

The Sort-Merge-Shrink Join

CHRISTOPHER JERMAINE, ALIN DOBRA, SUBRAMANIAN ARUMUGAM,
SHANTANU JOSHI, and ABHIJIT POL

University of Florida, Gainesville

One of the most common operations in analytic query processing is the application of an aggregate function to the result of a relational join. We describe an algorithm called the *Sort-Merge-Shrink* (SMS) *Join* for computing the answer to such a query over large, disk-based input tables. The key innovation of the SMS join is that if the input data are clustered in a statistically random fashion on disk, then at all times, the join provides an online, statistical estimator for the eventual answer to the query as well as probabilistic confidence bounds. Thus, a user can monitor the progress of the join throughout its execution and stop the join when satisfied with the estimate's accuracy or run the algorithm to completion with a total time requirement that is not much longer than that of other common join algorithms. This contrasts with other online join algorithms, which either do not offer such statistical guarantees or can only offer guarantees so long as the input data can fit into main memory.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems—*Query processing*;
G.3 [Probability and Statistics]: —*Probabilistic algorithms*

General Terms: Algorithms

Additional Key Words and Phrases: Online algorithms, OLAP, nonparametric statistics

1. INTRODUCTION

One promising approach to data management for analytic processing is *online aggregation* [Hellerstein et al. 1997, 1999] (OLA). In OLA, the database system tries to quickly discover enough information to immediately give an approximate answer to a query over an aggregate function such as COUNT, SUM, AVERAGE, and STD_DEV. As more information is discovered, the estimate is incrementally improved. A primary goal of any system performing OLA is providing some sort of statistical guarantees on the result quality, usually in the form of statistical *confidence bounds* [Cochran 1977]. For example, at a given moment,

Material in this article is based on work supported by the National Science Foundation Under Grant No. 0347408.

A preliminary version of this paper appeared as “A Disk-Based Join With Probabilistic Guarantees” in the SIGMOD 2005 Conference.

Authors' address: CISE Department, University of Florida, Gainesville, FL; email: {cjermain, adobra, sa2, ssjoshi, apol}@cise.ufl.edu.

Permission to make digital or hard copies part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 0362-5915/06/1200-1382 \$5.00

the system might say, “0.32% of the banking transactions were fraudulent, with $\pm 0.027\%$ accuracy at 95% confidence.” A few seconds later, the accuracy guarantee might be $\pm 0.013\%$. If the user is satisfied with the accuracy, or determines from the initial results that the query is uninteresting, then computation can be terminated by a mouse click, saving valuable human time and computer resources.

In practice, one of the most significant technical difficulties in OLA is relational join processing. In fact, join processing is even more difficult in OLA than in traditional transaction processing systems. In addition to being computationally efficient, OLA join algorithms must have meaningful and statistically-justified confidence bounds. The current state-of-the-art for join processing in OLA is the *ripple join* family of algorithms developed by Haas and Hellerstein [1999]. However, there is one critical drawback with the ripple join and others that have been proposed for OLA. All existing join algorithms for OLA (including the ripple join) are unable to provide the user with statistically meaningful confidence bounds as well as efficiency from start-up through completion, *if the total data size is too large to fit in main memory*. This is unfortunate, because a ripple join may run out of memory in a few seconds, but a sort-merge join or hybrid hash join [Shapiro 1986] may require hours to complete over a very large database. Using current technology, if the user is unsatisfied with the ripple join’s accuracy, the only option is to wait until an exact answer can be computed.

It turns out to be exceedingly difficult to design disk-based join algorithms that are amenable to the development of statistical guarantees. The fundamental problem is that OLA join algorithms must rely on randomness to achieve statistical guarantees on accuracy. However, randomness is largely incompatible with efficient disk-based join processing. Efficient disk-based join algorithms (such as the sort-merge join and hybrid hash join) rely on careful organization of tuples into blocks so that when a block is loaded into memory, all of its tuples are used for join processing before the block is unloaded from memory. This careful organization of tuples is the antithesis of randomness and makes statistical analysis difficult. This is the reason why, for example, the scalable version of the ripple join algorithm by Luo et al. [2002] can no longer maintain any statistical guarantees as soon as it runs out of memory.

Previously, Haas and Hellerstein have asserted that the lack of an efficient, disk-based OLA join algorithm is not a debilitating problem, because often the user is sufficiently happy with the accuracy of the estimate that the join is stopped long before the buffer memory is exhausted [Hellerstein et al. 1997]. While this argument is often reasonable, there are many cases where convergence to an answer will be very slow, and buffer memory could be exhausted long before a sufficiently accurate answer has been computed. Convergence can be slow under a variety of conditions, including:

- if the join has high selectivity;
- if a select-project-join query has a relational selection predicate with high selectivity;

- if the attributes queried over have high skew;
- if the query contains a `GROUP BY` clause with a large number of groups or if the group cardinalities are skewed; or
- if the database tuples or key values are large (e.g., long character strings) and so they quickly consume the available main memory.

1.1 Our Contributions

In this article, we develop a new join algorithm for OLA that is specifically designed to be amenable to statistical analysis at the same time that it provides out-of-core efficiency. Our algorithm is called the *SMS join* (which stands for *Sort-Merge-Shrink* join). At a high level, the SMS join is best viewed as a variation of the sort-merge join [Shapiro 1986] or, more specifically, the Progressive Merge Join [Dittrich et al. 2002] with the addition of significant statistical computations that allow the final result of the query to be guessed at throughout execution. The SMS join represents a contribution to the state-of-the-art in several ways.

- The SMS join is totally disk-based, yet it maintains statistical confidence bounds from start-up through completion.
- Because the first phase of the SMS join consists of a series of hashed ripple joins, the SMS join is essentially a generalization of the ripple join. If a satisfactory answer can be computed in the few seconds before the join runs out of core memory, the SMS join is equivalent to a ripple join.
- Despite the fact that it maintains statistical guarantees, the SMS join is competitive with traditional join algorithms such as the sort-merge join, even with the extra computation required to maintain statistical estimates throughout the process. Our prototype implementation of the SMS join is not significantly slower than our own sort-merge join implementation. Our experiments show that, if the SMS join is carefully implemented, the join is only around 10% slower than the traditional sort-merge join. Given this, we argue that if the SMS join is used to answer aggregate queries, online statistical confidence bounds can be maintained virtually for free with the only additional cost being a small performance hit required to complete the entire join compared to a more traditional algorithm.

1.2 Paper Organization

The remainder of the article is organized as follows. Section 2 gives some background on the problem of developing out-of-core joins for OLA. Section 3 gives an overview of the SMS join, and the subsequent three sections describe the details of the algorithm. Section 7 considers how the SMS join can be implemented so that it runs with high efficiency. Our benchmarking is described in Section 8, related work is described in Section 9, and the article is concluded in Section 10. The article's appendices discuss statistical considerations, including how the SMS join can be applied if data are not clustered randomly on disk.

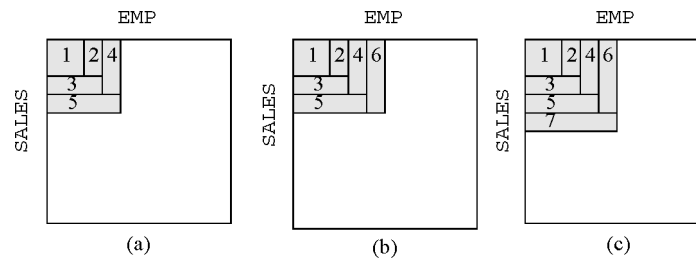


Fig. 1. Performing sampling steps during a ripple join.

2. DISK-BASED JOINS FOR OLA: WHY ARE THEY SO DIFFICULT?

2.1 A Review of the Ripple Join Algorithm

We begin with a short review of the ripple join family of algorithms, which constitute the state-of-the-art in join algorithms for OLA [Haas and Hellerstein 1999]. We will describe the ripple join in the context of answering the query: “Find the total sales per office for the years 1980-2000” over the database tables EMP (OFFICE, START, END, NAME) and SALES (YEAR, EMP NAME, TOT_SALES). The first table describes the time period that an employee is associated with an office, and the second table gives the total sales attributed to an employee per year. If an employee can only be associated with one office at a time, SQL for this query is given as:

```
SELECT SUM (s.TOT_SALES), e.OFFICE, s.YEAR
FROM EMP e, SALES s
WHERE e.NAME = s.EMP_NAME AND s.YEAR BETWEEN e.START
      AND e.END AND s.YEAR BETWEEN 1980 AND 2000
GROUP BY e.OFFICE, s.YEAR
```

To answer this query, a ripple join will scan the two input relations EMP and SALES in parallel, using a series of operations known as *sampling steps*. At the beginning of each sampling step, a new set of tuples of size n_{EMP} is loaded into memory from the input relation EMP. These new tuples from EMP are joined with all of the tuples that have been encountered thus far from the second input relation, SALES. This is depicted as the addition of region 6 to Figure 1(b) from Figure 1(a). Next, the roles of EMP and SALES relations are swapped: n_{SALES} new input tuples are retrieved from SALES and joined with all existing tuples from EMP (shown as the addition of region 7 in Figure 1(c)). Finally, both the estimate for the answer to the query and the statistical confidence interval for the estimate’s accuracy are updated to reflect the new tuples from both relations and then output to the user via an update to the graphical user interface. The estimate and bounds are computed by associating a normally distributed random variable N with the ripple join; on expectation, the ripple join will give a correct estimate for the query, and the error of the estimate is characterized by the variance of N , denoted by σ^2 . A key requirement of the ripple join is that all input relations must be clustered randomly on disk, so that there is absolutely no correlation between the ordering of the tuples on disk

and the contents of the tuples. Even the slightest correlation can invalidate the statistical properties of the join, leading to inaccurate estimates and confidence bounds.

The ripple join is actually a family of algorithms and not a single algorithm, since the technique does not specify exactly how the new tuples read during a sampling step are to be joined with existing tuples. This can be done using an index on one or both of the relations, or in the style of a nested-loops join, or using a main memory hash table to store the processed tuples.

In their work, Haas and Hellerstein [1999] showed that the hashed version of the ripple join generally supports the fastest convergence to an accurate answer. To perform a hashed ripple join, a single main memory hash table is used to hold tuples from both relations. When a new tuple e is read from input relation EMP, it is added to the table by hashing the key(s) that the input relations are joined on. Any tuple s from the other input relation SALES which matches e will be in the same bucket, and the tuples can immediately be joined.

Haas and Hellerstein's work demonstrated that other types of ripple joins are usually less efficient than the hashed ripple join. A *nested-loops ripple join* gives very slow convergence and is expectedly even slower to run to completion than the traditional (and often very slow) nested-loops join. The reason for this is that since the ripple join is symmetric, each input relation must serve as the inner relation in turn, leading to redundant computation in order to maintain statistics and confidence intervals, compared with even a traditional nested-loops join. An *indexed ripple join* can be faster than a nested-loops ripple join but, since random clustering of the input relations is an absolute requirement for estimation accuracy, the index used must be a secondary index, and an indexed ripple join is essentially nothing more than a slower version of the traditional indexed nested-loops join. Thus, it still requires at least one or two disk seeks per input tuple to perform an index lookup to join each tuple. Even using modern hardware, one would expect a processing rate no faster than a few thousand tuples per-second per-disk under such a regime.

2.2 OLA and Out-Of-Core Joins

While it is very fast in main memory, the hashed ripple join becomes unusable as soon as the central hash table becomes too large to fit into main memory. The problem is that, when a large fraction of the hash table must be off-loaded to disk, each additional tuple encountered from the input file will require one or two random disk head movements in order to add it to the hash table, join it with any matching previously-encountered tuples, and then update the output statistics. At a few milliseconds per seek using modern hardware, this equates at most to only a few thousand tuples processed per-second per-disk.

