RJ 10126 (95001) June 30, 1998
Computer Science

Research Report

JOIN ALGORITHMS FOR ONLINE AGGREGATION

Peter J. Haas

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Joseph M. Hellerstein

Computer Science Division
University of California
Berkeley, CA 94720-1776

IBM Research Division
Yorktown Heights, New York • San Jose, California • Zurich, Switzerland

JOIN ALGORITHMS FOR ONLINE AGGREGATION

Peter J. Haas

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
e-mail: peterh@almaden.ibm.com

Joseph M. Hellerstein

Computer Science Division
University of California
Berkeley, CA 94720-1776
e-mail: jmh@cs.berkeley.edu

ABSTRACT: We provide a new family of join algorithms, called ripple joins, for online processing of complex, multi-table aggregation queries in a relational database management system (DBMS). Such queries arise naturally in interactive exploratory decision-support applications. As opposed to traditional offline join algorithms, which are designed to minimize the time to completion of the query, 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. This time is independent of the size of the input relations. Ripple joins are adaptive, adjusting their behavior in accordance with the statistical properties of the data, and permit the user to trade off the time between the 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 provide formulas and algorithms both for computing confidence intervals and for adaptively setting 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.

Keywords: relational database, aggregation queries, join methods, sampling, online processing, decision support

1. Introduction

Current relational database management systems do not handle *ad hoc* decision-support queries well, even though such queries are becoming increasingly important in applications. Many decision-support queries consist of a complex sequence of joins and selections over large tables, followed by grouping of the result and computation of aggregates over the groups. The input tables often are extremely large, and standard OLAP techniques may not be directly applicable, either because the *ad hoc* queries are not simple aggregations over datacubes or the data has not been preprocessed into a special warehouse format. Current systems process *ad hoc* queries in *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 are typically used to get a "big picture" of a data set, a more attractive approach is to perform *online aggregation*, in which progressively-refined running aggregates are continuously displayed to the user. An online aggregation systems attempts 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 facilitate efficient online processing for multi-table queries.

Ripple joins generalize traditional nested-loops and hash joins, and are designed to meet the performance needs of an online query processing system. Ripple joins are non-blocking, thereby permitting the running aggregates 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 running aggregate approaches the final result at each update.

In this paper we present naive, block, index, and hash variants of the ripple join. We also derive statistical formulas for running estimators and confidence intervals that generalize those developed for simpler join algorithms, and we present techniques for adaptively tuning the parameters of the algorithms. Finally, we demonstrate the utility and efficiency of ripple joins with results from our implementation in the POSTGRES DBMS.

1.1. Motivation

As 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;
```
(1.1)

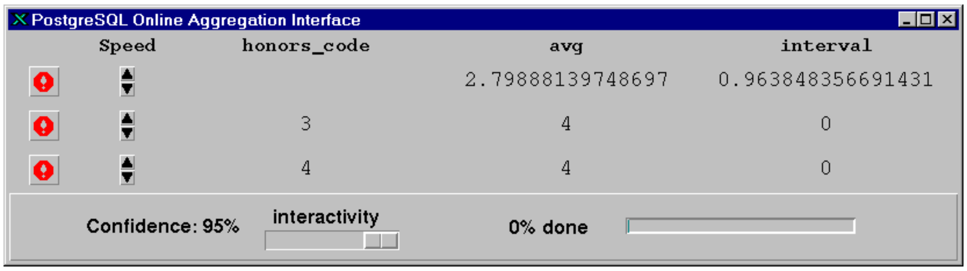

Figure 1: An online aggregation interface.

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 average of all grades found so far (after less than 1% of the records in the Cartesian product have been scanned) is displayed in the column entitled `avg`. The "stop-sign" buttons can be used to pause the 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. As described in subsequent sections, tuples are processed in random order, so that each running aggregate can be viewed as a statistical estimator of the final query result. The estimated proximity of a running aggregate to its final value can therefore be indicated by means of an associated *running confidence interval*. For example, the running average of about 2.799 for the first group in Figure 1 is within ± 0.9639 of the final average with probability 95%.

The goal of the current work is to extend the above interface to permit online processing not only of single-table queries, but of multi-table queries of the form

$$
\boxed{\begin{array}{l}
\texttt{SELECT } \mathit{op}(\mathit{expression}) \\
\texttt{FROM } R_1, R_2, \ldots, R_K \\
\texttt{WHERE } \mathit{predicate} \\
\texttt{GROUP BY } \mathit{columns};
\end{array}}
$$

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. Such an extension clearly requires that the database support the relational join operator. 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 hybrid 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 $|S| < |R|$. At each sampling step, a random tuple s is retrieved from S. 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 aggregate 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 even moderately large (as is often the case for decision-support queries), then the amount of time between successive updates to the running aggregate 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 arguments of the aggregation function consist of the values in a single column of S (i.e., the values in the columns of R have no effect on the value of the aggregation function). 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 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 combat the above problems. The user can explicitly trade off the time between successive updates of the running aggregate with the amount by which the confidence-interval length decreases at each update. This tradeoff is effected using the `interactivity` slider shown in Figure 1 and discussed in detail in Section 5. By adjusting the interactivity setting, we obtain a family of algorithms that covers the entire spectrum of possible algorithms alluded to above.

1.2. 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]. This topic also has been treated in [GGMS96, HNSS96, HNS94, HS92, HS95, HOD91, LN90, LNS90, LNSS93],

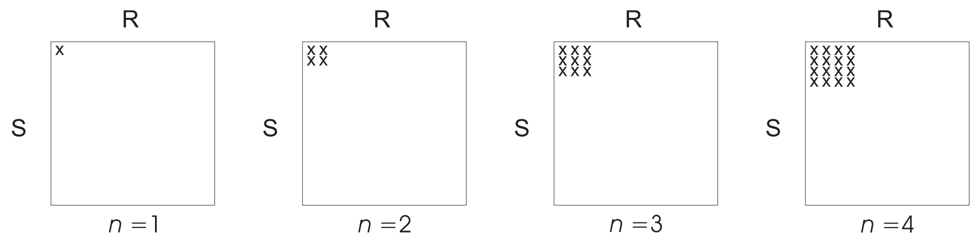

Figure 2: The elements of $R \times S$ that have been seen after n sampling steps of a "square" ripple join ($n = 1, 2, 3, 4$).

usually under the assumption that there exists an index on one or more of the base relations. Techniques that are applicable to other types of aggregation queries follow from results in Olken and Rotem [OR86] and Ozsoyoglu, et al. [ODT$^+$91]; the "acceptance/rejection" sampling techniques described in these papers do not appear directly applicable to online aggregation.

Confidence-interval formulas for multi-table aggregation queries are given in [HHW97] and derived in [Haa96, Haa97]. Although these formulas implicitly assume that some sort of nested-loops join or ripple join is used, the algorithmic aspects are not treated in these papers and there are no performance comparisons. Moreover, all the formulas in [HHW97, Haa96, Haa97] assume that the samples are retrieved at the same rate from each of the base relations; as shown below, it can be much more efficient to use different sampling rates for different base relations. The new contributions of the current paper with respect to the previous papers include: practical implementation details for explicit algorithms, the generalization of previous confidence-interval formulas to handle nonhomogeneous sampling rates, simple cost models for ripple join, the formulation and solution of the optimal-sampling-rate problem, and a performance comparison of different online join algorithms.

The ripple join algorithm presented here 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. As noted in Section 3.5, the more recent pipelining hash join of [WA91] has a similar flavor as well. Neither of these earlier techniques considers the relative rates of the two operands, which is a major focus of our work here. Of the two prior ideas, pipelining hash join is a more natural predecessor for online query processing (and was proposed for use in [HHW97]), but it is designed for relations that fit in memory, which is atypical in the decision-support scenarios most relevant to online aggregation.

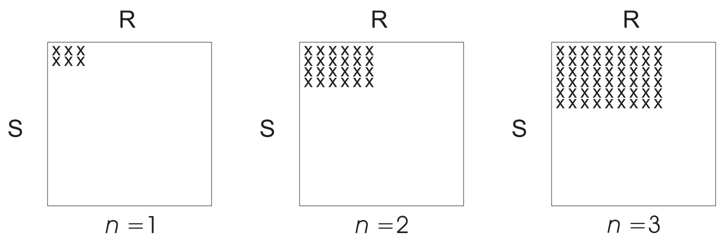

Figure 3: The elements of $R \times S$ that have been seen after n sampling steps of a "rectangular" ripple join ($\beta_1 = 3$, $\beta_2 = 2$, $n = 1, 2, 3$).

2. 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.[1]

The "square" version of the ripple join described above draws samples from R and S at the same rate. In order to provide the shortest possible confidence intervals, it is often necessary to sample one relation at a higher rate than another. 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 $\beta_2 = 2$.

When $\beta_K = 1$ and $\beta_{K-1} = |R_{K-1}|, \ldots, \beta_1 = |R_1|$, the ripple join reduces to a nested-loops join; see Figures 4(a)–4(c) for $K = 2$. In Figure 4(d), each point $(r, s) \in R \times S$

[1]Retrieving tuples from a relation in random order is the same as simply scanning the relation either when the tuples have been randomly permuted prior to query processing or when, as verified by statistical testing, the storage order of the tuples on disk is independent of the values of the attributes involved in the aggregation query. Alternatively, it may be desirable to either sample 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.

[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".

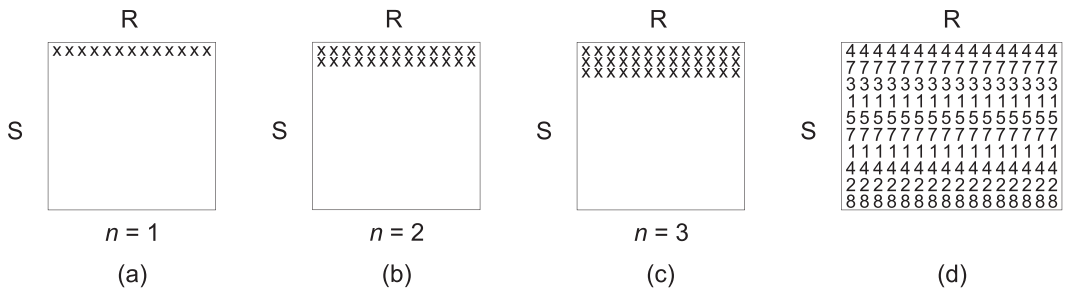

Figure 4: The elements of $R \times S$ that have been seen after n sampling steps of an online nested-loops join ($n = 1, 2, 3$) and a worst-case scenario for online nested-loops join.

```
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.

is represented by the argument of the aggregation function produced from r and s; the values displayed in this figure correspond to the problematical extreme case discussed in the previous section. In choosing a nested-loops join, a traditional query optimizer would take S to be the outer relation in this case, since $|S| < |R|$ in Figure 4(d). This decision is clearly incorrect for the purposes of online aggregation; the optimizer's mistake is in not explicitly taking the statistical characteristics of the data into consideration.

3. Ripple Join Algorithms

As noted above, a ripple join with an extremely skewed aspect ratio is equivalent to a traditional nested-loops join. In fact, ripple join is 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 $\texttt{max} = n$ 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 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.

