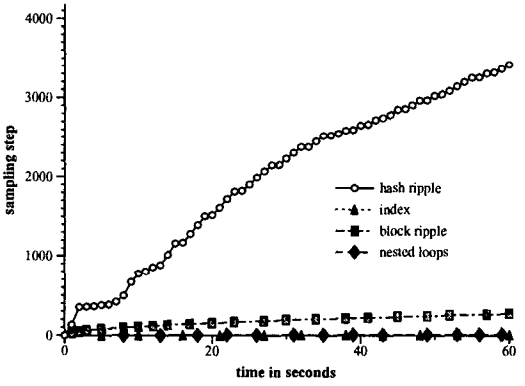

Figure 12: Number of sampling steps completed over time.

Figure 11 shows how the confidence interval half-width ϵ shrinks over time for the query. Despite the maximal animation speed, hash ripple join gives extremely tight confidence intervals within seconds, performing comparably to index ripple join (but without requiring an index). By contrast, block ripple join shrinks much more slowly. Note the initial instability in the block ripple join estimator, before the number of sampling steps is sufficiently large for the CLT-based estimator; cf. Figure 12. This effect could have been masked by using a (much wider) conservative confidence interval until a small number of tuples had been fetched. Index ripple join also demonstrates some instability during its startup phase.

To compare the ripple joins with traditional algorithms, note that nested loops join takes over 10 seconds to even begin giving estimates—this is because each sampling step requires a full scan of the **student** table. The best batch join algorithm is hybrid hash join, which is the choice of the POSTGRES optimizer for this query. Hybrid hash runs for 208 seconds before completing the join, at which point it produces a precise result. Even if we replaced POSTGRES with the world's fastest database software, our system's disk

transfer rate[10] of 8.2Mb/sec would still require about 42 seconds simply to read the two relations into memory. Note that online aggregation in POSTGRES produces very tight bounds a full two orders of magnitude faster than batch mode in the same system, and one order of magnitude faster than an ideal system. Presumably an ideal system would also do online aggregation considerably faster than POSTGRES as well. The performance advantage of ripple join would increase if the size of the relations being joined were increased—calculations similar to [HNS94] show that for a key/foreign-key join as in our example, the I/O cost ratio of hybrid hash to ripple join increases roughly as the square root of the table cardinality due to the beneficial effects of sampling.

While block ripple join looks quite unattractive in this example, it is important to note that this key/foreign-key equijoin discards most tuples, with only 1 in every 63,446 tuples of the cross-product satisfying the WHERE clause. Block ripple join is more appropriate for joins with large result sizes. For such joins a large fraction of the cross-product space contributes to the output, so that ϵ shrinks at an acceptable rate despite the high I/O cost. Moreover, whenever the result size is large because the join is a non-equijoin, then block ripple is applicable but hash ripple is not.

In the previous example, the high animation speed forced a square aspect ratio. To demonstrate the advantages of adaptive aspect-ratio tuning at lower animation speeds, we consider a query that returns the average ratio of Education student to Agriculture student grades, normalized by year[11]:

```
SELECT ONLINE AVG(d.grade/a.grade)
FROM enroll d, enroll a
WHERE d.college = 'Education'
  AND a.college = 'Agriculture'
  AND a.year = d.year;
```

We ran this query using block ripple joins of differing aspect ratios; POSTGRES chose block ripple in this case because the result size of the join is quite large. The resulting performance is shown in Figure 13. In all the joins, the Education relation instance was the left operand, and the Agriculture relation instance was the right; the curves are labeled with the left×right aspect ratio. As can be seen from Figure 13, it is best to sample the Agriculture relation instance at a much higher rate than Education. The adaptive block ripple join's aspect ratio starts at the initial default value of 1×1 and then, after some fluctuation, settles to a ratio of around 1×6 in favor of Education. Note the relatively smooth shape of the curve for the square aspect ratio, which produces a result as often as possible (animation speed set to the maximum.) By contrast, the other curves—particularly the adaptive curve, which had animation speed set to 90%—have a "staircase" shape, reflecting long sampling steps during which running estimates remain fixed. This clearly illustrates the tradeoff between estimation quality and animation speed that was described in Section 5.