Furthermore, designing a join algorithm that can operate efficiently from disk as well as provide tight, meaningful confidence bounds is a difficult problem. Consider a hybrid hash join [Shapiro 1986] over a query of the form:

```
SELECT expression ( $e$ ,  $s$ )
FROM EMP  $e$ , SALES  $s$ 
WHERE predicate ( $e$ ,  $s$ )
```

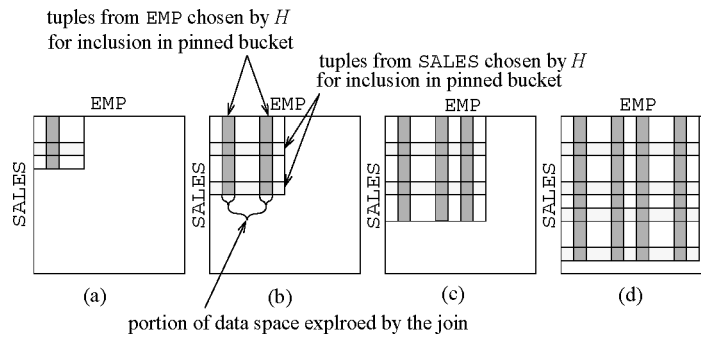


Fig. 2. Progression of the first pass of the symmetric version of the hybrid hash join.

where *predicate* is a conjunction containing the clause $e.k = s.k$. A hybrid hash join would operate by first scanning the first input relation EMP, hashing all of EMP's tuples on the attribute EMP. k to a set of buckets that are written out to disk as they are constructed, except for a single, 'special' bucket that is pinned in main memory. After hashing EMP, the second relation SALES is then scanned and hashed on the attribute SALES. k . Any tuples from SALES that hash to the special pinned bucket are joined immediately, and the results are output online. All other tuples are written to disk with the other tuples from their respective buckets. In a second pass over the buckets from both EMP and SALES, all matching buckets are joined, and resulting tuples are output as they are produced.

One could very easily imagine performing a ripple-join-like symmetric variation on the hybrid hash join where the first passes through both SALES and EMP are performed concurrently, and all additions to the special pinned bucket are joined immediately and used to produce an estimate to the query answer. If a hash function is used which truly picks a random subset of the tuples from each relation to fall in the special pinned buckets, then the resulting join is amenable to statistical analysis. As the relation EMP is scanned, a random set of tuples from EMP will be chosen by the hash function. Because any matching tuples from SALES will be pinned in memory, any tuples from EMP chosen by the hash function can immediately be joined with all tuples encountered thus far from SALES. If the join attribute is a candidate key of the EMP relation, the set of tuples chosen thus far from EMP is a true random sample of EMP, and this set is independent from the set of all tuples that have been read from SALES. Thus, a statistical analysis very similar to the one used by Haas [1997], Haas et al. [1996] and Haas and Hellerstein [1999] can be used to describe the accuracy of the resulting estimator. The progression of the join with enough memory to buffer four tuples from each input relation is shown in Figure 2.

However, there are several serious problems with this particular online join, including the following.

—*Early on, the resulting estimator will have less accuracy than a ripple join.* This is particularly worrisome, because if OLA is used for exploratory data analysis, it is expected that many queries will be aborted early on. Thus, a fast and accurate estimate is essential. In the case of the symmetric hybrid

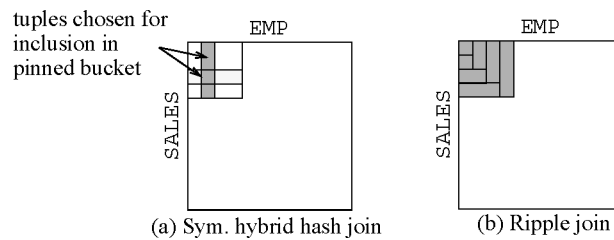


Fig. 3. Comparison of a symmetric hybrid hash join and a ripple join with enough memory to buffer four tuples from each input relation.

hash join, the issue is that the pinned bucket containing tuples from EMP will not fill until EMP has been entirely scanned during the first pass of the join. Contrast this with a ripple join: if we have enough memory to buffer n_{EMP} tuples from EMP, then after n_{EMP} tuples have been read from EMP, a ripple join will use all n_{EMP} tuples immediately to estimate the final answer to the join. On the other hand, after scanning n_{EMP} tuples, the symmetric hybrid hash join would use only around $n_{EMP}/|EMP|$ tuples to estimate the final answer. A comparison between the two algorithms after four tuples from each input relation have been read is shown in Figure 3. While the symmetric, hybrid hash join has ignored three of the four tuples from each input relation, the ripple join has fully processed all four.

- If the join attribute is not a candidate key of EMP, then statistical analysis of the join is difficult.* The problem in this case is that the tuples chosen by the hash function will not be a random sample of the tuples from EMP since the function will choose all tuples with a given key value k . The statistical properties of the join under such circumstances are exceedingly difficult to analyze.
- Extending the statistical analysis of the symmetric hybrid hash join to the second phase is difficult as well.* From a statistical perspective, the issue is that each pair of buckets from EMP and SALES joined during the second phase are not independent since they contain tuples hashed to matching buckets. Taking into account this correlation is difficult. Furthermore, the tuples from each on-disk bucket from SALES have already been used to compute the estimator associated with the growing region of Figure 2 so each join bucket is not independent of the initial estimator computed during the hash phase.

3. THE SMS JOIN: AN OVERVIEW

The difficulties described in the previous section illustrate why developing out-of-core joins with statistical guarantees is a daunting problem. We now give a brief overview of the SMS join, which is designed specifically to maintain statistically meaningful confidence bounds in a disk-based environment. The SMS join is a *generalization* of the ripple join, in the sense that if it is used on a small dataset or if it is terminated before memory has been exhausted, then it is identical to the ripple join. However, the SMS join continues to provide confidence bounds throughout execution, even when it operates from disk. Intuitively, the

SMS join is best characterized as a disk-based sort-merge join with several important characteristics that differentiate it from the classic sort-merge join.

3.1 Preliminaries

In our description of the SMS join, we assume two input relations, EMP and SALES (as described in Section 2.1) which are joined using the following generic query:

```
SELECT SUM  $f(e, s)$ 
FROM EMP as  $e$ , SALES as  $s$ 
```

We restrict our discussion to the case of two input relations, though just like a sort-merge join, the SMS join can handle additional input relations as long as the same sort order can be used to join all of them. Just as in the case of the ripple join, in order to provide our statistical confidence bounds, we will require that input relations EMP and SALES be clustered in a statistically random fashion on disk¹. One way to achieve the desired randomization in a scalable fashion is to associate a pseudo-random number with each tuple. Then, a scalable disk-based sorting algorithm (such as a two-phase, multiway merge sort) is used on the tuples to order them based upon the associated pseudo-random numbers. Once the tuples are randomized, the numbers can then be discarded. The function f can be any function over a pair of tuples from the two input relations and can encode arbitrary relational selection predicates or join conditions over the input tuples. We assume that f evaluates to zero unless a conjunction of boolean expressions over the two input tuples evaluates to true and that one of these boolean expressions is an equality check of the form $(e.k = s.k)$, so that a classical hash join or sort-merge join could be used over the attribute k . We subsequently refer to the attribute k as the *key*. In our discussion, we also assume that the aggregate function evaluated is SUM; AVERAGE or ST_DEV queries can be handled by treating the query as an expression over multiple queries that are evaluated concurrently. Since the associated confidence bounds must simultaneously take into account the potential for error in each of the constituent queries, one of several methods must be chosen to combine the estimates. One method suggested previously is the use of Bonferroni's inequality [Haas et al. 1996].

In general, we will assume that B blocks of main memory are available for buffering input tuples while computing the join. For simplicity, we assume that additional main memory (in addition to those B blocks) is available for buffering output tuples and storing the required statistics; see Section 6.4 for a detailed discussion of the additional memory required by the SMS join compared to the sort-merge join. We will use the notations $|EMP|$ and $|SALES|$ to denote the size of the EMP and SALES input relations measured by the number of tuples, respectively. $|EMP|_b$ and $|SALES|_b$ denote the number of blocks used to store these relations.

¹If such an ordering is not available or is not practical, the SMS join can still be used; see Appendix B for a discussion.

3.2 Three Phases of the SMS Join

Given these preliminaries, the SMS join has three phases.

- (1) The *sort phase* of the SMS join corresponds to a modified version of the sort phase of a sort-merge join. At a high level, the sort phase of the SMS join closely resembles the sort phase of the Progressive Merge Join [Dittrich et al. 2002] in that all of the data currently in memory from both relations are used to search for early output results. The process of reading in and sorting each pair of runs from EMP and SALES is treated as a series of hashed ripple joins, each of which is used to provide a separate estimate for the final result of the join. Using the techniques of Haas and Hellerstein, it is possible to characterize the estimate corresponding to the i th ripple join as a random variable N_i whose distribution becomes normal or Gaussian with increasing sample size. As described subsequently, all of the estimators can be combined to form a single running estimator N , where on expectation N provides the correct estimate for the query result. The sort phase of the SMS join is illustrated in Figure 4(a) through (d) and Figure 5(e).
- (2) The *merge phase* of the SMS join corresponds to the merge phase of a sort-merge join. Just as in a sort-merge join, the merge phase pulls tuples from the runs produced during the sort phase and joins them using a lexicographic ordering. The key difference between the SMS join and the sort-merge join or the Progressive Merge Join is that in the SMS join, the merge phase runs concurrently with (and sends information to) the shrinking phase of the join.
- (3) The *shrinking phase* of the SMS join is performed concurrently with the merge phase. Conceptually, the merge phase of the SMS join divides the data space into four regions, shown in Figure 5(f) through (h). If $\overline{\text{EMP}}$ and $\overline{\text{SALES}}$ refer to the subsets of EMP and SALES that have been merged thus far, then as depicted in Figure 5, the four regions of the data space are $(\text{EMP}-\overline{\text{EMP}}) \bowtie (\text{SALES}-\overline{\text{SALES}})$, $(\text{EMP}-\overline{\text{EMP}}) \bowtie \overline{\text{SALES}}$, $\overline{\text{EMP}} \bowtie (\text{SALES}-\overline{\text{SALES}})$, and $\overline{\text{EMP}} \bowtie \overline{\text{SALES}}$. Assuming that the query to be evaluated is SUM or COUNT, then the final answer to our query is simply the sum over each of these four regions.

Just as in a classical sort-merge join, the merge phase of the SMS join allows us to compute the value of $\text{SUM } f(e, s)$ for the latter three joins exactly (note that, in the case of $(\text{EMP}-\overline{\text{EMP}}) \bowtie \overline{\text{SALES}}$ and $\overline{\text{EMP}} \bowtie (\text{SALES}-\overline{\text{SALES}})$, this value will always be zero). However, in the classical sort-merge join, the region corresponding to $(\text{EMP}-\overline{\text{EMP}}) \bowtie (\text{SALES}-\overline{\text{SALES}})$ in Figure 5 would be ignored. Because the goal of the SMS join is to produce a running estimate for the eventual query answer, the value for $\text{SUM } f(e, s)$ over $(\text{EMP}-\overline{\text{EMP}}) \bowtie (\text{SALES}-\overline{\text{SALES}})$ will be continually estimated by the shrinking phase of the join algorithm.