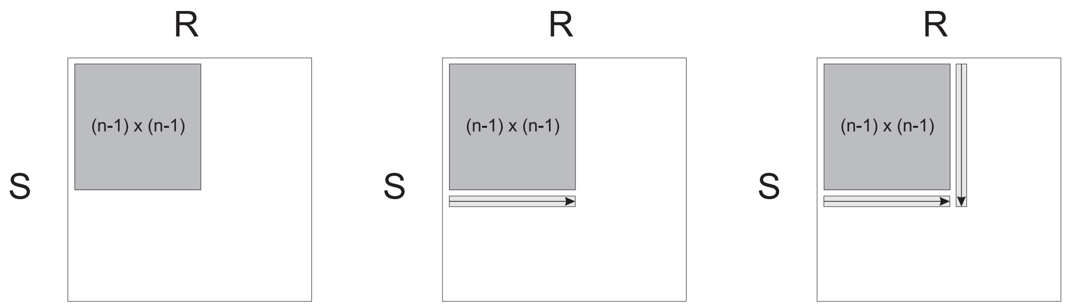

Figure 6: Three phases of a sampling step for a square ripple join.

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 order. First, since most DBMS's use a "pull" or *iterator* model for relational operators [Gra93], we present iterator implementations in the rest of the paper. Second, we handle non-unitary aspect ratios, which require some care at the end of a sampling step. Third, we enhance the performance of simple ripple join with blocking, indexing, and hashing techniques. Finally, we consider the details involved in pipelining multiple binary ripple joins in a query plan tree.

3.1. A Square Naive 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;       // current sampling step
    relation R, S;     // operands
    relation currel;  // relation currently acting as ´inner´
    boolean innerlooping; // are we in midst of inner loop?
    init() {
        R.pos = 1;  S.pos = 0; // cursor positions in R and S
        curstep = 1;
        currel = S;
        innerlooping = true;
    }
    next() {
      do { // loop until return() is called
        if (innerlooping) { // scanning one 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])
                } }
            innerlooping = false; // no more matches for currel.pos
        }
        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;
            innerlooping = true;
        } } } }
```

Figure 7: A simple iterator for square ripple join.

3.2. Non-Unitary Aspect Ratios and the Final Join Phase

As described 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 one, 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

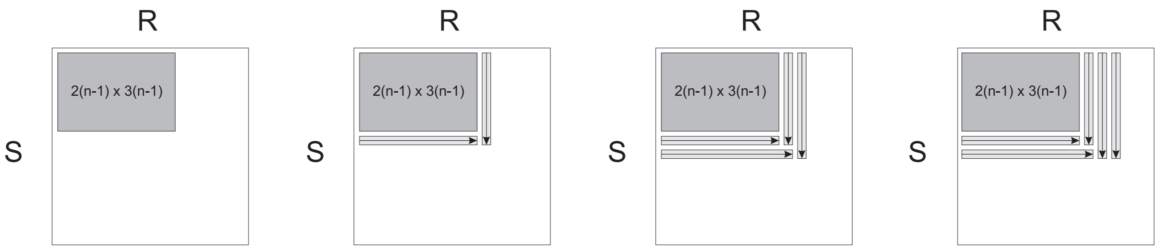

Figure 8: Four phases of a sampling step for a rectangular ripple join with $\beta_1 = 3$ and $\beta_2 = 2$.

combines it with all tuples seen so far from the previous relation. The resulting traversal of $R \times S$ is illustrated in Figure 8. Note in the illustration how the traversal first wraps the entire original rectangle with $\min(\beta_1, \beta_2)$ "layers" and then pads out the longer side with $\max(\beta_1, \beta_2) - \min(\beta_1, \beta_2)$ additional layers in a nested-loops fashion[3].

Until now we have ignored the final phase of the join, in which the smaller operand of the join has been fully scanned, signifying that the highest position visited equals the cardinality. Our mechanism for handling non-unitary aspect ratios provides a ready means of dealing with this final phase. All the tuples of a fully-scanned relation are "old", so the iterator described above combines the remaining tuples of the larger relation with the tuples of the smaller in traditional nested-loops fashion. This behavior falls out naturally because no "new" tuple is ever reached again in the smaller relation. A small amount of additional logic is needed to deal with problems that arise when S has fewer tuples than R. In this case, the next sampling step cannot start with the retrieval of new tuples from S as in all of the previous sampling steps. These difficulties are avoided by changing the "starter role" on the fly: when the starter relation (S) reaches its maximal tuple, the starter role is switched to the other relation (R).

3.3. Block Ripple Join

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. When a new block of one relation (say R) is read from disk, each tuple in that block is compared with all the old tuples of the other relation, S. When a new tuple of S is reached, 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. Visually, one

[3]An alternative, reasonable approach is to pad one side with β_1 layers of length $(n-1)\beta_2$, and the other with β_2 layers of length $n\beta_1$ for sampling step n; the pseudocode for that is a minor variation, and has no effect on confidence intervals or number of I/Os. Although the choice between the two has, in principle, some implications for buffer management, we will see in Sections 3.3 and 3.5 that block ripple join and hash ripple join manage their memory explicitly, so that such buffer management issues do not actually arise.

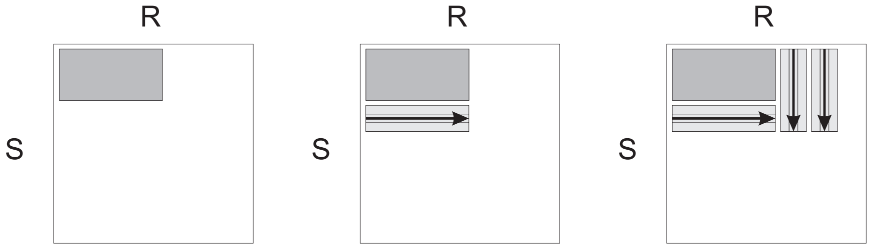

Figure 9: Three phases of a sampling step for a rectangular block ripple join with $\beta_1 = 2$, $\beta_2 = 1$, and block size = 3 tuples.

can think of block ripple join the same way as standard ripple join, but with "thick" arrows consisting of multiple tuples at once (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.

3.4. Index Ripple Join

Consider a binary join with input relations R and S, and suppose that there is an index on the join attributes of R. The performance of the nested-loops join depicted in Figure 4 can be improved by using 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 there is no choice of aspect ratio for index ripple joins — each complete probe of the index on R 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.

3.5. Hash Ripple Join

Simple ripple join is a generalization of nested-loops join, and as such shares (and in fact aggravates) the inefficiencies of nested-loops join. In particular, in a complete ripple join of two relations, each tuple from one relation is read and re-read from the disk numerous times. For equijoin queries, it is natural to consider improving ripple join using hashing techniques. 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). Wilschut and Apers suggested this scheme for a main-memory parallel DBMS [WA91], though their discussion does not consider non-unitary aspect ratios.

This 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. It is tempting in this context to utilize the ideas of hybrid hash join [DKO$^+$84] for spilling overflow tuples to disk. Unfortunately, this approach results in an algorithm that is *not* a ripple join — in addition to sweeping out rectangles of increasing perimeter length, it also sweeps out certain stray "lines" that lie outside the bounds of the current rectangle. Since the resulting statistical properties of the running aggregate in such an algorithm are uncertain at best, we do not discuss this approach further. Moreover, we will see in Section 6 that very tight confidence intervals often can be achieved long before memory is filled.

3.6. Pipelining Multiple Ripple Joins

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

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 $R_1, \dots, R_j$ and subtree S contains relations $S_1, \dots, S_k$, 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; the identity of the starter relation may change during subsequent processing. In general, when the cursor of a non-starter relation R

exceeds $n\beta_R$ for the nth sampling step, it returns a failure code as if it had reached end-of-file. When the cursor of a starter relation, say S, exceeds $n\beta_S$, it increments the sampling step to $n+1$.

If the right operand of the root node is fully scanned before the left operand, then the starter role is passed to the left operand. If this left operand is not a base relation, it assigns the starter role to its own right operand. If, however, the left operand is a base relation, it becomes the starter relation. When the leftmost relation in the tree is the starter relation and becomes fully scanned, the query is complete[4].

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

A fully-detailed iterator for ripple join is shown in Appendix A. The pseudocode includes support for multiple joins, non-unitary aspect ratios, and relations of different sizes. Initially, all aspect-ratio parameters are set to 1; they are then dynamically modified at the end of certain sampling steps, as described in Section 5.

4. Confidence Intervals for Block Ripple Joins

As discussed in [Haa96, Haa97, HHW97], there are several types of running confidence intervals that can be constructed after n sampling steps of the block ripple join algorithm. These include *conservative* confidence intervals, which contain the final value μ of the running aggregate with a probability that is greater than or equal to a prespecified probability p, and *large-sample* confidence intervals, which contain μ with a probability approximately equal to p and are based upon central limit theorems. We focus primarily on large-sample intervals because they are applicable to a wider range of aggregates than conservative intervals and, when both types of interval are applicable, the large-sample interval usually is much shorter. Large-sample confidence intervals are appropriate when n is small enough so that the tuples retrieved so far can be treated as a sample drawn with replacement but large enough so that approximations based on CLT's are accurate. (For well-behaved data, large-sample intervals can be accurate when n is as small as 20 to 40 sampling steps.) Note that the formulas and algorithms developed below are also applicable to hash ripple joins: in Phase 1 a hash ripple join can be treated as a block ripple join with unitary blocking factor and in Phase 2 the hash ripple join becomes a block ripple join.

We focus on methods for computing running aggregates and (large-sample) confidence intervals when a block ripple join is used to process a query of the form

$$\boxed{\begin{array}{l}\texttt{SELECT}\ op(expression)\\ \texttt{FROM}\ R_1, R_2, \ldots, R_K\\ \texttt{WHERE}\ predicate;\end{array}} \tag{4.1}$$

[4]The discussion here assumes a left-deep tree for simplicity. In principle, it is possible to construct bushy trees of ripple joins; this might be advantageous for hash ripple joins in a parallel environment, for example. In such a scenario, the starter role must be passed among the leaves of the tree in a different fashion.

where $K \geq 2$ and *op* is one of `COUNT`, `SUM`, or `AVG`. We assume that the blocking factor α and the aspect-ratio parameters $\beta_1, \beta_2, \ldots, \beta_K$ are fixed, and set $B = \beta_1 \beta_2 \cdots \beta_K$. (The quantities $\alpha, \beta_1, \beta_2, \ldots, \beta_K$ are expressed in units of tuples. Methods for determining these parameters are discussed in Section 5.) Throughout, we denote the tuples in R_k $(1 \leq k \leq K)$ by $t_{k,1}, t_{k,2}, \ldots, t_{k,m_k}$, where m_k is the number of tuples in R_k. We also denote by $L_{k,i}$ the random index of the ith new tuple retrieved from relation R_k; that is, the ith tuple retrieved from R_k is tuple $t_{k,L_{k,i}}$. We assume that for each k the random variables $L_{k,1}, L_{k,2}, \ldots$ are each uniformly distributed on $\{1, 2, \ldots, m_k\}$ and are independent (at least to a good approximation); see [Haa97] for further discussion.

4.1. SUM and COUNT Queries

We focus first on the case in which *op* is equal to `COUNT` or `SUM`. Set $v(i_1, i_2, \ldots, i_K)$ equal to $m_1 m_2 \cdots m_K$ times the value of *expression* when computed for tuples $t_{1,i_1}, t_{2,i_2}, \ldots, t_{K,i_K}$ if these tuples jointly satisfy *predicate*; otherwise, set $v(i_1, i_2, \ldots, i_K) = 0$. (We assume that if *op* is equal to `COUNT`, then the value of *expression* is equal to 1 for all combinations of tuples from the base relations.) Denote by $\tilde{\mathcal{F}}$ the set of real-valued functions defined on $\Lambda = \{1, 2, \ldots, m_1\} \times \cdots \times \{1, 2, \ldots, m_K\}$, and for $f \in \tilde{\mathcal{F}}$ set

$$\tilde{\theta}(f) = \frac{1}{m_1 m_2 \cdots m_K} \sum_{i_1=1}^{m_1} \sum_{i_2=1}^{m_2} \cdots \sum_{i_K=1}^{m_K} f(i_1, i_2, \ldots, i_K). \tag{4.2}$$

With these definitions, the final answer $\tilde{\mu}$ to the query in (4.1) can be written as

$$\tilde{\mu} = \tilde{\theta}(v). \tag{4.3}$$

Suppose that n sampling steps of the ripple join have been executed. The running aggregate for a `COUNT` or `SUM` query is $\tilde{Y}_n = \tilde{T}_n(v)$, where[5]

$$\tilde{T}_n(f) = \frac{1}{B(\alpha n)^K} \sum_{i_1=1}^{\alpha\beta_1 n} \sum_{i_2=1}^{\alpha\beta_2 n} \cdots \sum_{i_K=1}^{\alpha\beta_K n} f(L_{1,i_1}, L_{2,i_2}, \ldots, L_{K,i_K}). \tag{4.4}$$

for $f \in \tilde{\mathcal{F}}$.

For this running aggregate, we wish to obtain a $100p\%$ confidence interval of the form $I_n = [\tilde{T}_n(v) - \epsilon_n, \tilde{T}_n(v) + \epsilon_n]$, set $M = m_1 m_2 \cdots m_K$, and then set

$$\tilde{\theta}(f; k, j) = \frac{m_k}{M} \sum_{i_1=1}^{m_1} \cdots \sum_{i_{k-1}=1}^{m_{k-1}} \sum_{i_{k+1}=1}^{m_{k+1}} \cdots \sum_{i_K=1}^{m_K} f(i_1, \ldots, i_{k-1}, j, i_{k+1}, \ldots, i_K) \tag{4.5}$$

[5]In general, we use Greek letters to denote quantities that are computed over all of the tuples in the base relations, while corresponding Roman letters denote the same quantities, but computed over only the tuples that have been retrieved after n sampling steps.

for $1 \leq k \leq K$, $1 \leq j \leq m_k$, and $f \in \tilde{\mathcal{F}}$. Also set

$$\zeta(f;k) = \frac{1}{m_k} \sum_{j=1}^{m_k} \Bigl(\tilde{\theta}(f;k,j) - \tilde{\theta}(f) \Bigr)^2 \tag{4.6}$$

and

$$\tilde{\theta}_2(f) = \sum_{k=1}^{K} \frac{\zeta(f;k)}{\alpha\beta_k}. \tag{4.7}$$

It follows from a generalization of the proof of Theorem 6 in [HNSS96] that, as n becomes large, the distribution of the running aggregate $\tilde{Y}_n$ becomes approximately normal with mean $\tilde{\theta}(v)$ and variance $\tilde{\theta}_2(v)/n$. We can estimate the unknown parameter $\tilde{\theta}_2(v)$ by $\tilde{T}_{n,2}(v)$ where, for $f \in \tilde{\mathcal{F}}$,

$$\tilde{T}_{n,2}(f) = \sum_{k=1}^{K} \frac{Z_n(f;k)}{\alpha\beta_k}, \tag{4.8}$$

$$Z_n(f;k) = \frac{1}{\alpha\beta_k n - 1} \sum_{j=1}^{\alpha\beta_k n} \Bigl(\tilde{T}_n(f;k,j) - \tilde{T}_n(f) \Bigr)^2, \tag{4.9}$$

and

$$\begin{aligned} \tilde{T}_n(f;k,j) &= \frac{\beta_k}{B(\alpha n)^{K-1}} \\ &\times \sum_{i_1=1}^{\alpha\beta_1 n} \cdots \sum_{i_{k-1}=1}^{\alpha\beta_{k-1} n} \sum_{i_{k+1}=1}^{\alpha\beta_{k+1} n} \cdots \sum_{i_K=1}^{\alpha\beta_K n} f(L_{1,i_1}, \ldots, L_{k-1,i_{k-1}}, L_{k,j}, L_{k+1,i_{k+1}}, \ldots, L_{K,i_K}). \end{aligned} \tag{4.10}$$

Given this estimate of $\tilde{\theta}_2(v)$, a standard argument (cf. [Haa97]) yields the following formula for the confidence-interval half-length ϵ_n:

$$\epsilon_n = \left(\frac{z_p^2 \tilde{T}_{n,2}(v)}{n} \right)^{1/2}, \tag{4.11}$$

where z_p is the $(p+1)/2$ quantile of the standard normal distribution function. Observe that the length of the confidence interval $(2\epsilon_n)$, expressed as a fraction of the sum or count, depends on the statistical characteristics of the data (i.e., the value of $\tilde{\theta}_2(v)$) and the number of samples taken (n), but not on the size of the input relations. Thus, as the sizes of these relations increase, the cost of ripple join relative to any given offline join algorithm decreases (assuming that the statistical characteristics of the tables remain invariant under scaling).

The quantity $\tilde{T}_{n,2}(v)$ in (4.11) can be computed by a generalized version of the algorithm in [Haa97, Figure 2], using the formula in [CGL83, Equation (1.5b)] for computing the combined variance of two groups of observations from the individual variance for each group. We describe the procedure for the case in which the blocking factor and aspect-ratio parameters remain fixed during aggregation processing. (In general, we allow these parameters to change over time in an adaptive manner; see Section 5.) Initialize the computation by setting $x_1 = \cdots = x_K = 0$, $w_1 = \cdots = w_K = 0$, and $n = 0$. After each sampling step of the block ripple join, execute the algorithm in Figure 10 with $\Delta n = 1$ and with $l_k = \alpha\beta_k(n-1)$ and $\delta_k = \alpha\beta_k$ for $1 \leq k \leq K$. In the algorithm, l_k $(1 \leq k \leq K)$ denotes the number of tuples read from R_k prior to the current sampling step and δ_k denotes the number of tuples read from R_k during the current sampling step. Moreover, J denotes the set of index vectors $(i_1, i_2, \ldots, i_K)$ that correspond to new nonzero observations:

(i) $l_k < i_k \leq l_k + \delta_k$ for some $1 \leq k \leq K$; and

(ii) $f(L_{1,i_1}, L_{2,i_2}, \ldots, L_{K,i_K}) \neq 0$.

See [HNSS96] for a detailed explanation of a similar algorithm. The idea is that each $Z_n(f;k)$ in (4.8) can be viewed as the variance of the "data points" $\tilde{T}_n(f;k,1), \tilde{T}_n(f;k,2), \ldots, \tilde{T}_n(f;k,\alpha\beta_k n)$, so that recursive schemes for computing the variance are applicable. The other key idea is to exploit the fact that most new observations of the form $f(L_{1,i_1}, \ldots, L_{K,i_K})$ obtained during a given sampling step are likely to be equal to zero. Therefore, we first use the formula in [CGL83] to update each Z_n under the assumption that each new observation obtained during the sampling step will be equal to 0. (This step corresponds to the first outer **for** loop in Figure 10.) Then for each nonzero observation, the formula in [CGL83] is used twice, first in a backward manner to remove an (erroneous) zero observation and then in the normal forward manner to incorporate the actual nonzero observation. (This step corresponds to the second outer **for** loop in Figure 10.) Just after the algorithm has executed for the nth time, the variable z is equal to $\tilde{T}_{n,2}(f)$, the variable y is equal to $\tilde{T}_n(f)$, each variable $w_k/(L'_k - 1)$ is equal to $Z_n(f;k)$, and each variable $q_{k,j}/L'_k$ is equal to $\tilde{T}_n(f;k,j)$. It can be seen that after n sampling steps, the cumulative computation cost is roughly $O(KM_n)$, where $M_n = \sum_{i=1}^{n} J_i$ and J_i is the number of elements in the set J during the ith execution of the algorithm. Assuming that sparse storage techniques are used, the cumulative storage required is $O(V_n)$, where $V_n = \sum_{k=1}^{K} \sum_{j=1}^{n} B_n(f;k,j)$ and

$$B_n(f;k,j) = \begin{cases} 1 & \text{if } \tilde{T}_n(f;k,j) \neq 0; \\ 0 & \text{otherwise.} \end{cases}$$

At the start of a sampling step, the algorithm in effect assumes that all new observations of the form $f(L_{1,i_1}, \ldots, L_{K,i_K})$ seen during the sampling step will be equal to zero, and updates the running statistics accordingly. Then, for each non-zero observation that is actually encountered during sampling and using appropriate updating formulas, a zero value is removed from the running statistics and the true observation is added.