To once more compare online to batch performance, we tried this query in POSTGRES. The POSTGRES optimizer chose a naive nested loops join, and the query took so long to complete that it had to be aborted. The most sensible batch algorithm for this low-selectivity query is block nested-loops join. An idealized implementation would make one pass of `enroll` for the outer loop, and $\lceil |\texttt{enroll}|/|\text{buffer pool}| \rceil = \lceil 327/80 \rceil = 5$ passes of `enroll` for the inner relation. Assuming buffer pool hits on memory-resident portion of `enroll`, each inner pass would require 327-80=247Mb of I/O, for a total of $327 + 5 * 247 = 1562$ I/Os. At the peak transfer rate of 8.2 Mb/s, this would require about 190 seconds to complete—between one and two orders of magnitude longer than required for ripple join to produce good estimates in POSTGRES.

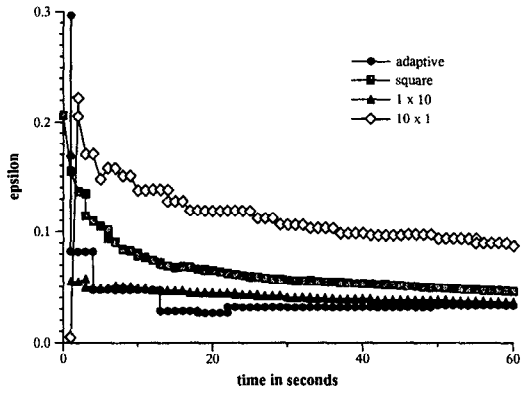

Figure 13: Confidence-interval half-width (ϵ) over time for block ripple joins of differing aspect ratios.

7 Conclusions and Future Work

A complete implementation of online aggregation must be able to handle multi-table queries. This paper introduces ripple joins, a family of join algorithms designed to meet the performance needs of an online aggregation system. Ripple joins generalize block nested-loops and hash join, and automatically adjust their behavior to provide precise confidence intervals while updating the estimates at a rapid rate. Users can trade off precision and updating rate on the fly by changing an "animation speed" parameter that controls the aspect ratio of the rectangles swept out by the join. In our experiments, the time required to produce reasonably precise online estimates was up to two orders of magnitude smaller than the time required for the best offline join algorithms to produce exact answers. A key observation is that the time required to achieve a confidence interval of a specified "acceptable" length is a sub-linear (and sometimes constant!) function of the cardinality of the input relations; cf [HNS94]. It follows that, as the size of databases increases, online join algorithms should appear more and more attractive relative to their offline counterparts for a wide variety of queries.

This paper opens up a number of areas for future work. Although the ripple join is symmetric, it is still not clear how a query optimizer should choose among ripple join variants, nor how it should order a sequence of ripple joins. As we have seen in this paper, the optimization goals for an online aggregation system are different than for a traditional DBMS: even for a simple binary nested-loops join, the traditional choice of outer and inner is often inappropriate in an online scenario.

[10] We measured disk transfer rate using raw I/Os on the Quantum Fireball holding the data. The rate of 8.2Mb/sec is the best-case behavior at the outer tracks; the inner tracks provide only about 5.1 Mb/sec.

[11] Although this query is actually a self-join, we process it as a binary join. We can do this because the rows for Agriculture students and Education students form two disjoint subtables of the `enroll` table. The idea is independently sample a row (or set of rows) from each subtable at every sampling step. Moreover, for this query we can use catalog statistics to obtain the precise cardinalities of the two subtables. See Section 7 for further discussion of self-joins.

Another challenge is the development of efficient techniques for processing self-joins that avoid the need for two separate running samples from the input table; such self-joins arise naturally in a variety of queries. When the input $expression_p(r, s)$ to a SUM or AVG aggregation function is a symmetric function of r and s, it appears that results for "U-statistics" [Hoe48] can be used to obtain confidence-interval formulas based on a single running sample. This approach needs to be developed and extended to deal with arbitrary self-join queries.

Although the POSTGRES DBMS was useful for rapid prototyping, there are a number of performance issues that need to be studied in an industrial-strength system. One important area is the parallelization of ripple joins. If base relations are horizontally partitioned across processing nodes, random retrieval of tuples from different nodes can be viewed as a stratified sampling scheme, and the confidence-interval formulas presented here can be adjusted accordingly. The "stratified" estimates generated at the nodes must be combined in an efficient manner to yield an overall running estimate and a corresponding confidence interval. To study these and other issues, we are currently implementing ripple join and hash ripple join in high-performance, parallel commercial DBMS's.

In this paper we present ripple joins in the context of a statistical estimation problem. We believe, however, that ripple joins will be useful for other, non-statistical modes of data exploration, particularly data visualization. We are currently exploring online visualization [HAR99], and plan to test the effectiveness of ripple join at producing quick, meaningful visualizations of very large data sets.