The specific task of the shrinking phase is handling the removal of tuples from the sort phase ripple join estimators, and updating the estimator N corresponding to the value of $\text{SUM } f(e, s)$ over the join $(\text{EMP}-\overline{\text{EMP}}) \bowtie (\text{SALES}-\overline{\text{SALES}})$. Tuples that are consumed by the merge phase must be removed from each N_i

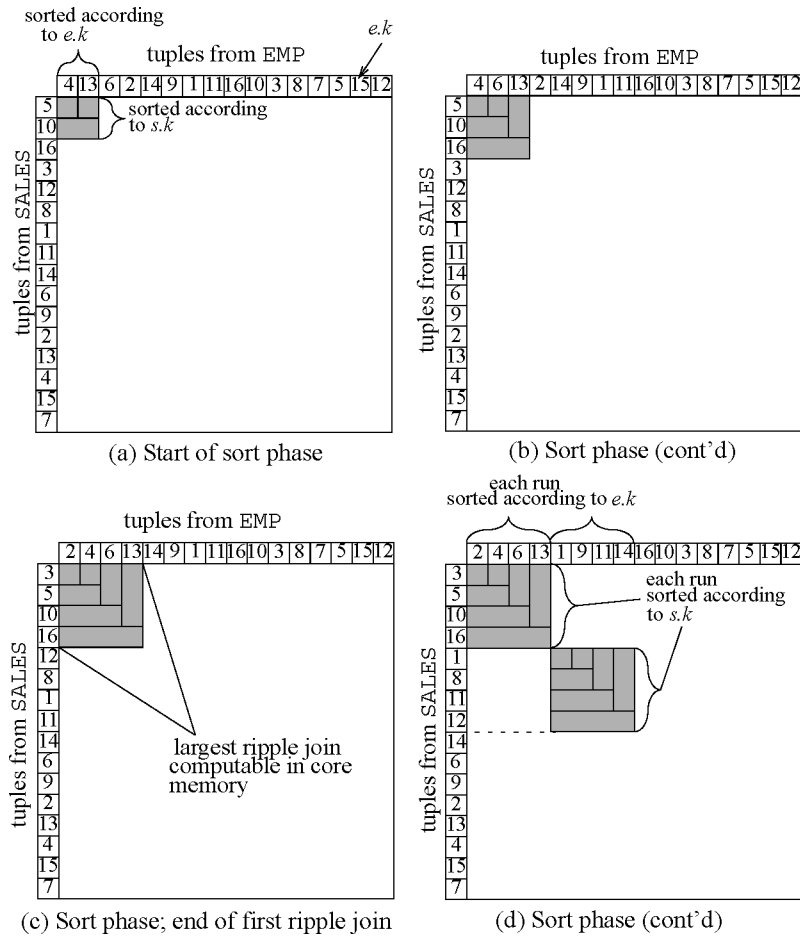


Fig. 4. Outline of an SMS join, assuming enough main memory to buffer four tuples from each input relation. The remainder of the join is depicted in Figure 5.

that contributes to N so that we ensure that each of the four regions depicted in Figure 5(f) through (h) remain disjoint. As long as this is the case, the value of $\text{SUM } f(e, s)$ over each of the four regions can simply be summed up to produce a single estimate for the eventual query result.

We now describe each of the phases of the SMS join in more detail.

4. THE SORT PHASE

The *sort* phase of the SMS join is very similar to the first phase of a sort-merge join. The phase is broken into a series of steps where, at each step, a subset of each input relation of total size small enough to fit in main memory is read from disk, sorted, and then written back to disk. Despite the similarity, some changes will be needed to provide an online estimate for the eventual answer to our query.

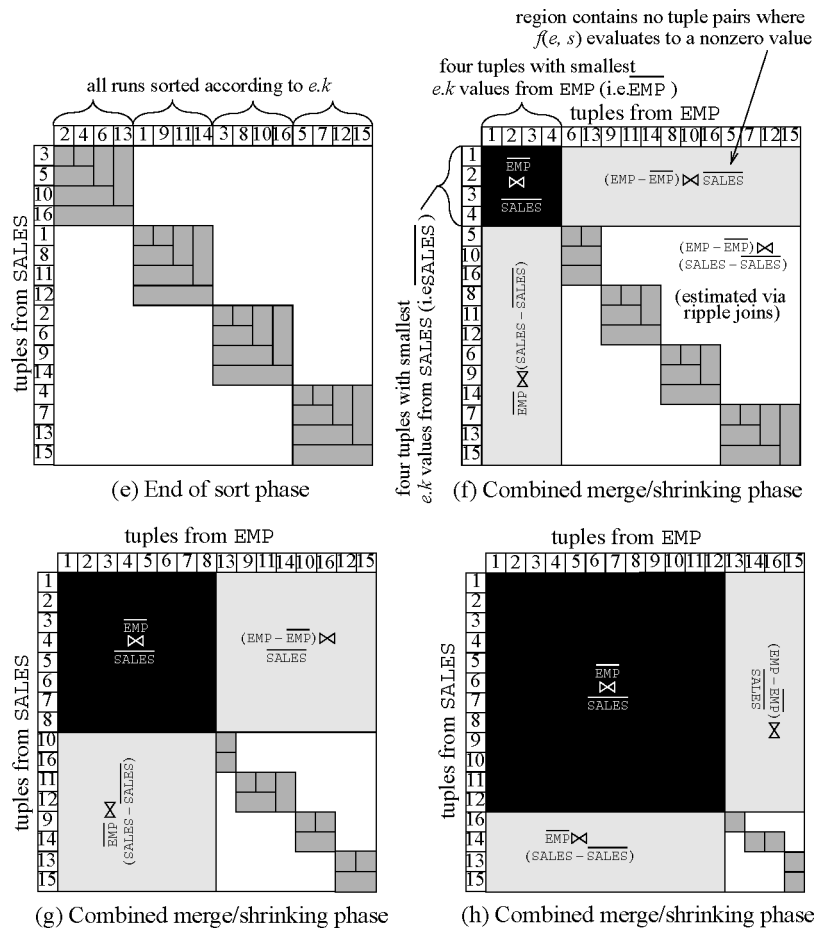


Fig. 5. Outline of an SMS join, continued.

4.1 The Basic Algorithm

Pseudocode for the sort phase of the SMS join is given in Algorithm 1. During each step of the sort phase, a subset of the blocks from each input relation is first read in parallel and joined in memory using a hashed ripple join. After the blocks from each relation have consumed all of the available buffer memory, the tuples of each subset (called a run) are then sorted on the join attribute(s). The operation of the sort phase is visualized in Figure 4(a) through (d) and Figure 5(e).

4.2 Statistical Analysis

We now address the problem of providing online estimates for the eventual answer to the query during the sort phase of the SMS join, along with statistical guarantees on the accuracy of those estimates.

Algorithm 1. Sort Phase of the SMS Join

Input: Amount of available main memory B , parameter to control the interleaving of reads and writes m

- (1) Let $m_{EMP} = (B \times |EMP|_b) / (|EMP|_b + |SALES|_b)$
 - (2) Let $m_{SALES} = (B \times |SALES|_b) / (|EMP|_b + |SALES|_b)$
 - (3) Let $n_{EMP} = m_{EMP} / m$, $n_{SALES} = m_{SALES} / m$
 - (4) Perform a ripple join of the first m_{EMP} and m_{SALES} blocks from EMP and SALES, respectively; these blocks will be EMP_1 and $SALES_1$
 - (5) Sort EMP_1 and $SALES_1$ on $EMP.k$ and $SALES.k$
 - (6) For int $i = 1$ to $(|EMP|_b + |SALES|_b) / B - 1$
 - (7) While $EMP_i \neq \phi$
 - (8) Write n_{EMP} blocks from EMP_i to disk
 - (9) Write n_{SALES} blocks from $SALES_i$ to disk
 - (10) Read n_{EMP} blocks from EMP into EMP_{i+1}
 - (11) Read n_{SALES} blocks from SALES into $SALES_{i+1}$
 - (12) Join the new blocks from SALES with EMP_{i+1}
 - (13) Join the new blocks from EMP with $SALES_{i+1}$
 - (14) Sort EMP_{i+1} and $SALES_{i+1}$
-

4.2.1 *Overview.* It is possible to characterize the estimate provided by each of the individual sort phase estimators by any of a number of methods, including making use of the analysis provided by Haas and Hellerstein [1999] as a starting point. However, the problem we face is significantly different than that faced by Haas and Hellerstein: we have many sort phase ripple join estimators, not just a single one. The question we must tackle when analyzing the SMS join is: How can these estimates be combined in a meaningful way so as to produce a single estimator that is better than any of its constituent parts?

The basic tactic we use is as follows. First, let N_i be a random variable corresponding to the estimate provided by the i th sort phase ripple join. Note that each individual N_i is identical to Haas and Hellerstein's ripple join estimator and is computed in exactly the same way. However, to extend our analysis to many ripple joins used by the SMS join, we associate a weight w_i with each individual estimate, subject to the constraint that $\sum_i^n w_i = 1$. Assuming a total of n ripple joins in the sort phase, the random variable $N = \sum_i^n N_i w_i$ is then itself an unbiased estimator for the eventual query answer. By carefully choosing the weights associated with the various ripple joins, we can maximize the accuracy of the estimator. The remainder of this section focuses on the three-part problem of first characterizing the accuracy of N , then tuning the weights within N so as to maximize its accuracy, and, finally, estimating the accuracy of N on-the-fly.

4.2.2 *Characterizing the Accuracy of N .* We weight each ripple join because a careful choice of the various weights will tend to increase the accuracy of the estimate provided by N . A new ripple join that has just begun is likely to have a large degree of inaccuracy, and should thus be given less weight than the sort phase ripple joins that have completed. In the remainder of the article, we use the standard statistical technique of measuring the accuracy of an unbiased estimator by measuring its *variance*. Given a set of weights, the general

formula for the variance of N is:

$$\text{Var}(N) = \sum_i^n w_i^2 \text{Var}(N_i) + \sum_i^n \sum_{j \neq i}^n w_i w_j \text{Cov}(N_i, N_j). \quad (1)$$

In Eq. (1), the function *Cov* refers to the *covariance* of the two ripple join estimators. The covariance appears in the formula for the variance of N because the various ripple join estimators are not independent: if a tuple is used during the i th ripple join, then it cannot be used during the j th ripple join.

Unfortunately, deriving formulas for $\text{Var}(N_i)$ and $\text{Cov}(N_i, N_j)$ is a difficult task. We could use Haas and Hellerstein's [1999] derivation for a single ripple join as the basis for our derivation of these terms, but this would present two primary difficulties.

- (1) First, Haas and Hellerstein's derivation approximates sampling without replacement using sampling with replacement. As a result, in their analysis, all samples can be assumed to be independent. In the application domain imagined for the original ripple join, this is reasonable since it was assumed that the join will be terminated after only a rough estimate of the eventual answer has been obtained. This means that the join probably will end before a significant fraction of either input relation has been processed. In such a situation, the difference between sampling with replacement and sampling without replacement is negligible. In our own application where each join may cover a significant fraction of the database (and all tuples will eventually be part of exactly one ripple join), such an approximation is very hard to justify.
- (2) Second, the Haas and Hellerstein's derivation was nontrivial to say the least, and can be traced through a series of papers by Peter Haas and his co-authors [Haas et al. 1996, 1999; Haas 1997]. Extending this analysis to the more complicated estimator described in this article seems like a very daunting task, specifically because the covariance between the various ripple join estimators must be considered as well.

As a result, we give exact permutational formulas for these values in Appendix A (the results are given as Theorem A.1, Corollary A.2, Theorem A.3, and Corollary A.4). The permutational analysis given in the article's electronic appendix is arguably simpler than the without-replacement analysis that led to a characterization of the original ripple join; this is the main reason that it is even feasible to give an analysis of the more complicated estimator described in this article. While the exact formulas and their derivation are not important to the exposition presented here, the salient properties of the formulas are the following.