```
n := n + Δn;
for k = 1, 2, ..., K {
  l'_k := l_k + δ_k;
  L_k := ∏_{j≠k} l_j;
  L'_k := ∏_{j≠k} l'_j;
  x_k := x_k L_k / L'_k;
  for l = l_k + 1, l_k + 2, ..., l_k + δ_k {
    q_{k,l} := 0;
  }
  if n > 1 {
    w_k := w_k (L_k / L'_k)^2 + δ_k (x_k / l_k)(x_k / l'_k);
  }
}
for (i_1, ..., i_K) ∈ J {
  for k = 1, 2, ..., K {
    j := i_k;
    d := q_{k,j} / L'_k;
    x_k := x_k − d;
    w_k := w_k − (x_k − (l'_k − 1)d) / l'_k)
             × (x_k − (l'_k − 1)d) / (l'_k − 1);
    q_{k,j} := q_{k,j} + f(L_{1,i_1}, ..., L_{K,i_K});
    d := q_{k,j} / L'_k;
    w_k := w_k + (x_k − (l'_k − 1)d) / l'_k)
             × (x_k − (l'_k − 1)d) / (l'_k − 1);
    x_k := x_k + d;
  }
}
y := x_1 / l'_1;
z := Σ_{k=1}^{K} w_k / ((l'_k / n)(l'_k − 1));
```

Figure 10: An algorithm for updating $\tilde{T}_{n,2}(f)$ and $\tilde{T}_n(f)$.

In practice, the computation cost for the confidence intervals (over and above the cost of computing the aggregate itself) is minimal. The storage requirement is proportional to the number of tuples from the output relation that have been produced so far. This increasing storage requirement can become burdensome after the initial stages of processing. Typically, however, we expect the user to abort the query before the storage problem becomes severe. If the user does not abort the query, several approaches are available for handling the later stages of query processing. One approach is to switch to a conservative confidence interval or deterministic confidence interval as described in [HHW97]; such intervals are wider 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. Finally, the statistics can be discarded and a new running aggregation computation initiated; the new running aggregate can be combined with the previous running aggregate(s) to yield a final running result. (This approach will be investigated in future work.)

4.2. AVG Queries

We now consider a query of the form (4.1) with *op* equal to `AVG`. Set $v(i_1, i_2, \ldots, i_K)$ equal to the value of *expression* when applied to tuples $t_{1,i_1}, t_{2,i_2}, \ldots, t_{K,i_K}$ (whether or not these tuples jointly satisfy *predicate*). Denote by S the subset of $R_1 \times R_2 \times \cdots \times R_K$ such that $(t_{1,i_1}, t_{2,i_2}, \ldots, t_{K,i_K}) \in S$ if and only if tuples $t_{1,i_1}, t_{2,i_2}, \ldots, t_{K,i_K}$ jointly satisfy *predicate*. Set $u(i_1, i_2, \ldots, i_K) = 1$ if $(t_{1,i_1}, t_{2,i_2}, \ldots, t_{K,i_K}) \in S$ and $u(i_1, i_2, \ldots, i_K) = 0$ otherwise. If *predicate* is empty, so that the average is computed over the Cartesian product of the input relations, then the final answer to the query can be written in the form (4.3) and the confidence-interval formulas of Section 4.1 apply. If *predicate* is not empty, then the final answer $\tilde{\mu}(S)$ can be written in the form $\tilde{\mu}(S) = \tilde{\theta}(uv)/\tilde{\theta}(u)$, where $\tilde{\theta}(f)$ is defined as in (4.2) and $uv(i_1, \ldots, i_K) = u(i_1, \ldots, i_K)v(i_1, \ldots, i_K)$ for $(i_1, i_2, \ldots, i_K) \in \Lambda$. After n sampling steps, the running aggregate is given by $\tilde{Y}_n(S) = \tilde{T}_n(uv)/\tilde{T}_n(u)$, where $\tilde{T}_n(f)$ is defined by (4.4).

As with a `COUNT` or `SUM` query, the goal is to obtain a $100p\%$ confidence interval for the running aggregate of the form $I_n = [\tilde{Y}_n(S) - \epsilon_n, \tilde{Y}_n(S) + \epsilon_n]$. We proceed in a manner similar to the case of `COUNT` and `SUM`. Define $\tilde{\theta}(f; k, j)$ and $\tilde{\theta}_2(f)$ as in (4.5) and (4.7), respectively. Set

$$\zeta(f, g; k) = \frac{1}{m_k} \sum_{j=1}^{m_k} \Big(\tilde{\theta}(f; k, j) - \tilde{\theta}(f)\Big)\Big(\tilde{\theta}(g; k, j) - \tilde{\theta}(g)\Big) \tag{4.12}$$

for $1 \leq k \leq K$ and $f, g \in \tilde{\mathcal{F}}$, and set $\tilde{\theta}_{1,1}(f, g) = \sum_{k=1}^{K} \zeta(f, g; k)/(\alpha\beta_k)$. It follows from a generalization of the argument in [Haa97, Section 3.2] that, after n sampling steps of the ripple join algorithm (with n moderately large), the distribution of the running aggregate $\tilde{Y}_n(S)$ is approximately normal with mean $\tilde{\mu}(S)$ and variance $\tilde{\gamma}/\big(n\tilde{\theta}^2(u)\big)$, where $\tilde{\gamma} = \tilde{\theta}_2(uv) - 2\tilde{\rho}\tilde{\theta}_{1,1}(uv, u) + \tilde{\rho}^2\tilde{\theta}_2(u)$ and $\tilde{\rho} = \tilde{\theta}(uv)/\tilde{\theta}(u)$. For $f, g \in \tilde{\mathcal{F}}$, set

$$Z_n(f, g; k) = \frac{1}{\alpha\beta_k n - 1} \sum_{j=1}^{\alpha\beta_k n} \Big(\tilde{T}_n(f; k, j) - \tilde{T}_n(f)\Big)\Big(\tilde{T}_n(g; k, j) - \tilde{T}_n(g)\Big) \tag{4.13}$$

and

$$\tilde{T}_{n,1,1}(f, g) = \sum_{k=1}^{K} \frac{Z_n(f, g; k)}{\alpha\beta_k}, \tag{4.14}$$

where $\tilde{T}_n(f; k, j)$ is defined as in (4.10). We can then estimate $\tilde{\gamma}/\tilde{\theta}^2(u)$ by $\tilde{G}_n/\tilde{T}_n^2(u)$, where $\tilde{T}_n(u)$ is defined via (4.4), $\tilde{G}_n = \tilde{T}_{n,2}(uv) - 2\tilde{R}_n\tilde{T}_{n,1,1}(uv, u) + \tilde{R}_n^2\tilde{T}_{n,2}(u)$, and $\tilde{R}_n = \tilde{T}_n(uv)/\tilde{T}_n(u)$. A standard argument then yields the final result

$$\epsilon_n = \left(\frac{z_p^2\tilde{G}_n}{n\tilde{T}_n^2(u)}\right)^{1/2}. \tag{4.15}$$

As with `SUM` and `COUNT` queries, the length of the confidence interval depends only on the statistical characteristics of the data and the number of samples taken.

Note that we must compute $\tilde{T}_{n,2}(uv)$, $\tilde{T}_{n,2}(u)$, $\tilde{T}_n(uv)$, $\tilde{T}_n(u)$, and $\tilde{T}_{n,1,1}(uv, u)$ in order to compute $\tilde{G}_n$ and hence the confidence-interval length. The first four quantities can be computed using the algorithm in Figure 10. An almost identical algorithm can be used to compute the quantity $\tilde{T}_{n,1,1}(uv, u)$; see Appendix B.

4.3. Other Types of Aggregation Queries

Confidence intervals can be obtained for various types of aggregation queries not explicitly discussed in the current paper. For example, queries with `GROUP BY` or `DISTINCT` clauses can be handled as discussed in Section 2.4 of [Haa97]. Similarly, confidence-interval formulas for `VARIANCE` and `STDEV` queries can be obtained by appropriately extending the formulas given in [Haa97]. Moreover, an extension of the general approach outlined in [Haa97] can be used to obtain confidence-interval formulas for new aggregation functions as they are encountered in practice.

5. Determining the Aspect Ratio

Consider a general aggregation query of the form (4.1) involving K base relations. The two key goals of a ripple join algorithm are to maximize the rate at which the running aggregate and confidence intervals are updated and to maximize the amount by which the length of the confidence interval decreases at each update. These goals are typically contradictory: increasing the number of samples obtained between successive updates increases the time between the updates but usually results in a larger interval-length decrease at each update. Our approach is to permit the user to determine the tradeoff between these two goals by setting a lower bound on the updating rate. In our example interface of Figure 1, this tradeoff is controlled via the "interactivity" slider. Given a specified "interactivity" setting, we then attempt to select values of the aspect-ratio parameters and the blocking factor (the latter in the case of block ripple join) that will minimize the length of the running confidence interval.

To select optimal values for the ripple join parameters, we combine the confidence-interval formulas developed in Section 4 with an I/O cost model for the ripple join. In this paper we sketch a simplified approach to the optimization process in which we estimate I/O costs using a simple conservative model that assumes one I/O per tuple scanned. In the case of block ripple join we simplify the problem further by simply selecting a reasonable blocking factor α and then optimizing the aspect-ratio parameters $\beta_1, \beta_2, \ldots, \beta_K$. A more sophisticated approach is outlined in Appendix E; in this appendix we indicate how to jointly optimize the blocking factor and aspect-ratio parameters, as well as how to take into account page-level I/O.

5.1. Block Ripple Join

Consider a block ripple join with blocking factor α and aspect-ratio parameters $\beta_1, \beta_2, \ldots, \beta_K$, and suppose that *op* is equal to `SUM` or `COUNT` in (4.1). As shown in Appendix C, the cumulative number of I/O's required for n sampling steps of a block ripple join is $\beta_1\beta_2\cdots\beta_K\alpha^{K-1}n^K + o(n^K)$. 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 smaller this quantity and the faster the updating rate. (The updating rate is maximized when $\beta_1 = \beta_2 = \cdots = \beta_K = \alpha = 1$; that is, when we have a non-blocked square ripple join.)

On the other hand, recall that the confidence interval length is computed as in (4.11) after n sampling steps. Also recall that the key quantity $\tilde{T}_{n,2}(v)$ that appears in (4.11) estimates the parameter $\tilde{\theta}_2(v)$ defined via (4.7). The larger the β_k's, the smaller the quantity $\tilde{\theta}_2(v)$ and hence the smaller the confidence-interval length at each update. (The confidence-interval length is minimized when $\beta_k = m_k/\alpha$ for $1 \leq k \leq K$; that is, when we have a nested-block join.)

To deal with these conflicting requirements on the β_k's, we attempt to minimize $\tilde{\theta}_2(v)$, subject to a lower bound on the updating rate or, equivalently, an upper bound on the quantity $\beta_1\beta_2\cdots\beta_K\alpha^{K-1}$. Specifically, given a user-selected interactivity value $0 \leq q \leq 1$, we wish to select values $\beta_1, \beta_2, \ldots, \beta_K$ that solve the optimization problem

$$\begin{aligned} &\text{minimize} \sum_{k=1}^{K} \frac{\zeta(v;k)}{\alpha\beta_k} \\ &\text{such that} \\ &\qquad \beta_1\beta_2\cdots\beta_K\alpha^{K-1} \leq c(q), \\ &\qquad 1 \leq \beta_k \leq m_k/\alpha \text{ for } 1 \leq k \leq K, \\ &\qquad \beta_1, \beta_2, \ldots, \beta_K \text{ integer}, \end{aligned} \tag{5.1}$$

where $\zeta(\,\cdot\,;k)$ is as in (4.6) and $c(q)$ is a smooth decreasing function of q with $c(0) = m_1m_2\cdots m_K/\alpha$ and $c(1) = \alpha^{K-1}$. One suitable function is

$$c(q) = \left\lfloor \left(\frac{m_1m_2\cdots m_K}{\alpha^K}\right)^{(1-q)^h} \alpha^{K-1} \right\rfloor,$$

where h is a fixed system parameter and $\lfloor x \rfloor$ denotes the largest integer less than or equal to x; this choice of c ensures that the interactivity slider behaves in a responsive but stable manner. The interactivity parameter q determines the tradeoff between the updating rate and the confidence-interval length. When $q = 1$ (maximum interactivity), we have $c(q) = \alpha^{K-1}$, and the only feasible solution to the problem in (5.1) is $\beta_1 = \beta_2 = \cdots = \beta_K = 1$. Thus, the updating rate is maximized and we have a square ripple join. When $q = 0$, it is easy to see that the solution to the problem is given by $\beta_k = m_k/\alpha$ for $1 \leq k \leq K$, and we have a nested-block join. (In this case, the base relation R_k with the smallest value of $\zeta(v;k)$ is selected as the outermost relation.) It follows from the implementation details of the algorithm that, given a solution $\beta_1, \beta_2, \ldots, \beta_K$ with greatest common divisor equal to $\gamma > 1$, the actual behavior of the algorithm is unchanged if each β_k is divided by γ.