Acknowledgements

The following people provided helpful suggestions on early drafts of this paper: Eric Anderson, Paul Aoki, Vijayshankar Raman and Megan Thomas. Remzi Arpaci-Dusseau provided the code to measure disk transfer rates. The second author 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. The Wisconsin student database was graciously provided by Bob Nolan of the UW-Madison Department of Information Technology (DoIT).

References

[AS72] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions.* Dover, New York, 1972. Ninth printing.

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

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

[DKO+84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Michael R. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. 1984 ACM SIGMOD Intl. Conf. Managment of Data*, pages 1–8. ACM Press, 1984.

[GG97] J. Gray and G. Graefe. The five-minute rule ten years later and other computer storage rules of thumb. *SIGMOD Record*, 26(4), 1997.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surveys*, 25(2):73–170, June 1993.

[Haa97] P. J. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *Proc. Ninth Intl. Conf. Scientific and Statist. Database Management*, pages 51–63. IEEE Computer Society Press, 1997.

[HAR99] J. M. Hellerstein, R. Avnur, and V. Raman. Informix under CONTROL: Online query processing. Submitted for publication, 1999.

[HH98] P. J. Haas and J. M. Hellerstein. Join algorithms for online aggregation. IBM Research Report RJ 10126, IBM Almaden Research Center, San Jose, CA, 1998.

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

[HN96] J. M. Hellerstein and J. F. Naughton. Query Execution Techniques for Caching Expensive Methods. In *Proc. 1996 ACM SIGMOD Intl. Conf. Managment of Data*, pages 423–424. ACM Press, 1996.

[HNS94] P. J. Haas, J. F. Naughton, and A. N. Swami. On the relative cost of sampling for join selectivity estimation. In *Proc. Thirteenth ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Sys.*, pages 14–24. ACM Press, 1994.

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

[Hoe48] W. Hoeffding. A class of statistics with asymptotically normal distribution. *Ann. Math. Statist.*, 19:293–325, 1948.

[HOT88] W. Hou, G. Ozsoyoglu, and B. Taneja. Statistical estimators for relational algebra expressions. In *Proc. Seventh ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Sys.*, pages 276–287. ACM Press, 1988.

[HOT89] W. Hou, G. Ozsoyoglu, and B. Taneja. Processing aggregate relational queries with hard time constraints. In *Proc. 1989 ACM SIGMOD Intl. Conf. Managment of Data*, pages 68–77. ACM Press, 1989.

[IBM97] IBM Corporation. *IBM DB2 Universal Database Administration Guide, Version 5.* North York, Ontario, Canada, 1997.

[Inf97] Informix Corporation. *Informix Universal Server Guide to SQL: Syntax, Version 9.01.* Menlo Park, CA, March 1997.

[ODT+91] G. Ozsoyoglu, K. Du, A. Tjahjana, W. Hou, and D. Y. Rowland. On estimating COUNT, SUM, and AVERAGE relational algebra queries. In D. Dimitris Karagiannis, editor, *Database and Expert Systems Applications, Proceedings of the International Conference in Berlin, Germany, 1991 (DEXA 91)*, pages 406–412. Springer-Verlag, 1991.

[OJ93] V. O'day and R. Jeffries. Orienteering in an information landscape: How information seekers get from here to there. In *Human Factors in Computing Systems: INTERCHI '93 Conf. Proc.*, pages 438–445. ACM Press, 1993.

[Olk93] F. Olken. *Random Sampling from Databases.* Ph.D. Dissertation, University of California, Berkeley, CA, 1993. Available as Tech. Report LBL-32883, Lawrence Berkeley Laboratories, Berkeley, CA.

[Ora97] Oracle Corporation. *Oracle8 Server SQL Reference, Release 8.0.* Redwood Shores, CA, June 1997.

[Pos98] PostgreSQL Home Page, 1998. http://www.postgresql.org.

[RRH99] V. Raman, B. Raman, and J. M. Hellerstein. Online dynamic reordering for interactive data processing. Technical Report UCB//CSD-99-1043, Computer Science Division, UC Berkeley, 1999. Submitted for publication.

[RSS94] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. *Trans. Knowledge and Data Engrg.*, 6(4):501–517, 1994.

[WA91] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. First Intl. Conf. Parallel and Distributed Info. Sys. (PDIS)*, pages 68–77. IEEE Computer Society Press, 1991.