- (1) They give an exact characterization of the accuracy of our estimators for any data sets and queries of the type considered in this article.
- (2) Given a few sufficient statistics about the datasets used by the join (or estimates for the value of these statistics), it is an easy matter to use the formulas to compute the variance of N , which will be described in Section 4.2.4.

4.2.3 *Tuning the Weights for Maximum Accuracy.* As discussed, the value of N and the variance of N both depend on the weights that are chosen for each of the constituent ripple joins. To maximize the accuracy of the estimate provided by N , it is necessary to compute the various weights so as to minimize the variance of N .

Performing this minimization is a straightforward task using basic calculus. Before we begin, we make note of two important observations that will guide the calculation. Assuming that we are currently computing the k th sort phase ripple join where $1 \leq k \leq n$, then the following observations simplify the problem.

- (1) First, we know that $\text{Var}(N_i) = \text{Var}(N_k)$ for all $1 \leq i < k$, so it must be the case that $w_i = w_k$ for all $1 \leq i < k$ in any optimal weighting scheme. This is because (by definition) all of the sort phase ripple join estimators that have completed make use of the same number of tuples, and so the properties of their estimators are identical (this assumes that we disregard the tiny amount of bias associated with the fact that the number of ripple joins may not divide evenly into the number of tuples in either input relation, and we assume a serial and not parallel execution of the join).
- (2) Second, we know that $\text{Var}(N_i) = \infty$ for all $i > k$ because no estimate has been produced by any ripple join past the k th one. Thus, it must be the case that $w_i = 0$ for all $i > k$.

Using these two facts, along with the fact that the formula for the covariance is independent of the amount of data used to compute each ripple join (Corollary A.4), Equation 1 can be rewritten as:

$$\text{Var}(N) = (k-1)w_1^2 \text{Var}(N_1) + w_k^2 \text{Var}(N_k) + (1 - (k-1)w_1^2 - w_k^2)Q_U.$$

In this expression, Q_U is the part of the expression for the covariance that is independent of the weighting that is chosen (see Appendix A and the proof of Theorem A.3 in the article's electronic appendix for an exact expression for Q_U). To minimize this value and maximize the accuracy of N , we make use of the fact that all weights must add to one, so $(k-1)w_1 + w_k = 1$. Let $j = k-1$. Substituting, we have:

$$\text{Var}(N) = jw_1^2 \text{Var}(N_1) + (1 - jw_1)^2 \text{Var}(N_k) + (1 - jw_1^2 - (1 - jw_1)^2)Q_U.$$

We wish to minimize this value. Taking the partial derivative with respect to w_1 gives:

$$\begin{aligned} \frac{\partial}{\partial w_1} \text{Var}(N) &= 2jw_1 \text{Var}(N_1) + (-2j + 2j^2w_1) \text{Var}(N_k) \\ &\quad + (-2jw_1 + 2j - 2j^2w_1)Q_U. \end{aligned}$$

Setting this value to zero and solving gives the optimal value for w_1 :

$$w_1^{opt} = \frac{\text{Var}(N_k) - Q_U}{\text{Var}(N_1) + j\text{Var}(N_k) - Q_U - jQ_U}.$$

Given this formula for the optimal weighting of w_1 , the weight w_k can then be computed using simple substitution.

One concern is that, while this solution does find a “flat spot” on the variance function with respect to w_1 , it is not obvious that this must be a minimum. To address this, we note that the second partial derivative with respect to w_1 is always constant when the other variables are fixed:

$$\frac{\partial^2}{\partial^2 w_1} \text{Var}(N) = 2j \text{Var}(N_1) + 2j^2 \text{Var}(N_k) + (-2j - 2j^2)Q_U.$$

This means that the curve has no local minima (or maxima) that is not also a global minima (or maxima). Since the feasible values of w_1 are all from 0 to 1 inclusive, we can then conclude that the minimum variance value must be either at $w_1 = 0$, 1, or w_1^{opt} . Checking all three values is then guaranteed to yield an optimal solution.

4.2.4 Estimating the Parameters Needed to Compute $N, \text{Var}(N)$. Of course, the previous exposition assumes that we have access to $\text{Var}(N_k)$, $\text{Var}(N_1)$, and Q_U . It is an often overlooked fact that, given most sampling-based estimators, it is not possible to compute the variance of an estimator exactly since computing the variance requires that we have access to all of the data. It is not realistic to assume that we have easy access to all of the data; if we did, then there would not be a need to sample. As a result, the variance of most sampling-based estimators must be estimated as well, and the SMS join is no different.

Looking over the formulas given in Appendix A for the various components of the variance of the SMS join, there are four separate quantities that must be estimated to compute the various components of the variance of N . These quantities are:

- Q , which is the eventual answer to the query;
- $\sum_{e \in \text{EMP}} (\sum_{s \in \text{SALES}} f(e, s))^2$, which we refer to as c_1 ;
- $\sum_{s \in \text{SALES}} (\sum_{e \in \text{EMP}} f(e, s))^2$, which we refer to as c_2 ; and
- $\sum_{(e,s) \in \text{EMP} \times \text{SALES}} f^2(e, s)$, which we refer to as c_3 .

Once each of these quantities is known or estimated, then using the formulas given in Appendix A, it is an easy matter to compute $\text{Var}(N)$.

Accurately estimating each of these values requires some careful consideration. For example, consider c_1 . To compute a high-quality estimator for c_1 , we first produce a series of individual c_1 estimators where each individual estimator is computed using the data that are loaded into memory as each of the ripple joins is processed (the process of computing each of these individual estimates is described in detail later). At any given time during the execution of the SMS join, we can estimate c_1 as a simple linear combination of the estimators computed along with each individual ripple join. Let $\hat{c}_{1,i}$ denote the value of c_1 estimated using the data input into the i th ripple join. After k ripple joins have been computed, we can simply use $\hat{c}_1 = \sum_{i=1}^{k-1} \frac{\hat{c}_{1,i}}{k-1}$ as an estimate for c_1 . It is important to note that \hat{c}_1 does not make use of $\hat{c}_{1,k}$, where $\hat{c}_{1,k}$ is the value of c_1 estimated by the ripple join that is currently being executed. The reason for this is that $\hat{c}_{1,k}$ is likely to have poorer accuracy than the other estimates (this

is because if the j th ripple join is still being processed, then $\hat{c}_{1,k}$ will be estimated using fewer data points). Given a formal characterization of the effect of the reduced data on the accuracy of $\hat{c}_{1,k}$, it would be possible for each $\hat{c}_{1,i}$ to be weighted properly. In the absence of such a characterization, as long as $k > 1$, it is preferable to ignore $\hat{c}_{1,k}$ during the computation.

Estimating \hat{Q}_i , and $\hat{c}_{3,i}$. Estimating each of Q , c_1 , c_2 , and c_3 as a mean of estimates over each individual ripple join requires that we have a principled way to compute \hat{Q}_i , $\hat{c}_{1,i}$, $\hat{c}_{2,i}$, and $\hat{c}_{3,i}$ over each ripple join. For \hat{Q}_i , this is easy to do: since we do not know the eventual answer to the query, it is natural to use $\hat{Q}_i = N_i$. For $\hat{c}_{3,i}$ this is easy as well. Let EMP_i denote the set of tuples from EMP that has been assigned to the i th ripple join, and let SALES_i denote the analogous set for SALES. Let α_i denote the fraction of tuples from EMP processed by the i th ripple join, and let β_i denote the analogous fraction for SALES. Then

$$\hat{c}_{3,i} = \frac{1}{\alpha_i \beta_i} \sum_{(e,s) \in \text{EMP}_i \times \text{SALES}_i} f^2(e,s)$$

is an unbiased estimator for c_3 (this is true because $\hat{c}_{3,i}$ does nothing more than run a ripple join estimator using f^2 rather than f , so if the ripple join is itself unbiased, then so is $\hat{c}_{3,i}$). Since \hat{c}_3 is then a linear combination of unbiased estimators, it is unbiased as well.

Estimating $\hat{c}_{1,i}$ and $\hat{c}_{2,i}$. Things are not quite so simple for c_1 and c_2 , and care must be taken in developing estimators for these values. For example, consider the following natural estimator for c_1 over the i th ripple join:

$$\hat{c}'_{1,i} = \frac{1}{\alpha_i} \sum_{e \in \text{EMP}_i} \left(\frac{1}{\beta_i} \sum_{s \in \text{SALES}_i} f(e,s) \right)^2.$$

Unfortunately, this is not an unbiased estimator for $c_{1,i}$, in particular, the expected value or mean of this estimator is approximately:

$$E[\hat{c}'_{1,i}] \approx \frac{1 - \beta_i}{\beta_i} \sum_{e \in \text{EMP}_i} \sum_{s \in \text{SALES}_i} f^2(e,s) + c_{1,i}$$

(see the article's electronic appendix for a derivation of this expression). The quantity inside the parentheses is larger than $c_{1,i}$ by the amount

$$E[\hat{c}'_{1,i}] - c_{1,i} = \frac{1 - \beta_i}{\beta_i} \sum_{e \in \text{EMP}_i} \sum_{s \in \text{SALES}_i} f^2(e,s).$$

This quantity is the *bias* of $\hat{c}'_{1,i}$, and we must compensate for this bias. Thus, the estimator for c_1 that we use in our implementation of the SMS join is:

$$\hat{c}_{1,i} = \frac{1}{\alpha_i} \sum_{e \in \text{EMP}_i} \left[\left(\frac{1}{\beta_i} \sum_{s \in \text{SALES}_i} f(e,s) \right)^2 + \left(\frac{1}{\beta_i} - \frac{1}{\beta_i^2} \right) \sum_{s \in \text{SALES}_i} f^2(e,s) \right].$$

By simply switching the relations EMP and SALES, a formula for $\hat{c}_{2,i}$ is easily derived.

$$\hat{c}_{2,i} = \frac{1}{\beta_i} \sum_{e \in \text{SALES}_i} \left[\left(\frac{1}{\alpha_i} \sum_{s \in \text{EMP}_i} f(e, s) \right)^2 + \left(1 - \frac{1}{\alpha_i} \right) \frac{1}{\alpha_i} \sum_{s \in \text{EMP}_i} f^2(e, s) \right].$$

Given these various estimators, it is then an easy matter to compute N and an estimator for the variance of N .

4.3 Of Aspect Ratios and Performance Hiccups

To ensure a smooth increase of the estimation accuracy that allows the statistical information given to the user to be constantly updated as the sort phase progresses, one requires that the processing of all of the input relations during the sort phase should progress at the same rate and that the processing of each relation should finish at the same time. This is the reason for the calculations of the first few lines of Algorithm 1, which together guarantee that every time we begin the next iteration of the loop of line (6), the same fraction of each input relation has been processed.

Why would it be a problem if we find ourselves in a situation where EMP has been totally processed and organized into runs, while SALES has only been half processed (or vice versa)? The issue is that, if one relation were completely processed while the other was not, there would be a ‘hiccup’ where the estimate and accuracy could not be updated as the remainder of the other relation was sorted into runs as a prelude to the merge phase of the join.

This is one of the reasons that we do not consider the possibility of altering the relative rate at which the various input relations are processed, or *aspect ratio* of the sort phase, as was suggested by Haas and Hellerstein in their work on the ripple join [1999]. The aspect ratio of the individual ripple joins could be altered by incorporating the adaptive techniques described by Haas and Hellerstein into lines (10) and (11) of Algorithm 1, but these techniques should only be applied locally to each individual ripple join. To provide smooth, consistent increases in estimation accuracy, the requirement is that after each ripple join that makes up the sort phase has been completed, the same fraction of each input relation must have been consumed.