Of course, in practice the parameters $\zeta(v;1), \zeta(v;2), \ldots, \zeta(v;K)$ are unknown. However, we can use the following two-phase approach to select the aspect-ratio parameters. Initially, set $\beta_1 = \beta_2 = \cdots = \beta_K = 1$. After a specified number of sampling steps, estimate each $\zeta(v;k)$ by $Z_n(v;k)$, defined via (4.9) and computed as described in Section 4. Next, update the aspect-ratio parameters by solving the optimization problem (5.1), but with each $\zeta(v;k)$ replaced by $Z_n(v;k)$. Then, during the next sampling step, "wrap" the current hyper-rectangle as described in Section 3 so that the length of the kth side $(1 \leq k \leq K)$ becomes an appropriate multiple of the updated value of β_k. Now continue to use the ripple join algorithm with the new parameter values. If the aspect ratio is readjusted periodically, then the block ripple join becomes fully adaptive.

The procedure by which quantities such as $\tilde{T}_n(f)$ and $\tilde{T}_{n,2}(f)$ are updated must be modified slightly from the procedure described in Section 4 when the aspect ratio is readjusted. Specifically, suppose that the ripple join algorithm has executed sampling step n and the aspect-ratio parameters are updated prior to the next sampling step. Then, after the next sampling step, the quantities $\tilde{T}_n(f)$ and $\tilde{T}_{n,2}(f)$ are updated using the algorithm in Figure 10, but with different values for the parameters $\Delta n, l_1, l_2, \ldots, l_K$ than those given in Section 4. For example, suppose that $K = 2$, $n = 2$, $\alpha = 1$, $\beta_1 = 2$, and $\beta_2 = 3$, so that just after sampling step $n = 2$ the ripple join has swept out a 4×6 rectangle. Also suppose that, prior to the next sampling step, the aspect-ratio parameters are changed to $\beta_1 = \beta_2 = 1$. Accordingly, at the end of the next sampling step the ripple join has swept out a 7×7 rectangle. (We require that at least one tuple is sampled from each input relation at a sampling step, and so we do not sweep out a 6×6 rectangle.) The quantities $\tilde{T}_n(f)$ and $\tilde{T}_{n,2}(f)$ are then updated using the algorithm in Figure 10 with $\Delta n = 5$, $l_1 = 4$, $l_2 = 6$, $\delta_1 = 3$, and $\delta_2 = 1$. Observe that the new value of the sampling-step index is $n = 7$, which is consistent with the new values of the aspect-ratio parameters and the size of the rectangle that has been swept out. Whenever the aspect-ratio parameters change, we wrap the rectangle in such a way that the value of n is guaranteed to strictly increase.

Exact solution of the nonlinear integer-programming problem in (5.1) can be expensive and complicated to code. In Appendix D, we describe a simple approximate solution algorithm. The idea is to quickly solve a relaxed version of (5.1) in which all the constraints in (5.1) but the first are dropped; this solution is then adjusted in a heuristic manner so that the remaining constraints are satisfied.

Other aggregation functions such as `AVG`, `VARIANCE`, and `STDEV` can be treated in a similar manner. For each of these queries, we need to solve an optimization problem of the form (5.1), except that each $\zeta(v;k)$ is replaced by a parameter ϕ_k that depends on the particular aggregation function. For example, it can be seen from our earlier discussion that, for the case of an `AVG` query, we have

$$\phi_k = \big(\zeta(uv;k) - 2\rho\zeta(uv,u;k) + \rho^2\zeta(u;k)\big)/\tilde{\theta}^2(u)$$

for $1 \le k \le K$, where $\zeta(\,\cdot\,;k)$ and $\zeta(\,\cdot\,,\,\cdot\,;k)$ are defined by (4.6) and (4.12), respectively, and $\rho = \tilde{\theta}(uv)/\tilde{\theta}(u)$ as before. An adaptive approach as described above can be used to select the aspect-ratio parameters: after n sampling steps, estimate the quantity ϕ_k $(1 \le k \le K)$ by

$$D_k = \big(Z_n(uv;k) - 2\tilde{R}_n Z_n(uv,u;k) + \tilde{R}_n^2 Z_n(u;k)\big)/\tilde{T}_n^2(u)$$

and update each β_k accordingly. Here $\tilde{R}_n = \tilde{T}_n(uv)/\tilde{T}_n(u)$ and $\tilde{T}_n(f)$, $Z_n(f;k)$, and $Z_n(f,g;k)$ are defined as in (4.4), (4.9), and (4.13) respectively.

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, $\epsilon_{n,j}$ $(1 \le j \le m)$ is the half-length of the confidence interval for the jth aggregate after n sampling steps, and $w_1, w_2, \ldots, w_m$ are weights chosen by the user. Then the resulting minimization problem is of the form (5.1) with $\phi_k = \sum_{j=1}^{m} w_j \phi_k^{(j)}$, where $\phi_k^{(j)}$ is the numerator of the kth term in the objective function when the minimization problem as in (5.1) is stated for the jth aggregation function individually. 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 (5.1).

5.2. Hash Ripple Join

The above methodology carries over without modification to Phase 2 of the hash ripple join, since during this phase the algorithm behaves like a block ripple join with blocking factor $\alpha = 1$. During Phase 1, the aspect-ratio parameters are optimized in a manner similar to the procedure for block ripple join. It is easy to see that, in Phase 1, the cumulative number of I/O's required for n sampling steps is given by $(\beta_1 + \beta_2 + \cdots + \beta_K)n$, since each input tuple is read from disk exactly once. Assuming that *op* is equal to `SUM` or `COUNT` in (4.1), the relevant optimization problem is

$$\begin{aligned} &\text{minimize} \sum_{k=1}^{K} \frac{\zeta(v;k)}{\beta_k} \\ &\text{such that} \\ &\qquad \beta_1 + \beta_2 + \cdots + \beta_K \le c(q), \\ &\qquad 1 \le \beta_k \le m_k \text{ for } 1 \le k \le K, \\ &\qquad \beta_1, \beta_2, \ldots, \beta_K \text{ integer}, \end{aligned} \tag{5.2}$$

where

$$c(q) = \left\lfloor K \left(\frac{d}{K}\right)^{(1-q)^h} \right\rfloor,$$

d is a memory-capacity parameter that indicates the maximum number of tuples that can be stored in the hash table during Phase 1, and $h = 1$ is a fixed system parameter as before. When the interactivity parameter q is equal to 0, the confidence-interval length is not updated until the end of the hashing phase; when $q = 1$, the interval length is updated every time a tuple is read from each of the input relations.

As with block ripple join, the parameters $\zeta(v;1), \zeta(v;2), \ldots, \zeta(v;K)$ are unknown, and so we use a two-phase or fully adaptive approach. Also as with block ripple join, we can solve the nonlinear integer-programming problem in (5.2) in an approximate manner, using the algorithm described in Appendix D. Aggregation functions such as AVG, VARIANCE, and STDEV, as well as simultaneous multiple aggregates, can be handled as described for block ripple join.

6. Performance

In this section, we present results from an initial implementation of the ripple join algorithm in POSTGRES[6]; 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.

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 interactivity of the display).

In our first experiment we ran the query in (1.1) for two minutes, using square versions of block ripple join and hash ripple join, along with index ripple join and block nested-loops join. For this query and this data, a square aspect ratio is best. 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.

Figure 11 shows how the confidence interval half-width ϵ shrinks over time for one of the groups in the query. For this example, the index ripple and hash ripple join curves happen to overlap entirely — both algorithms obtain enough samples to produce very tight bounds on the estimates in short order. Block ripple takes much longer to drop below $\epsilon = 0.5$, and continues to shrink even more slowly as the sampling steps become larger. Nested-loops join takes almost 35 seconds to even begin giving estimates — this is because each sampling step requires a full scan of the `student` table. Note also the pronounced "staircase" pattern of the nested-loops join. The width of the stairs corresponds to the time required to perform a sampling step (i.e. the time to scan `student`). By comparison, the best offline join algorithm is hybrid hash join, which is the choice of the POSTGRES optimizer for this query.

[6]Our implementation is based on the publicly available PostgreSQL distribution [Pos97], Version 1.3. 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 5.0 distribution (kernel version 2.0.31). 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. The machine was rebooted between query runs to prevent any effects due to operating system caching.

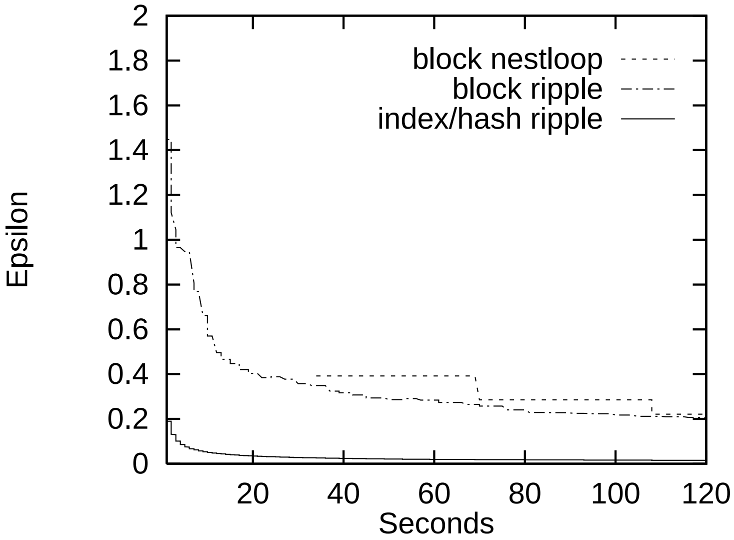

Figure 11: Confidence-interval half-width (ϵ) over time.

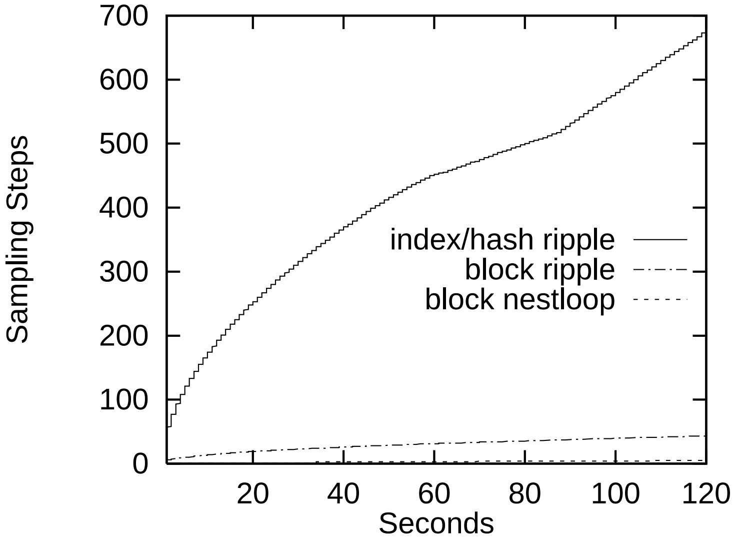

Figure 12: Number of sampling steps over time.

Hybrid hash runs for 446.03 seconds before completing the join, at which point it produces a precise result. Note that ripple hash join produces very tight bounds a full two orders of magnitude faster! This performance ratio would increase if the size of the relations being joined were increased — unlike hybrid hash, the estimation performance of ripple join is not affected by the size of its inputs.

Figure 12 shows the number of sampling steps each algorithm takes over time, which corresponds to the number of times the user sees an update in the estimate. Moreover, for the square ripple joins, the estimates at each step are identical, so this graph also highlights the relative speeds at which the square ripple joins narrow their confidence intervals. As expected, both hash ripple join and index ripple join provide the highest update rate. The update rate for block ripple join is far lower, and the update rate for nested-loops join is barely visible at the x axis. While block ripple join looks relatively unattractive in this scenario, it is important to remember that this particular key/foreign-key equijoin discards most tuples, with only 1 tuple in every 194,778 satisfying the join condition. Block ripple join is suitable for non-equijoin queries, where the fraction of tuples that participate in the join is often far higher, and hence ϵ shrinks at an acceptable rate. Note that nested-loops suffers in both metrics on this query: it is slow to shrink the confidence interval, and provides updates to the estimates very infrequently.

Since a unitary aspect ratio was optimal for the query in (1.1), we present the following query in order to demonstrate the importance of carefully choosing the aspect ratio:

```
SELECT ONLINE AVG(enroll.grade)
FROM enroll first, enroll later
WHERE first.sid = later.sid
AND first.crseid = later.crseid
AND later.passfail = ´t´
AND first.year < later.year;
```

In this query we look at the average grade given to students who subsequently take the very same course "pass/fail" in a later year. We ran the query using hash ripple joins of differing aspect ratios, and the resulting performance is shown in Figure 13. In all the joins, the `first` relation instance was the left operand, and the `later` relation instance was the right; the curves are labeled with the left$\times$right aspect ratio. As can be seen from Figure 13, it is best to sample the `first` relation instance at a much higher rate than the `later` relation instance. The confidence-interval lengths are smaller for the 10×1 curve and this curve stabilizes faster. (The initial instability of the curves is due to the fact that the large-sample confidence interval is not valid until a sufficiently large number of samples are obtained.) Again, the advantage of the hash ripple join (with 10×1 aspect ratio) is striking: hash ripple join produced reasonably precise estimates within a minute, while the best available offline algorithm (hybrid hash join again) required about 25 minutes to complete.

7. Conclusions and Future Work

A complete implementation of online aggregation must be able to handle multi-table queries. This paper presents full details for ripple joins, a family of join algorithms designed to meet the performance needs of an online aggregation system. Ripple joins generalize nested-loops and hash join, and automatically adjust themselves to the performance goals of

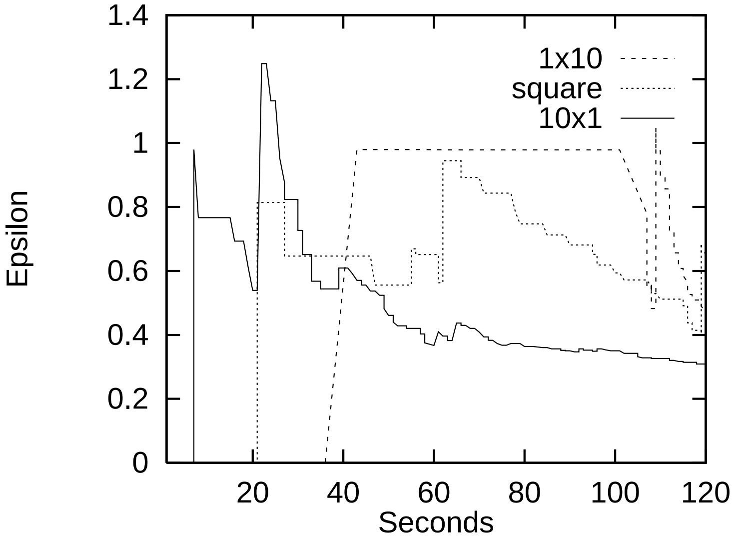

Figure 13: Confidence-interval half-length (ϵ) over time for hash ripple joins of differing aspect ratios.

an online system: maximizing the accuracy of estimates while providing updated estimates at small intervals. The tradeoff between accuracy and updating rate can be made by users on the fly by changing an "interactivity" parameter that controls the aspect ratio of the rectangles swept out by the join. We present the details of the algorithms and, for multi-table queries that involve standard SQL aggregates, we provide techniques for computing running confidence intervals. 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 depends on the number of samples taken and on the statistical properties of the data, but not on the size of the input relations. 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 methods, 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.

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

There are a number of detailed performance issues that need to be studied in a more performance-minded system than POSTGRES. 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 aggregate and a corresponding confidence interval. To study these and other issues, we are currently implementing ripple join and hash ripple join in a high-performance, parallel commercial DBMS.

Acknowledgements

The second author was supported by a grant from Informix Corporation and a California MICRO grant. The Wisconsin student database was graciously provided by Bob Nolan of the UW-Madison Department of Information Technology (DoIT).

References

[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.

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

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

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

[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.

[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.

[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.

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

[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.

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

[HS95] P. J. Haas and A. N. Swami. Sampling-based selectivity estimation using augmented frequent value statistics. In *Proc. Eleventh Intl. Conf. Data Engrg.*, pages 522–531. IEEE Computer Society Press, 1995.

[LN90] R. J. Lipton and J. F. Naughton. Query size estimation by adaptive sampling. In *Proc. Ninth ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Sys.*, pages 40–46. ACM Press, 1990.

[LNS90] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. 1990 ACM SIGMOD Intl. Conf. Managment of Data*, pages 1–11. ACM Press, 1990.

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

[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.

[OR86] F. Olken and D. Rotem. Simple random sampling from relational databases. In *Proc. 12th Intl. Conf. Very Large Data Bases*, pages 160–169, 1986.

[Pos97] PostgreSQL Home Page, 1997. URL http://www.postgresql.org.

[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, Miami Beach, December 1991.

A. Full Iterator Pseudo-Code

```
int curstep; // global to query plan: current sampling step

class RIPL {
  iterator left, right; // operands
  int lcount, rcount;   // our view of the operands´ positions
  int lbeta, rbeta;     // aspect ratio of left and right
  int lmax, rmax;       // maximum position seen so far
  int llimit, rlimit;   // maximum position for this sampling step
  bool leof, reof;      // operand reported end-of-file
  iterator currel;      // relation currently acting as ´inner´
  iterator otherrel;    // the other relation
  bool innerlooping;    // are we in midst of inner loop?
  relation starter;     // which relation (if any) is the starter relation


  // maintain max position so far, to distinguish ``old´´ and ``new´´ tuples
  incr_pos(Relation rel) {
  if (rel == left) {
          lcount++;
          if (lcount > lmax) lmax = lcount;
      } else {
          rcount++;
          if (rcount > rmax) rmax = rcount;
  }  }
  pos(Relation rel)    { if (rel == left) return lpos;  else return rpos;  }
  repos(Relation rel)  { if (rel == left) lpos = 0; else rpos = 0;  }
  max(Relation rel)    { if (rel == left) return lmax;  else return rmax; }
  eof(Relation rel)    { if (rel == left) return leof; else return reof; }
```

```
seteof(Relation rel) { if (rel == left) leof = TRUE; else reof = TRUE;  }
init() {
    lcount = 0; rcount = 0; // cursor positions in left and right
    lmax = rmax = 0;
    llimit = rlimit = 1; // set up to fetch 1,1
    leof = reof = false;
    curstep = 1;
    currel = left; otherrel = right;
    innerlooping = true;
    if (this node is the root of the tree) starter = right;
    else starter = NULL;
}
next() {
  do { // loop until return() is called
      if (innerlooping) { // scanning one side of a rectangle
        while (pos(currel) < max(currel)) {
          currel.next();  incr_pos(currel);
          if (!eof(currel)) {
             if (predicate(R[R.pos],S[S.pos]))
                 return(R[R.pos], S[S.pos])
          } }
        innerlooping = false; // no more matches for otherrel.pos
      }
      else { // done with one side of a rectangle, pos(currel)==max(currel)
          if (pos(currel) < limit(currel)
                   && !eof(currel))
              swap(currel,otherrel); // at corner, start a new layer
          else if (eof(currel) && !eof(otherrel) && starter==currel) {
              // time to hand off starter role
              if (otherrel is a join) {
                  starter = NULL;
                  otherrel.starter = otherrel.inner;
              } else starter = otherrel;
          }
          else if ((pos(currel) == limit(currel) || eof(currel))
                   && (pos(otherrel) == limit(otherrel) || eof(otherrel))
                   {
            // we are at end of step
            if (starter == currel) {
              updateStep();
              swap(currel,otherrel);
            }
            else return(NULL); // must be at eof for both rels
          }
          if (!otherrel.next()) {
             innerlooping = false;  // either pad side again, or step´s done
             if (pos(otherrel) < limit(otherrel))
               seteof(otherrel);  // must have hit end-of-file
             swap(currel,otherrel); // always finish looping on currel
          }
          else {
```

```
              repos(currel) = 0;          // now proceed with a new layer
              incr_pos(otherrel);
              innerlooping = true;
          }
  } } }
  updateStep() {
    if (left is a join) {
       left.updateStep();
       llimit = left.llimit * left.rlimit;
    }
    else llimit += lbeta;
    rlimit += rbeta;
  }