5. THE MERGE PHASE OF THE SMS JOIN

After the sort phase of the SMS join completes, the *merge phase* of the SMS join is invoked. The merge phase of the SMS join is similar to the merge phase of a classical sort-merge join. At all times, the merge phase of the join maintains the value:

$$total = \sum_{e \in \overline{\text{EMP}}} \sum_{s \in \overline{\text{SALES}}} f(e, s).$$

$total$ is simply the total value of the aggregate function f applied to the subsets $\overline{\text{EMP}}$ and $\overline{\text{SALES}}$ of the input relations as depicted in Figure 4 and Figure 5.

To process the input relations, the merge phase repeatedly pulls tuples off the sort phase runs and joins them. Just as in a classical sort-merge join, two sets of tuples $E \subset \text{EMP}$ and $S \subset \text{SALES}$ are pulled off the head of each run produced by

the sort phase, such that for all $e \in E$ and $s \in S$, $e.k = s.k$. Assuming that the aggregate function in question is SUM or COUNT, the merge phase then computes the value:

$$v = \sum_{e \in E} \sum_{s \in S} f(e, s).$$

Given v and $total$, then the value of f applied to $(\overline{EMP} \cup E) \bowtie (\overline{SALES} \cup S)$ is then $total = v + total$.

However, unlike in a classical sort-merge join, the merge phase of the SMS join is assigned one additional task; it must ask the shrinking phase of the join to compute an estimate over the portion of the data space that has not yet been incorporated into $total$. At the same time that the merge phase is executed, the shrinking phase is charged with maintaining the estimator for the aggregate value associated with the region $(EMP - \overline{EMP}) \bowtie (SALES - \overline{SALES})$. This estimator is then added to $total$ to produce an unbiased estimate for the eventual answer to the query.

To produce its estimate, the merge phase asks the shrinking phase to remove EMP and $SALES$ from the computation of the estimator associated with $(EMP - \overline{EMP}) \bowtie (SALES - \overline{SALES})$ via the function call $shrink(\mathbf{K})$, where \mathbf{K} is the most recent key value to be removed from both relations. In response to this function call, $shrink$ returns the value and variance of a random variable $N_{(k > \mathbf{K})}$ to the merge phase, where $N_{(k > \mathbf{K})}$ estimates the sum of f over the portion of the data space that has not yet been merged. Given this information, the current estimate for the answer to the query is simply $total + N_{(k > \mathbf{K})}$, and the variance of this estimate is $Var(N_{(k > \mathbf{K})})$. The next section considers the problem of how the shrinking phase can provide $N_{(k > \mathbf{K})}$ as well as an estimate for $Var(N_{(k > \mathbf{K})})$ to the merge phase of the SMS join.

6. THE SHRINKING PHASE OF THE SMS JOIN

As the portion of the data space associated with the merge phase grows, the task of the shrinking phase is to update the estimator N associated with the region $(EMP - \overline{EMP}) \bowtie (SALES - \overline{SALES})$ by incrementally updating each of the estimators N_1, N_2, \dots, N_n , to reflect the fact that tuples are constantly being removed from the portion of the data space covered by the sort phase estimators. Updating N to reflect the continual loss of the tuples to the sets \overline{EMP} and \overline{SALES} is equivalent to removing these tuples from each of the ripple joins, recomputing the statistics associated with each individual ripple join, and then recombining the estimators N_1, N_2, \dots, N_n (as described in Section 4.2) to produce a new estimator associated with the reduced portion of the data space. The shrinking phase systematically undoes the work accomplished during the sort phase by removing tuples from each of the sort phase ripple joins.

6.1 Sufficient Statistics for the Shrinking Phase

When the merge phase begins, keys are removed one at a time from both EMP and $SALES$ in lexicographic order. For example, imagine that the smallest key in the dataset has the value 1. The first step undertaken by the merge phase will be the removal of all tuples with key value 1 from both EMP and $SALES$. Thus, the shrinking phase must have access to statistics that are sufficient to estimate

the result of the join (EMP - {all tuples with key value 1}) \bowtie (SALES - {all tuples with key value 1}). We will use the notation $N_{(k>1)}$ to denote this new estimator. If the next key value is 2, then the next step undertaken by the merge phase is removal of all tuples with key value 2 to compute $N_{(k>2)}$. Thus, the shrinking phase should also have access to statistics that are sufficient to recompute the value of N over the relations (EMP - {all tuples with key value 1 or 2}) \bowtie (SALES - {all tuples with key value 1 or 2}) as well.

Just as N is computed as a combination of the individual estimators $N_1, N_2, \dots, N_n, N_{(k>K)}$ for the key value K is computed as a combination of the individual estimators $N_{1,(k>K)}, N_{2,(k>K)}, \dots, N_{n,(k>K)}$, the variance of $N_{(k>K)}$ is estimated just as is described in Section 4, and the formulas to estimate the variance of $N_{(k>K)}$ are almost identical to the estimator for $Var(N)$. However, the one issue that does require careful consideration is how to efficiently compute these values for each $N_{i,(k>K)}$.

6.2 The Sort Phase Revisited

Computing $N_{(k>K)}$ using the algorithms of Section 4.2 requires that we have access to $N_{i,(k>K)}, \hat{c}_{1,i,(k>K)}, \hat{c}_{2,i,(k>K)}$ and $\hat{c}_{3,i,(k>K)}$ for every value of i . However, it is impractical to reread each of the runs from disk every time that *shrink* is called in order to compute these values. This would render the SMS join far slower than classical disk-based join algorithms like the sort-merge join and hybrid hash join. To avoid the necessity of rereading each of the runs, we will instead compute $N_{i,(k>K)}$ as well as the statistics necessary to compute the variance of $N_{(k>K)}$ during the sort phase of the join, at the same time that each run is in memory.

Algorithm 2. Running a Reverse Ripple Join

- (1) Let $\alpha = \frac{|EMP_i|}{|EMP|}, \beta = \frac{|SALES_i|}{|SALES|}$
 - (2) While (*true*)
 - (3) Let K be $\max(\text{head}(EMP_i), \text{head}(SALES_i))$
 - (4) While ($\text{head}(EMP_i) == K$)
 - (5) $e \leftarrow \text{Pop}(EMP_i)$
 - (6) Add e to the ripple join RJ
 - (7) While ($\text{head}(SALES_i) == K$)
 - (8) $s \leftarrow \text{Pop}(SALES_i)$
 - (9) Add s to the ripple join RJ
 - (10) If ($\text{head}(EMP_i) == \text{head}(SALES_i) == \text{null}$) /* i th ripple join done */
 - (11) Break
 - (12) Again let K be $\max(\text{head}(EMP_i), \text{head}(SALES_i))$
 - (13) $N_{i,(k>K)} = \frac{1}{\alpha\beta} \sum_{\{e,s\} \in RJ} f(e, s)$
 - (14) $\hat{c}_{1,i,(k>K)} = \frac{1}{\alpha} \sum_{e \in RJ} \left[\left(\frac{1}{\beta} \sum_{s \in RJ} f(e, s) \right)^2 + \left(1 - \frac{1}{\beta} \right) \frac{1}{\beta} \sum_{s \in RJ} f^2(e, s) \right]$
 - (15) $\hat{c}_{2,i,(k>K)} = \frac{1}{\beta} \sum_{s \in RJ} \left[\left(\frac{1}{\alpha} \sum_{e \in RJ} f(e, s) \right)^2 + \left(1 - \frac{1}{\alpha} \right) \frac{1}{\alpha} \sum_{e \in RJ} f^2(e, s) \right]$
 - (16) $\hat{c}_{3,i,(k>K)} = \frac{1}{\alpha\beta} \sum_{\{e,s\} \in RJ} f^2(e, s)$
-

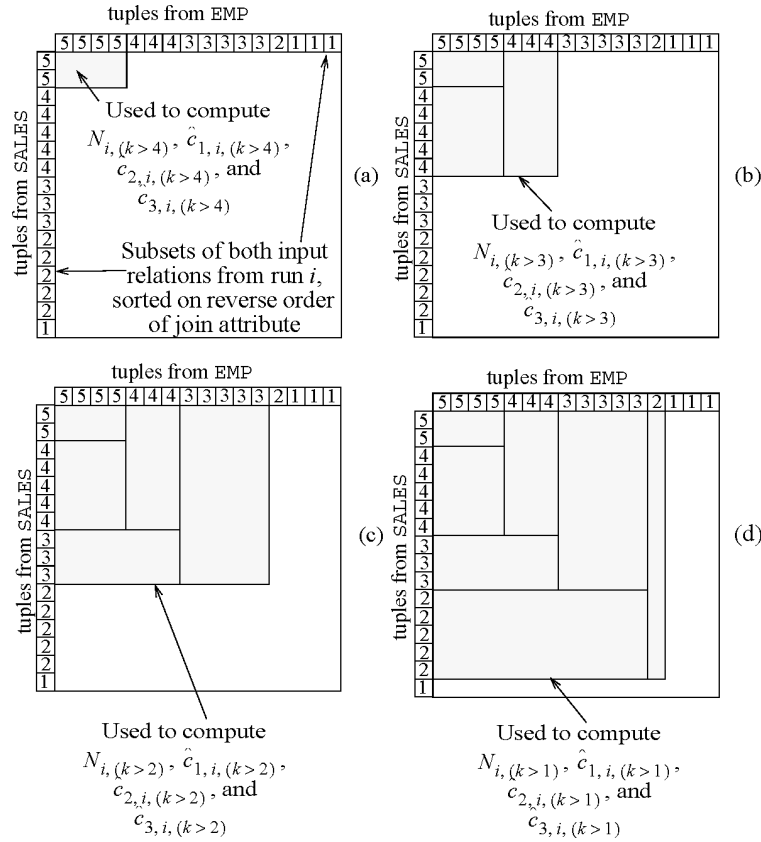


Fig. 6. Reverse ripple join used to compute statistics used by the shrinking phase.

To do this, we make the following modification to Algorithm 1. After the runs EMP_i and $SALES_i$ from EMP and SALES have been read into memory and joined via a ripple join, but before the runs have been written back to disk, both are sorted in *reverse order* according to $EMP.k$ and $SALES.k$. Then Algorithm 2 is executed. The execution of this algorithm is shown pictorially in Figure 6. Essentially, this algorithm operates exactly as a normal ripple join does except that it operates over sets of tuples with the same key values, and it operates in reverse order. As this reverse ripple join is computed, the values for each $N_{i,(k>K)}$, $\hat{c}_{1,i,(k>K)}$, $\hat{c}_{2,i,(k>K)}$ and $\hat{c}_{3,i,(k>K)}$ are computed and stored in memory for use during the shrinking phase of the join.

Note that, in general, for relatively large K , only a small portion of the data space is used to compute $N_{i,(k>K)}$. The expected value of this estimate will become smaller with increasing K . Assuming that the relative accuracy of the estimate is relatively constant over all K , (i.e., assuming that $\frac{E[N_{i,(k>K)}]}{stddev(N_{i,(k>K)})}$ does not vary much with K) then $stddev(N_{i,(k>K)})$ for large K will then be less than for small K . This implies that the variance of the SMS join’s estimate decreases continuously as the shrinking phase progresses. Eventually, as the join completes, $N_{(k>K)}$ will always be zero, and the answer to the query depends only on the value *total*.

This reflects the fact that as the relations ($\overline{\text{EMP-EMP}}$) and ($\overline{\text{SALES-SALES}}$) lose more and more tuples during the merge phase of the join, most of the estimate for the eventual answer to the query is provided by the exact quantity *total* maintained by the merge phase of the join.

6.3 Reducing Main Memory Requirements

One potential problem with the tactic of precomputing and storing these values is that, if there is a large number of distinct key values in the database and/or a large number of runs needed to perform the join, then storing the statistics necessary to compute $N_{(k>\mathbf{K})}$ may require more core memory than is available. The solution to this problem is actually rather simple. We note that we do not actually need to store these statistics for every key value since the statistics are used exclusively to update the running estimate for the eventual query result that is presented to the user. Extremely frequent updates of the estimate are probably overkill and will be ignored by the user anyway. In other words, if the key value 5 has been processed, but we do not have $N_{i,(k>5)}$, $\hat{c}_{1,i,(k>5)}$, $\hat{c}_{2,i,(k>5)}$, and $\hat{c}_{3,i,(k>5)}$ available, then it is not a problem; we are simply unable to give the user an estimate for the query result until we process a key value for which we do have the necessary statistics.

Thus, when we process the first run during the sort phase, we choose an acceptably small and yet comprehensive set of key values for which to compute statistics. In our implementation of the SMS join, this is done in the following manner. First, we compute an estimate in seconds s for the amount of time that the merge phase will take to complete (a rough estimate for this is the time that is required for both EMP and SALES to be read in their entirety from disk). Then, to see whether we will compute and store statistics for a given key value k , we hash k and mod the result by s . If the result of the mod operation is zero, then statistics are computed and stored for that key. Assuming no repeated key values, this means that, in a very limited amount of main memory, we will store approximately enough statistics to update the estimate for the eventual result of the query every second.

6.4 Total Main Memory Requirements

Given this optimization, the main memory required by the SMS join (above and beyond the memory required by a classical sort-merge join) is described as follows. During the sort phase, we store \hat{Q}_i , $\hat{c}_{1,i}$, $\hat{c}_{2,i}$, and $\hat{c}_{3,i}$ (amounting to around 32 bytes total) for each individual ripple join. Note that the number of individual ripple joins is analogous to the number of runs in a classical sort-merge join. Since even for the largest joins, this number will probably never exceed a few thousand, this memory requirement is generally bounded by a few kilobytes.

The sort phase also collects enough data so that we can update $N_{(k>\mathbf{K})}$ (and associated confidence bounds) every few seconds during the shrinking phase of the join. As described in Section 6.2, this requires that we have access to $N_{i,(k>\mathbf{K})}$, $\hat{c}_{1,i,(k>\mathbf{K})}$, $\hat{c}_{2,i,(k>\mathbf{K})}$ and $\hat{c}_{3,i,(k>\mathbf{K})}$ for every value of i (i.e., for every individual ripple join) and for enough \mathbf{K} values that we can provide frequent estimate updates.

For example, consider a join over two database tables containing 10TB of data in all. If these data were stored on 100 disks, it would take approximately three hours of continuous sequential I/O to complete the join (at a sequential I/O rate of 50 MB/sec per disk). Of this time, approximately one hour would be devoted to the shrinking phase. 1000 sets of statistics would then be enough for one update every 3.6 seconds during the shrinking phase. If approximately 10GB of RAM were devoted to this join (resulting in 1000 individual ripple joins), we might expect on the order of 32MB of main memory in total to store the required statistics: $(32 \text{ bytes per set of stats}) \times (1000 \text{ ripple joins}) \times (1000 \text{ updates}) = 32\text{MB}$.

Though this is relatively little memory for a modern data warehousing system archiving terabytes of data, we do note that multiple sets of values may be required if many estimates are produced simultaneously by a single SMS join (e.g., if the task to estimate an aggregate over many groups in a GROUP BY query). This may increase the memory requirement significantly. Continuing with our previous example, for a GROUP BY query with 1,000 groups, we would require 32GB of storage total, as opposed to 32MB. This *is* a significant memory requirement. However, in this case the requirement could be reduced arbitrarily by flushing this data to disk and reading it back as needed during the merge phase. Note that the statistics are both accessed and produced in an orderly fashion, based upon a linear ordering of K values. Thus, reading these values from disk would cause little thrashing and would lead to only a marginal increase in the total I/O cost. In our example, an additional 32GB of I/O is negligible compared to the 30TB of I/O that would be required to evaluate such a large join using a scalable join algorithm.

7. REDUCING THE PERFORMANCE HIT

Aside from the additional memory required by the SMS join compared to the sort-merge join, there is another concern. Might the additional computation render the SMS join significantly slower than the sort-merge join? Note that as described thus far, the following steps are required for each sort-phase ripple join:

- (1) read in a run of data from both EMP and SALES;
- (2) join them using a hashed ripple join;
- (3) sort both runs in reverse order with respect to the join key;
- (4) rejoin the data in reverse order using a second hashed ripple join in order to collect the required $N_{i,(k>K)}$, $\hat{c}_{1,i,(k>K)}$, $\hat{c}_{2,i,(k>K)}$ and $\hat{c}_{3,i,(k>K)}$ values;
- (5) flip the order of both runs so they are sorted in the correct order;
- (6) write the sorted runs back to disk.

Given all of these steps, significant extra computation may be required compared to the sort phase of a classical sort-merge join. Specifically, the SMS join requires steps (2), (4), and (5), while the classic sort-merge join does not. The result is that a naive or straightforward implementation of the SMS join may be somewhat slow compared to typical implementation of the sort-merge join as we will demonstrate in the next section of the article.

Fortunately, through careful implementation, it turns out that it is possible to speed up the SMS join considerably by effectively eliminating most of the additional computation that is required. To accomplish this, we note that just as no particular sort order is required by a classical sort-merge join, there is nothing particularly special about the ordering used by the reverse ripple join described in Section 6.2. In fact, any lexicographic ordering is sufficient as long as that ordering is the reverse of the ordering used by the merge phase of the SMS join. We can use this observation to our advantage in order to reduce the performance hit associated with the SMS join. In the following, we outline an implementation of the SMS join that will have very little performance penalty compared to a classical sort-merge join. There are three important considerations.

Hashing During the Sort-Phase Ripple Join. The key to achieving the desired efficiency is to carefully choose how tuples are partitioned into buckets during each sort-phase ripple join so that we can use this bucketing to reduce the amount of computation that is required later on. To accomplish this, we make use of a randomization function H that provides a random mapping from the set of all possible join key values to the set of all possible join key values when running each sort-phase ripple join. In other words, if the join key is a 32-bit integer, we make sure to use a hash function that is a random bijection (invertible function) from the set of all 32-bit integers to the set of all 32-bit integers.

Then, given a domain size of k for the join key, in order to determine the identifier of the bucket that tuple r is hashed into, we use the function $H(r.key) \text{div } (k \text{div } b)$, where b is the desired number of buckets. The result is that, after the runs from both input relations have been read into memory and joined using the hashed ripple join, for each tuple pair r_1 and r_2 , if $H(r_1.key) < H(r_2.key)$, then the identifier of the bucket that r_1 is contained in is always less than the identifier of the bucket that r_2 is in. This will facilitate the following two optimizations.

Eliminating the Reverse Ripple Join. This ordering is important because we can effectively run our reverse ripple join without sorting (or even looking at) the tuples in a bucket by simply looking at each bucket as a whole, in order, from back to front. Strictly speaking, this does not eliminate the requirement for a reverse ripple join, but it almost eliminates the additional computation associated with it. If, during the forward ripple join, each bucket B_i is annotated with the summations $\sum_{\{e,s\} \in B_i} f(e,s)$, $\sum_{e \in B_i} (\sum_{s \in B_i} f(e,s))^2$, $\sum_{s \in B_i} (\sum_{e \in B_i} f(e,s))^2$, and $\sum_{\{e,s\} \in B_i} f^2(e,s)$, and each \mathbf{K} is chosen so that \mathbf{K} is the smallest $H(r.key)$ value in some bucket m , then we can collect the statistics we need from the j th reverse ripple join by simply using the formulas:

$$\begin{aligned}
N_{j,(k>\mathbf{K})} &= \frac{1}{\alpha\beta} \sum_{i=m}^b \sum_{\{e,s\} \in B_i} f(e,s), \\
\hat{c}_{1,j,(k>\mathbf{K})} &= \frac{1}{\alpha} \sum_{i=m}^b \sum_{e \in B_i} \left[\left(\frac{1}{\beta} \sum_{s \in B_i} f(e,s) \right)^2 + \left(1 - \frac{1}{\beta} \right) \frac{1}{\beta} \sum_{s \in B_i} f^2(e,s) \right], \\
\hat{c}_{2,j,(k>\mathbf{K})} &= \frac{1}{\beta} \sum_{i=m}^b \sum_{s \in B_i} \left[\left(\frac{1}{\alpha} \sum_{e \in B_i} f(e,s) \right)^2 + \left(1 - \frac{1}{\alpha} \right) \frac{1}{\alpha} \sum_{e \in B_i} f^2(e,s) \right], \\
\hat{c}_{3,j,(k>\mathbf{K})} &= \frac{1}{\alpha\beta} \sum_{i=m}^b \sum_{\{e,s\} \in B_i} f^2(e,s).
\end{aligned}$$

Note that these values for each and every K can be obtained in a simple linear scan of the buckets from back-to-front by simply maintaining a current running total for each of the four statistics.

Facilitating a Linear Time Sort. Finally, there is another important advantage to using the bucketing just described. Specifically, since each $H(r.key)$ value in a smaller bucket is strictly less than each $H(r.key)$ value in a larger bucket, if we simply internally sort the contents of each bucket, we can obtain a sorted lexicographic ordering of all of the tuples with respect to the function H by scanning the hash buckets in order. If the total number of tuples n is $\Omega(b)$, then there will be $O(1)$ tuples in each hash bucket and after the hash ripple join has completed, obtaining a list of tuples sorted on $H(r.key)$ is then an $O(n)$ operation (requiring only that we sort the $O(1)$ tuples in each bucket). This means that after running the sort-phase ripple join, we can obtain a sorted order that can be used by the merge phase of the join without actually sorting the tuples.

The net result of this is that, aside from the fact that we will have some extra computation to produce the running estimates and confidence bounds as the data are loaded, with a careful implementation, we may expect little or no performance hit in the SMS join compared to a sort-merge join. As we demonstrate experimentally in the next section, these optimizations typically cut the additional cost associated with the SMS join compared to the sort-merge join by a factor of two.

8. BENCHMARKING

This section details the results of a set of experiments aimed at benchmarking the SMS join. All of the data used in the electronic experiments can be accessed at <http://www.cise.ufl.edu/~cjermain/SMS>. The goal of the experiments was to answer two basic sets of questions:

- In reality, what are the statistical properties of the SMS join?* Specifically, does the variance of the SMS join's statistical estimator match the rather complicated formulas derived in the electronic appendix of the article? What is the shape of the estimator's distribution? Is it safe to assume a normal or bell-curved shape when computing confidence bounds?
- In practice, how does the join perform?* Specifically, how do the confidence bounds given by the SMS join shrink as a function of time? How does the time to completion of the SMS join compare to its close cousin, the sort-merge join?

8.1 Statistical Properties of the SMS Join

Due to space constraints, details related to first set of experiments are included in the electronic Appendix D of the article (available in the ACM Digital Library). The key results of this set of experiments are that (1) the variance formulas in the article do in fact seem to be accurate, and (2) that it is safe to assume normality when using those variance formulas to provide confidence bounds on the SMS join's estimates.