}
```

B. An Updating Algorithm for $\tilde{T}_{n,1,1}(f,g)$

The algorithm in Figure 14 can be used to update the quantity $\tilde{T}_{n,1,1}(f,g)$ after a sampling step of a block ripple join. This quantity (with $f = uv$ and $g = u$ as in Section 4.2) is used in the computation of running confidence intervals for aggregation queries of the form (4.1) with *op* equal to `AVG`. Initialize the computation by setting $x_1^{(f)} = \cdots = x_K^{(f)} = 0$, $x_1^{(g)} = \cdots = x_K^{(g)} = 0$, $w_1 = \cdots = w_K = 0$, and $n = 0$. After each sampling step of the block ripple join, execute the algorithm given in Figure 14 with $\Delta n = 1$ and with $l_k = \alpha\beta_k(n-1)$ and $\delta_k = \alpha\beta_k$ for $1 \le k \le K$. As with the algorithm in Figure 10, l_k $(1 \le k \le K)$ denotes the number of tuples read from R_k prior to the current sampling step and δ_k denotes the number of tuples read from R_k during the current sampling step. The quantity J denotes the set of index vectors $(i_1, i_2, \ldots, i_K)$ such that

(i) $l_k < i_k \le l_k + \delta_k$ for some $1 \le k \le K$; and

(ii) $f(L_{1,i_1}, L_{2,i_2}, \ldots, L_{K,i_K}) \ne 0$ or $g(L_{1,i_1}, L_{2,i_2}, \ldots, L_{K,i_K}) \ne 0$.

Just after the algorithm has executed for the nth time, the variable z is equal to $\tilde{T}_{n,1,1}(f,g)$, the variable $y^{(f)}$ is equal to $\tilde{T}_n(f)$, the variable $y^{(g)}$ is equal to $\tilde{T}_n(g)$, each variable $w_k/(L'_k - 1)$ is equal to $Z_n(f,g;k)$, each variable $q_{k,j}^{(f)}/L'_k$ is equal to $\tilde{T}_n(f;k,j)$, and each variable $q_{k,j}^{(g)}/L'_k$ is equal to $\tilde{T}_n(g;k,j)$.

C. The I/O Cost of Block Ripple Joins

In this section we consider the number of I/O's required to execute n (≥ 1) sampling steps of a K-relation ripple join with blocking factor α and aspect-ratio parameters $\beta_1, \beta_2, \ldots, \beta_K$. (More specifically, we consider a K-table query plan which consists of a left-deep tree of binary ripple joins, which in combination sweep out a sequence of K-dimensional hyper-rectangles with the appropriate aspect ratios.) As discussed in Section 5, we make

$n := n + \Delta n;$
for $k = 1, 2, \ldots, K$ {
  $l'_k := l_k + \delta_k;$
  $L_k := \prod_{j \neq k} l_j;$
  $L'_k := \prod_{j \neq k} l'_j;$
  $x_k^{(f)} := x_k^{(f)} L_k / L'_k;$
  $x_k^{(g)} := x_k^{(g)} L_k / L'_k;$
  for $l = l_k + 1, l_k + 2, \ldots, l_k + \delta_k$ {
    $q_{k,l}^{(f)} := 0;$
    $q_{k,l}^{(g)} := 0;$
  }
  if $n > 1$ {
    $w_k := w_k (L_k / L'_k)^2$
      $+ \delta_k (x_k^{(f)} / l_k)(x_k^{(g)} / l'_k);$
  }
}
for $(i_1, \ldots, i_K) \in J$ {
  for $k = 1, 2, \ldots, K$ {
    $j := i_k;$
    $d^{(f)} := q_{k,j}^{(f)} / L'_k;$
    $d^{(g)} := q_{k,j}^{(g)} / L'_k;$
    $x_k^{(f)} := x_k^{(f)} - d^{(f)};$
    $x_k^{(g)} := x_k^{(g)} - d^{(g)};$
    $w_k := w_k - \big(x_k^{(f)} - (l'_k - 1) d^{(f)}\big) / l'_k)$
      $\times \big(x_k^{(g)} - (l'_k - 1) d^{(g)}\big) / (l'_k - 1);$
    $q_{k,j}^{(f)} := q_{k,j}^{(f)} + f(L_{1,i_1}, \ldots, L_{K,i_K});$
    $q_{k,j}^{(g)} := q_{k,j}^{(g)} + g(L_{1,i_1}, \ldots, L_{K,i_K});$
    $d^{(f)} := q_{k,j}^{(f)} / L'_k;$
    $d^{(g)} := q_{k,j}^{(g)} / L'_k;$
    $w_k := w_k + \big(x_k^{(f)} - (l'_k - 1) d^{(f)}\big) / l'_k)$
      $\times \big(x_k^{(g)} - (l'_k - 1) d^{(g)}\big) / (l'_k - 1);$
    $x_k^{(f)} := x_k^{(f)} + d^{(f)};$
    $x_k^{(g)} := x_k^{(g)} + d^{(g)};$
  }
}
$y^{(f)} := x_1^{(f)} / l'_1;$
$y^{(g)} := x_1^{(g)} / l'_1;$
$z := \sum_{k=1}^{K} w_k / \big((l'_k / n)(l'_k - 1)\big);$

Figure 14: An algorithm for updating $\tilde{T}_{n,1,1}(f,g)$.

the conservative assumption that one I/O is incurred per tuple scanned. Because the exact expression for the I/O cost is extremely complicated, we provide a first-order approximation valid for large n.

We assume throughout that tuples are processed in a pipelined fashion. Under this assumption, the cost of sweeping out an $l_1 \times l_2 \times \cdots \times l_{k-1} \times \alpha \times l_{k+1} \times \cdots \times l_K$ hyper-rectangle in a nested-loops fashion with outermost relation R_k is given by

$$c_k(l_1, l_2, \ldots, l_{k-1}, l_{k+1}, \ldots, l_K) = \alpha + L/l_k,$$

where $L = l_1 l_2 \cdots l_K$. It then follows from the description of the ripple-join iterator in Appendix A that the cost $C_1^{(j)}(l_1, l_2, \ldots, l_K)$ of wrapping an existing $l_1 \times l_2 \times \cdots \times l_K$ hyper-rectangle by adding an α-tuple-thick layer along each of dimensions $j, j+1, \ldots, K$ is given by

$$C_1^{(j)}(l_1, l_2, \ldots, l_K) = \sum_{k=j}^{K} c_k(l_1, l_2, \ldots, l_{j-1}, l_j + \alpha, l_{j+1} + \alpha, \ldots l_{k-1} + \alpha, l_{k+1}, l_{k+2}, \ldots, l_K)$$

for $1 \le j \le K$, and the cost $C_b^{(j)}(l_1, l_2, \ldots, l_K)$ of wrapping the hyper-rectangle by adding a b layers, each α tuples thick, along each of dimensions $j, j+1, \ldots, K$ is given by

$$C_b^{(j)}(l_1, l_2, \ldots, l_K) = \sum_{m=0}^{b-1} C_1^{(j)}(l_1, l_2, \ldots, l_{j-1}, l_j + \alpha m, l_{j+1} + \alpha m, \ldots, l_K + \alpha m)$$

for $b \geq 1$. We can write

$$C_b^{(j)}(l_1, l_2, \ldots, l_K) = b \sum_{k=j}^{K} \frac{L}{l_k} + R, \tag{C.1}$$

where R is a sum of terms of the form $c l_1^{p_1} l_2^{p_2} \cdots l_1^{p_K}$ with nonnegative integer exponents $p_1, p_2, \ldots, p_K$ satisfying $p_1 + p_2 + \cdots + p_K \leq K - 2$. Now consider the nth sampling step of the ripple join, and assume without loss of generality that $\beta_1 \leq \beta_2 \leq \cdots \leq \beta_K$. During this step, we wrap a $\alpha\beta_1(n-1) \times \alpha\beta_2(n-1) \times \cdots \times \alpha\beta_K(n-1)$ hyper-rectangle by first adding β_1 α-tuple-thick layers along dimensions $1, 2, \ldots, K$, then adding $(\beta_2 - \beta_1)$ α-tuple-thick layers along dimensions $2, 3, \ldots, K$, and so forth. It follows from the representation in (C.1) that the cost $C_n(\beta_1, \beta_2, \ldots, \beta_K)$ of the nth sampling step can be written as

$$C_n(\beta_1, \beta_2, \ldots, \beta_K) = (\alpha n)^{K-1} B \sum_{j=1}^{K} \sum_{k=j}^{K} \frac{\beta_j - \beta_{j-1}}{\beta_k} + o(n^{K-1}),$$

where $\beta_0 = 0$ and $B = \beta_1 \beta_2 \cdots \beta_K$. Observe that

$$\sum_{j=1}^{K} \sum_{k=j}^{K} \frac{\beta_j - \beta_{j-1}}{\beta_k} = \sum_{k=1}^{K} \sum_{j=1}^{k} \frac{\beta_j - \beta_{j-1}}{\beta_k} = K,$$

so that $C_n(\beta_1, \beta_2, \ldots, \beta_K) = BK(\alpha n)^{K-1} + o(n^{K-1})$. Using the standard result that $\sum_{i=1}^{n} i^{k-1} = n^k/k + o(n^k)$ for $n, k \geq 1$, we find that the cumulative number of I/O's required for n sampling steps in the ripple join algorithm is $B\alpha^{K-1} n^K + o(n^K)$.

D. Approximate Solution of (5.1) and (5.2)

Note that when $K = 1$ the problem in (5.1) can be solved exactly: $\beta_1 = \min\bigl(c(q), m_k/\alpha\bigr)$. Our approach to approximate solution of the problem in (5.1) for $K \geq 2$ is to first solve the relaxed problem

$$\begin{aligned} &\text{minimize } \sum_{k=1}^{K} \frac{\zeta_k}{\alpha\beta_k} \\ &\text{such that} \\ &\qquad \beta_1 \beta_2 \cdots \beta_K \alpha^{K-1} \leq c, \\ &\qquad \beta_k \geq 0 \text{ for } 1 \leq k \leq K \end{aligned} \tag{D.1}$$

and then adjust the solution so that the remaining constraints in (5.1) are satisfied. To lighten notation, we have written ζ_k for $\zeta(v;k)$ and c for $c(q)$ in (D.1).

First suppose that each ζ_k in (D.1) is positive. Our approximate solution algorithm rests on the following result.

Lemma 1. *Suppose that $\zeta_k \geq 0$ for $1 \leq k \leq K$. Then the solution $\beta_1^*, \beta_2^*, \ldots, \beta_K^*$ to the minimization problem in* (D.1) *is given by*

$$\beta_k^* = \left(\frac{c}{\zeta_1\zeta_2\cdots\zeta_K}\right)^{1/K} \alpha^{(1-K)/K}\zeta_k \tag{D.2}$$

for $1 \leq k \leq K$.

Proof. Observe that an optimal solution will satisfy the first inequality constraint in (D.1) as a strict equality, so we can reformulate the problem in (D.1) as

$$\begin{aligned} &\text{minimize } h_{K-1}(\beta_1, \beta_2, \ldots, \beta_{K-1}; \alpha, c) \\ &\quad\text{such that } \beta_k \geq 0 \text{ for } 1 \leq k \leq K-1, \end{aligned}$$

where

$$h_{K-1}(\beta_1, \beta_2, \ldots, \beta_{K-1}; \alpha, c) = \frac{\zeta_1}{\beta_1} + \frac{\zeta_2}{\beta_2} + \cdots + \frac{\zeta_{K-1}}{\beta_{K-1}} + \frac{\zeta_K \alpha^{K-2}\beta_1\cdots\beta_{K-1}}{c}.$$

Given a solution $\beta_1^*, \beta_2^*, \ldots, \beta_{K-1}^*$ to the above problem, we then set

$$\beta_K^* = c/(\beta_1^*\beta_2^*\cdots\beta_{K-1}^*\alpha^{K-1}) \tag{D.3}$$

to obtain the complete solution to the original problem (D.1). We argue by induction on K. For $K = 2$, straightforward computations show that $h_1'(\beta_1^*; \alpha, c) = 0$, where β_1^* is given by (D.2) and h_1' denotes the first derivative of h_1. It is easy to verify that the function h_1 is convex over the set $[0, \infty)$ for any $\alpha, c > 0$, and hence β_1^* is the unique global minimizer of h_1. The desired result now follows for $K = 2$ since, by inspection, $c/(\alpha\beta_1^*) = \beta_2^*$, where β_2^* is given by (D.2). Now assume for induction that the result holds for a fixed value of K and consider the case of $K+1$ input relations. We find the optimal solution $\beta_1^*, \beta_2^*, \ldots, \beta_K^*$ by first minimizing h_K over $\beta_1, \beta_2, \ldots, \beta_{K-1}$ for fixed β_K and then minimizing the resulting function of β_K; we then compute β_{K+1}^* as in (D.3). For a fixed value of β_K, set

$$g(\beta_K) = \min_{\beta_1, \beta_2, \ldots, \beta_{K-1}} h_K(\beta_1, \beta_2, \ldots, \beta_K; \alpha, c).$$

Observe that