8.2 The SMS Join Over Large, Disk-Based Relations

8.2.1 Experimental Setup. The second set of experiments was designed to demonstrate the performance characteristics of the SMS join when it is used over a relatively large, disk-based data set and to test the ability of the SMS join to shrink its estimator's confidence interval over time (some additional experiments along these lines are detailed in Appendix D of the article). To perform these tests, we created a series of synthetic datasets. Each dataset consisted of a pair of synthetic relations, `EMP` and `SALES`, and all queries tested were of the form:

```
SELECT SUM (EMP.VAL)
FROM EMP, SALES
WHERE EMP.KEY = SALES.KEY
```

Each relation tested had a physical size of 20GB and consisted of 200 million, 100-byte tuples. The tuples in each dataset shared 100,000 distinct key values, and the assignment of key values to tuples was performed according to a Zipfian distribution with a parameter C . Specifically, the most frequent key value in each relation had a frequency proportional to $1/(1^C)$, the second most frequent key value had a frequency proportional to $1/(2^C)$, the third most frequent key value had a frequency proportional to $1/(3^C)$, and so on. Frequencies of key values in both `EMP` and `SALES` were independent, so the rank of a given key value in `EMP` is randomly chosen and is independent of the key value's rank in `SALES`. The attribute value `EMP.VAL` for each tuple was generated using a well-behaved normal random variable with a standard deviation that was just about 3% of the variable's mean.

All datasets were stored on a single, 80GB SCSI disk. Experiments were performed on a Pentium 4 machine running the Linux operating system, with 1GB of RAM. In the experiments, six sets of `EMP` and `SALES` relations were created, using Zipf parameter C in the list (0, 0.2, 0.4, 0.6, 0.8, 1.0). After creating each data set, an SMS join (without the optimizations described in Section 7) was used to run the corresponding query to completion, and the current estimate and confidence interval were tracked throughout execution.

For comparison, a sort-merge join was also used to evaluate each query, to determine what sort of total execution-time hit one might expect were an SMS join used instead of a more traditional join algorithm. While the classical sort-merge join is not the most advanced join algorithm available, it is still a good choice for comparison because it is ubiquitous in real database systems, its performance characteristics are very widely understood, and it forms the basis for the SMS join. We also implemented a second, optimized version of the SMS join which makes use of the ideas described in Section 7 of the article. For each of the six datasets having Zipf parameters (0, 0.2, 0.4, 0.6, 0.8, 1.0), we reran each join (the sort-merge join, the SMS join, and the optimized SMS join) to completion three separate times with a cold disk buffer and compute the total running time for each of the (dataset, join algorithm) combinations.

The results of these experiments are depicted in Figure 7 and Figure 8. Figure 7 shows the estimate and confidence interval of the SMS join (at a 95%

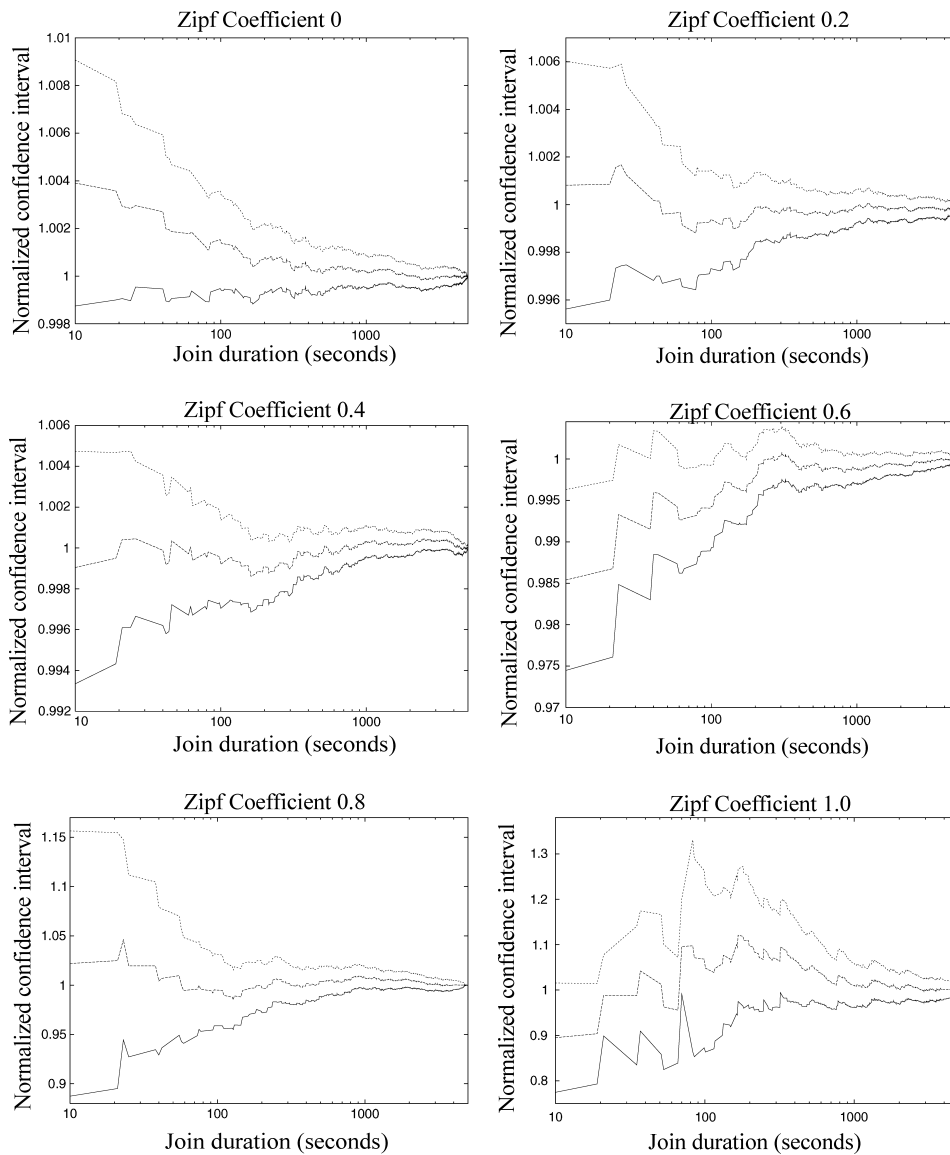


Fig. 7. Confidence intervals as a function of time for six joins over synthetic datasets with varying Zipfian skew. For each join, three lines are plotted: the upper and lower limits of the confidence interval, as well as the current estimate over time. The plots begin ten seconds into the SMS join, which is approximately the time that the first ripple join runs out of memory.

confidence level) as a function of time for each of the six tests. The confidence intervals were computed using an assumption of normality. The plot begins 10 seconds into the join, which is approximately the time at which the first ripple join runs out of memory. Figure 8 shows the time-to-completion for the SMS join, the optimized SMS join, and for the sort-merge join over the various joins depicted in Figure 7.

Zipf parameter	SMS Join	Opt. SMS Join	Sort-Merge Join	Ratio
$C = 0$	252:52	231:25	212:14	1.19 / 1.09 / 1.0
$C = 0.2$	249:26	229:17	212:29	1.17 / 1.08 / 1.0
$C = 0.4$	250:34	225:40	206:01	1.21 / 1.09 / 1.0
$C = 0.6$	244:57	231:42	205:56	1.19 / 1.12 / 1.0
$C = 0.8$	243:07	227:25	203:57	1.19 / 1.11 / 1.0
$C = 1.0$	228:06	215:59	204:09	1.11 / 1.05 / 1.0

Fig. 8. Comparison of the total time (mmm:ss) required to use the SMS join, the optimized SMS join, and a traditional, sort-merge join, to perform three separate joins over two, 20GB relations.

8.2.2 Discussion. These experiments show two main results. First, they demonstrate that the SMS join is effective in continuing to shrink confidence bounds long after main memory has been exhausted. Each of the six plots in Figure 7 shows that, compared to the point at which a standard ripple join would run out of memory (around 10 seconds in our experiments), the SMS join is able to again cut the confidence interval width in half to a third after around 100 seconds total, and then cut it in half to a third again after 1,000 seconds total. After 1,000 seconds (approximately 30% of the SMS join’s total execution time for our experiments), the confidence interval has typically shrunk to around 10–12% of the width that was achieved after only 10 seconds. We point out that such a reduction in confidence interval width is fairly significant. For example, consider the case of the 0.0 Zipf coefficient. The SMS join was able to reduce the $\pm 0.5\%$ error interval at the time that main memory was exhausted to around a $\pm 0.1\%$ error interval after 1,000 seconds. This effect was fairly uniform over all six of the queries tested, with the main difference among the queries being the width of the interval at the time that main memory was exhausted. The width at the time that main memory was exhausted ranged from a confidence interval of approximately $\pm 0.5\%$ for a Zipf coefficient of 0.0, to an interval of $\pm 12\%$ for a Zipf coefficient of 0.8, to even larger for a Zipf coefficient of 1.0 (though it should be noted that the confidence interval early on in the case of the 1.0 Zipf coefficient is somewhat unreliable, due to the high variance of the confidence interval estimator itself).

Another point worth mentioning is that, of all of the thousands of individual estimates plotted in Figure 7, more than 97% percent were correct in the sense that the interval did in fact contain the exact answer (despite the fact that 95% confidence bounds were specified). Since these estimates were computed using the assumption that the SMS join’s estimator is normally distributed, this would seem to be yet another data point that strongly argues for the safety (and perhaps even for the conservativeness) of an assumption of normality.

We note that according to Figure 8, we observed around a 20% hit in performance compared to the sort-merge join if the SMS join is used. This performance hit can be cut in half, to around 10%, if one makes use of the optimization described in Section 7 of the article. Overall, two main conclusions can be drawn

from these results. First, the SMS join offers clear benefits when online aggregation is a viable alternative to more traditional algorithms as it provides a very effective means of continuing to shrink confidence bounds long after main memory is exhausted. Second, with a careful implementation, the performance of the SMS join can be made very competitive with the performance of a classical sort-merge join. This means that the SMS join really does give accuracy guarantees essentially “for free” compared to the traditional sort-merge join.

9. RELATED WORK

Online aggregation was first proposed by Hellerstein et al. [1997]. Their work has its roots in a large body of earlier work concerned with using sampling to answer aggregate queries; the most well-known early papers are by Hou et al. [1998, 1999]. Hellerstein et al.’s initial work on online aggregation eventually grew into the UC Berkeley Control project [1999], resulting in the development of the ripple join [Haas and Hellerstein 1999] which serves as the basis for the sort phase estimators used by the SMS join.

In a follow-up work to the ripple join, Luo et al. [2002] developed a parallel join processing algorithm that maintains statistical guarantees as long as the data that are aggregated over can be buffered in main memory. However, unlike the SMS join, the scalable algorithm of Luo et al. does not maintain any statistical guarantees once enough data have been processed that they cannot be buffered in main memory. The only existing disk-based join algorithm that could conceivably be used to maintain statistical guarantees throughout execution over a large database as the SMS join does is the nested-loops version of the ripple join [Haas and Hellerstein 1999]. However, like other nested-loops joins, the nested-loops ripple join is not scalable. If the dataset to be aggregated is n times larger than main memory, then the join will require a quadratic number of disk I/Os with respect to n .

This article is an extended and revised version of an earlier paper that appeared in the 2005 SIGMOD Conference [Jermaine et al. 2005] and introduced the SMS join. The most significant technical differences between that paper and this one are related to how the accuracy guarantees associated with the join’s estimate are computed. First, the SIGMOD paper’s variance estimate makes use of Haas and Hellerstein’s infinite population variance analysis for a single ripple join. This may be a questionable decision because the general rule-of-thumb is that infinite and finite analyses are interchangeable until 10% of a population has been considered. In the case of the SMS join, all of the database data will be considered during the computation which results in a violation of the 10% rule. The version of the SMS join described in this article rectifies this problem by making use of an exact finite population analysis that is valid at all times during the SMS algorithm.