$$h_K(\beta_1, \beta_2, \ldots, \beta_K; \alpha, c) = h_{K-1}(\beta_1, \beta_2, \ldots, \beta_{K-1}; \alpha, \tilde{c}) + \frac{\zeta_K}{\alpha\beta_K},$$

where $\tilde{c} = \zeta_K c/(\alpha\beta_K\zeta_{K+1})$. It follows that, by induction, the values $\beta_1^0, \beta_2^0, \ldots, \beta_{K-1}^0$ that minimize $h(\beta_1, \beta_2, \ldots, \beta_K; \alpha, c)$ are given by

$$\beta_k^0 = \left(\frac{\tilde{c}}{\zeta_1\zeta_2\cdots\zeta_K}\right)^{1/K} \alpha^{(1-K)/K}\zeta_k = \left(\frac{c}{\zeta_1\zeta_2\cdots\zeta_{K-1}\zeta_{K+1}\beta_K}\right)^{1/K} \frac{\zeta_k}{\alpha} \tag{D.4}$$

for $1 \le k \le K-1$. Thus,

$$g(\beta_K) = h_K(\beta_1^0, \beta_2^0, \ldots, \beta_{K-1}^0, \beta_K) = K\left(\frac{\zeta_1\zeta_2\cdots\zeta_{K-1}\zeta_{K+1}\beta_K}{c}\right)^{1/K} + \frac{\zeta_K}{\alpha\beta_K}.$$

Although g is not in general convex, it is apparent from the form of g and its first derivative g' that g attains its minimum value over $[0, \infty)$ at the unique point

$$\beta_K^* = \left(\frac{c}{\zeta_1\zeta_2\cdots\zeta_{K+1}}\right)^{1/(K+1)} \alpha^{(1-(K+1))/(K+1)}\zeta_K.$$

Substituting this expression for β_K in (D.4) and then performing a calculation analogous to that in (D.3) yields the desired result. □

Thus, when each ζ_k in (D.1) is positive, set $\beta_k = \beta_k^*$ for $1 \le k \le K$, where each β_k^* is defined as in (D.2). Then execute the algorithm in Figure 15 to obtain the final values of $\beta_1, \beta_2, \ldots, \beta_K$. 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 (5.1). Observe, however, that this scaleup will cause the first constraint in (5.1) to be violated. To handle this problem, the algorithm then executes zero 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 the first constraint in (5.1) 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 inequality constraint is satisfied. Finally, each β_k is decreased further if necessary to ensure that the remaining constraints in (5.1) are satisfied. As discussed previously, the values of $\beta_1, \beta_2, \ldots, \beta_K$ produced by the algorithm can subsequently be scaled by their greatest common divisor.

Now suppose that $\zeta_k = 0$ for one or more values of k, and set

$$H_+ = \{\, 1 \le k \le K\colon\ \zeta_k > 0 \,\}.$$

For each relation R_k such that $k \notin H_+$, we can pretend, for purposes of online aggregation, that R_k contains exactly one tuple (since all the tuples in R_k behave identically in the

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

aggregation). Thus, we need only determine the aspect-ratio parameters for the relations $\{\, R_k\colon\ k \in H_+ \,\}$. This can be done by finding an approximate solution to the problem

$$\begin{aligned}
&\text{minimize } \sum_{k \in H_+} \frac{\zeta_k}{\alpha\beta_k} \\
&\text{such that} \\
&\qquad \alpha^{|H_+|} \prod_{k \in H_+} \beta_k \le c, \\
&\qquad 1 \le \beta_k \le m_k/\alpha \text{ for } k \in H_+, \\
&\qquad \beta_k \text{ integer for } k \in H_+
\end{aligned}$$

using the above method.

The problem in (5.2) can be solved in a similar way. We solve the relaxed problem

$$\begin{aligned}
&\text{minimize } \sum_{k=1}^{K} \frac{\zeta_k}{\beta_k} \\
&\text{such that} \\
&\qquad \beta_1 + \beta_2 + \cdots + \beta_K \le c(q), \\
&\qquad \beta_k \ge 0 \text{ for } 1 \le k \le K
\end{aligned} \tag{D.5}$$

and then adjust the solution so that the remaining constraints in (5.2) are satisfied, using the algorithm given in Figure 16. When each ζ_k is positive, an argument analogous to the proof of Lemma 1 shows that the solution to the relaxed problem is given by

$$\beta_k = \frac{c\sqrt{\zeta_k}}{\sqrt{\zeta_1} + \sqrt{\zeta_2} + \cdots + \sqrt{\zeta_K}}$$

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

Figure 16: Algorithm for modification of $\beta_1, \beta_2, \ldots, \beta_K$ in hash ripple join.

for $1 \le k \le K$. When one or more ζ_k are equal to 0, the computations are modified analogously to the case of block ripple joins. As with block ripple join, the values of $\beta_1, \beta_2, \ldots, \beta_K$ produced by the algorithm can subsequently be scaled by their greatest common divisor.

E. Optimizing Ripple Join Parameters: A General Approach

In this appendix we indicate a method for simultaneously setting both the aspect ratio parameters and the blocking factors for a block ripple join with input relations $R_1, R_2, \ldots, R_K$. We focus on the case in which *op* is equal to `SUM` or `COUNT` in (4.1). Suppose that tuples are retrieved a page at a time and that there are p_k (≥ 1) tuples per page of relation R_k for $1 \le k \le K$. For simplicity, suppose that the attribute values of a tuple are independent of the page on which the tuple resides; our techniques can be modified in a straightforward manner to handle the case in which tuples are clustered on pages (cf. [Haa97, Section 2.5]). We modify our notation slightly so that for $1 \le k \le K$ the relation size m_k and aspect-ratio parameter β_k are now expressed in units of pages rather than tuples. We allow the blocking factor to depend on the identity of the outermost relation, and denote this factor by α_k when the outermost relation is R_k. The blocking factor α_k is expressed in units of pages and, because of memory limitations, cannot exceed a specified constant α_k^*.

Under the above assumptions, an argument similar to that in Appendix C shows that the cumulative number C of I/O's required for n sampling steps of a block ripple join is given by

$$C = \alpha_1 \alpha_2 \cdots \alpha_K \beta_1 \beta_2 \cdots \beta_K n^K K^{-1} \left(\sum_{k=1}^{K} \frac{1}{\alpha_k} \right) + o(n^K).$$

On the other hand, the parameter $\tilde{\theta}_2(v)$ that determines the confidence-interval length after n sampling steps is now given by

$$\tilde{\theta}_2(v) = \sum_{k=1}^{K} \frac{\zeta(v;k)}{p_k \alpha_k \beta_k}, \tag{E.1}$$

where $\zeta(f;k)$ is defined by 4.6.

Analogously to Section 5, we wish to solve the optimization problem

$$\begin{aligned} &\text{minimize } \textstyle\sum_{k=1}^{K} \zeta(v;k)/(p_k \alpha_k \beta_k) \\ &\text{such that} \\ &\qquad \alpha_1 \cdots \alpha_K \beta_1 \cdots \beta_K \textstyle\sum_{k=1}^{K} (1/\alpha_k) \leq c(q)K, \\ &\qquad \alpha_k \leq \alpha_k^* \text{ for } 1 \leq k \leq K, \\ &\qquad 1 \leq \beta_k \leq m_k/\alpha_k \text{ for } 1 \leq k \leq K, \\ &\qquad \alpha_1, \ldots, \alpha_K, \beta_1, \ldots, \beta_K \text{ integer} \end{aligned} \tag{E.2}$$

for a fixed value $0 \leq q \leq 1$. Here $c(q)$ is a smooth decreasing function of q with $c(0) = m_1 m_2 \cdots m_K \sum_{k=1}^{K} (\alpha_k^* \wedge m_k)^{-1}$ and $c(1) = 1$. (Here $x \wedge y$ denotes the minimum of x and y.) When $q = 0$ (minimum interactivity), the optimal solution to (E.2) is given by $\alpha_k = \alpha_k^* \wedge m_k$ and $\beta_k = \lfloor m_k/(\alpha_k^* \wedge m_k) \rfloor$ for $1 \leq k \leq K$. When $q = 1$ (maximum interactivity) the only possible solution is $\alpha_k = \beta_k = 1$ for $1 \leq k \leq K$.

We conclude by sketching a heuristic method for finding an approximate solution to (E.2); given this solution, the actual setting of the parameters for the block ripple join proceeds almost exactly as described in Section 5. As in Appendix D, write $\zeta_k = \zeta(v;k)$ and $c = c(q)$. An inductive argument as in Appendix D shows that, for fixed values of $\alpha_1, \alpha_2, \ldots, \alpha_K$, the solution to the relaxed problem

$$\begin{aligned} &\text{minimize } \textstyle\sum_{k=1}^{K} \zeta_k/(p_k \alpha_k \beta_k) \\ &\text{such that} \\ &\qquad \beta_1 \cdots \beta_K \leq cK \big(\alpha_1 \cdots \alpha_K \sum_{k=1}^{K} (1/\alpha_k)\big)^{-1} \\ &\qquad \beta_k \geq 0 \text{ for } 1 \leq k \leq K \end{aligned}$$

is given by

$$\beta_k = \beta_k(\alpha_1, \alpha_2, \ldots, \alpha_K) = \left(\frac{cK p_1 p_2 \cdots p_K}{\zeta_1 \zeta_2 \cdots \zeta_K} \right)^{1/K} \left(\sum_{k=1}^{K} \frac{1}{\alpha_k} \right)^{-1/K} \frac{\zeta_k}{p_k \alpha_k} \tag{E.3}$$

for $1 \leq k \leq K$, and the corresponding optimal value v_{opt} of the objective function is

$$v_{\text{opt}} = K \left(\frac{\zeta_1 \zeta_2 \cdots \zeta_K}{cK p_1 p_2 \cdots p_K} \right)^{1/K} \left(\sum_{k=1}^{K} \frac{1}{\alpha_k} \right)^{1/K}.$$

This suggests the auxiliary problem

$$\begin{aligned}
&\text{minimize } \textstyle\sum_{k=1}^{K}(1/\alpha_k)\\
&\text{such that}\\
&\qquad 1 \le \beta_k(\alpha_1, \alpha_2, \dots, \alpha_K) \le m_k/\alpha_k,\\
&\qquad 1 \le \alpha_k \le \alpha_k^* \wedge m_k,\\
&\qquad \alpha_1, \dots, \alpha_K \text{ integer.}
\end{aligned}$$

To find an approximate solution to this problem, we consider the following relaxed version:

$$\begin{aligned}
&\text{minimize } \textstyle\sum_{k=1}^{K}(1/\alpha_k)\\
&\text{such that } \beta_k(\alpha_1, \alpha_2, \dots, \alpha_K) \ge 1,
\end{aligned} \tag{E.4}$$

It can be shown that the solution to (E.4) is given by

$$\alpha_k = \left(\frac{cKp_1p_2\cdots p_K}{\zeta_1\zeta_2\cdots\zeta_K}\right)^{1/(K-1)} \left(\sum_{k=1}^{K}\frac{p_k}{\zeta_k}\right)^{-1/(K-1)} \frac{\zeta_k}{p_k} \tag{E.5}$$

for $1 \le k \le K$. Collecting the above results, we obtain the following heuristic solution method:

1. Compute $\alpha_1, \alpha_2, \dots, \alpha_K$ according to (E.5).
2. Set $\theta = \min(\alpha_1, \alpha_2, \dots, \alpha_K)$.
3. Set $\alpha_k = \min(\lfloor \alpha_k/\theta \rfloor, \alpha_k^*, m_k)$ for $k = 1, 2, \dots, K$.
4. Compute $\beta_1, \beta_2, \dots, \beta_K$ according to (E.3).
5. Using an algorithm as in Figure 15, adjust $\beta_1, \beta_2, \dots, \beta_K$ to ensure that the first and third constraints in (E.2) are satisfied.