Second, the SIGMOD paper’s algorithm estimates the SMS join’s variance and weights the various ripple join estimators by associating an individual and independent variance estimate with each sort-phase ripple join. Unfortunately, we found this to be problematic for less well-behaved data. As the data become more skewed, it is likely that the variance estimates associated with one or

two individual ripple joins may be highly inaccurate. As a result, the weighting of the individual ripple joins that may be skewed towards those joins are inaccurately classified as having lower variance. Weighting such joins heavily can then result in wild fluctuations in the estimated query answer as well as relatively inaccurate confidence bounds. The variance estimator described in this article makes use of all available information at all times to perform the weighting and avoid this problem and tends to result in smoother and more reliable estimation behavior.

Finally, the SIGMOD paper's variance estimate also ignores the correlation between the various ripple joins. The SIGMOD paper argues that simply ignoring this correlation is acceptable since the covariance among ripple join estimates tends to be negative. Thus, by ignoring the correlation, the resulting accuracy bounds will actually tend to be more conservative than if the correlation had been taken into account. Unfortunately, a proof that the covariance among ripple join estimates is always negative has alluded us to date, and so there is a chance that datasets exist where ignoring the covariance may result in confidence bounds that are too aggressive.

There are a few other more minor differences between the SIGMOD paper and this one. For example, this article considers implementation issues more carefully, including how to reduce the performance hit associated with the additional computation necessitated by the SMS join compared to traditional join algorithms, and benchmarks the resulting algorithmic improvements. This article's electronic appendix demonstrates experimentally the accuracy of the new variance calculations and shows experimentally that the estimator resulting from the SMS join is approximately normal.

Aside from the work on online aggregation, the other existing work most closely related to this article can be divided into three categories. The first category is a set of papers that describes how to implement disk-based join algorithms that are able to give early results. The algorithm of this type that is most closely related to the SMS join is a variant of the sort-merge join proposed by Dittrich et al., [2002] called the Progressive Merge Join. The SMS join may be viewed as an extension to the basic ideas used to develop the Progressive Merge Join. Like the Progressive Merge Join, the SMS join reads runs from both input relations concurrently, and, during the sort phase, both the SMS join and the Progressive Merge Join search runs for early output results while their constituent tuples are still in memory. Aside from a few technical differences (such as the fact that the SMS join uses a hashed ripple join algorithm rather than a plane sweep to produce its early results), the primary difference between the Progressive Merge Join and the SMS Join is the addition of considerable machinery required to produce the estimate for the final query result as well as the statistical guarantees as the join progresses.

In later work, Dittrich et al. [2003] consider a generalization of the Progressive Merge Join where computational resources are targeted towards specific computations in order to directly control the balance between early results and total query evaluation time. The main relevance of their generalization to the SMS join is that, given the close relationship between the SMS join and the Progressive Merge Join, it may be possible to extend these ideas to the SMS

join. For example, consider Figure 4 and Figure 5. Rather than continuing with the third and fourth ripple joins (as in Figure 5(e)), we could instead combine the first and second from Figure 4(d) and draw new estimates from this combination before we continue with the additional ripple joins. This may increase accuracy more quickly at the cost of a longer total query evaluation time, just as Dittrich et al. suggest. Unfortunately, an extension to the SMS join of this sort is not straightforward, and thus we leave this as an issue for future work. The difficulty is that it is not immediately clear how to combine two individual ripple joins that do not cover both input relations so as to give statistically valid accuracy guarantees at the same time. In the SMS join it is possible to combine all of the sort-phase ripple join estimates online because we can treat the aggregate value associated as the total-joined portion of the data space (the growing black region of Figure 5) as a constant. If instead we attempt to merge two ripple joins together that do not cover an entire input relation, no portion of the data space is fully joined; thus, we would still need to estimate a total value for this portion of the data space and take into account the statistical properties of the process.

Aside from the Progressive Merge Join, other fast-first join algorithms have been proposed. Various hash-based schemes have also been proposed for producing initial result tuples fast: the Double-Pipelined Hash Join [Ives et al., 1999] the X-Join [Urhan and Franklin 2000] and the Hash-Merge Join [Mokbel and Lu 2004]. In addition, another recent paper considered how to design a nonblocking algorithm for spatial joins [Luo et al. 2002].

The next two categories of related work are also closely related to the contents of this article. The first of these two categories is a set of papers primarily focused on producing statistical guarantees for various aggregate functions approximated by sampling over joins. The second category is a set of papers that describes algorithms using methods besides sampling to perform the same task.

We begin by briefly considering the first set of papers, those dealing with statistical guarantees for sampling over joins. The papers most closely related our work are a series of papers analyzing joins over samples where more than one relation is sampled and the samples are joined. This series begins with an analysis of this type of join that is applicable to `COUNT` queries (the paper in question is authored by Hass et al. [1996]). The analysis was then extended to more general queries by Haas [1997] and ultimately led to the bounds presented in the ripple join paper [Hellerstein et al. 1997]. However, the key difference between the analysis presented in those papers and the analysis presented in the Appendix was outlined in Section 4.2.1: the exact, finite population, multiple-partition analysis presented here is quite different from the analysis given in those papers.

There is a larger body of work that deals with applying more traditional statistical methods to providing guarantees on sampling over joins. For example, we point out the earlier work on adaptive sampling over joins for selectivity estimation performed by Lipton and Naughton [1990] and Lipton et al. [1990], the join synopses of Achary et al. [1999] and the early work by Hou and Özsoyoglu [1991], and Hou et al. [1988, 1999]. However, this work is only tangentially related to our own work by the fact that these papers consider accuracy bounds

over joins. The work is not relevant in the case (as in the SMS join) where multiple input relations are sampled concurrently and on-the-fly, and the result of the join over the samples is used to estimate the eventual query result. These papers either consider sampling the output space of the join or sampling one relation and joining the sample exactly with the other. While such algorithms are amenable to traditional methods of analysis, the statistical properties of the sort of estimators used by the ripple join and the SMS join are far more difficult to reason about (hence the complexity of the analysis in the electronic appendices of this article). Specifically, there is no independent and identically distributed structure to the concurrent sampling performed by the ripple join and the SMS join since the estimates produced by two tuples sampled from the same relation are not independent and identically distributed if they are computed by joining them with samples from another relation. This difficulty corresponds precisely to the observation made by Olken in his Ph.D. thesis [1993] and later reiterated by Chaudhuri et al. [1999]: sampling and the join operation do not commute. A join of two sampled relations is not a sample from the result of a join.

In addition to the work on sampling for estimation over joins, a final category of related work contains a set of papers that describes algorithms computing statistical estimators with quality guarantees where the underlying estimator is not based on sampling. For example, several recent papers advocate the use of *sketches*, an idea pioneered by Alon et al. [1999, 2002], and later extended to more complex select-project join queries by Dobra et al. [2002]. Sketches have been the focus of even more recent work, such as a paper applying sketches to spatial joins [Das et al. 2004]. In addition to sketches, it is also possible to use histograms to estimate the value of an aggregate function over a join, and indeed histograms have been used for many years as a way to estimate the answer to COUNT queries over joins [Kooi 1980] (though most work of this kind is targeted to performing selectivity estimation). However, until very recently there has not been a serious attempt to characterize the accuracy of such estimates in a statistically rigorous fashion [Dobra 2005]. Though such methods can be used to develop useful statistical estimators over joins, the fundamental difference between these methods and the samples used by the SMS join is that samples can be used to produce online estimates. In other words, if a sample of size n is used to produce an estimate and the user does not like the accuracy of the result, then it is easy to incorporate new samples into the estimate to increase accuracy. This is why the SMS join is able to continuously increase the accuracy of its estimate throughout execution. Sketches and histograms have no such structure; they are constructed in fixed space in a single pass over the data, and so they always return an estimate with fixed precision.

10. CONCLUSIONS AND FUTURE WORK

In this article, we have introduced the SMS join, which is a join algorithm suitable for online aggregation. The SMS join provides an online estimate and associated statistical guarantees on the accuracy of the estimate from start-up through completion. The key innovation of the SMS join is that it is the first

join algorithm to achieve such statistical guarantees with little or no loss in performance compared with offline alternatives (like the sort-merge join or the hybrid hash join) even if the input tables are too large to fit in memory. Given that the SMS join combines statistical guarantees with efficiency in a disk-based environment, it would be an excellent choice for computing the value of an aggregate function over the result of a relational join.

Several main directions for future work became obvious to us during this project. Even in today's largest data warehouses (the most likely application domain for the SMS join), it is unlikely that more than a handful of relations will be too large to fit into memory. This is because, in most data warehouses, having more than one or two central fact tables is a rare occurrence. However, if there are more than two very large, disk-resident fact tables to join, the algorithms described in this article are not easily applicable. It is an interesting problem to extend the SMS join to handle the multi-table joins that may occur if there are many large fact tables present in the database. We are currently considering how to handle this. An often observed but almost always ignored problem when performing any sort of statistical estimation is that confidence bounds are themselves estimates and can be inaccurate when only a small amount of information is used during their computation. In particular, this is the reason for the slightly strange confidence bounds depicted in Figure 7 in the case of a 1.0 Zipf coefficient. Such inaccuracy is dangerous in online aggregation since it can give the user a false sense of security, and cause the user to abort the join early. Research aimed at addressing this danger by deriving safer and less aggressive variance estimates early on is an important future direction. One intriguing idea for allowing even faster reduction in confidence bound width would be to identify during the sort phase those important tuples which join with many other tuples in the dataset and to buffer those tuples in memory and never write them back to disk. This would result in a method that resembles an online version of the bifocal sampling algorithm of Ganguly et al. [1996]. Finally, we mention once again the discussion from Section 9 on incorporating the ideas of Dittrich et al. [2003] regarding the use of multiple merge phases to trade off accuracy and join completing time. Working out the technical problems associated with describing the accuracy of such an approach would make the SMS join a much more flexible tool.

APPENDIXES

APPENDIX A. VARIANCE AND COVARIANCE ANALYSIS

This Appendix gives the central theoretical results of the article with respect to the SMS join's accuracy. The proofs are included in the electronic Appendix E, available in the ACM Digital Library.

THEOREM A.1. THE SUM OF THE VARIANCES.

$$\begin{aligned} Q_V &= \sum_i^n w_i^2 \text{Var}(N_i) \\ &= \frac{|\mathbf{E}||\mathbf{S}| \sum_i^n w_i^2}{(|\mathbf{E}| - 1)(|\mathbf{S}| - 1)} \left[Q^2 \sum_i^n \left(\frac{(|E_i| - 1)(|S_i| - 1)}{|E_i||S_i|} - \frac{(|\mathbf{E}| - 1)(|\mathbf{S}| - 1)}{|\mathbf{E}||\mathbf{S}|} \right) \right] \end{aligned}$$

$$\begin{aligned}
& + c_1 \sum_i^n \left(\frac{(|E| - |E_i|)(|S_i| - 1)}{|E_i||S_i|} \right) + c_2 \sum_i^n \left(\frac{(|E_i| - 1)(|S| - |S_i|)}{|E_i||S_i|} \right) \\
& + c_3 \sum_i^n \left(\frac{(|E| - |E_i|)(|S| - |S_i|)}{|E_i||S_i|} \right) \Big]
\end{aligned}$$

where

$$\begin{aligned}
c_1 &= \sum_{e \in E} \left(\sum_{s \in S} f(e, s) \right)^2 \\
c_2 &= \sum_{s \in S} \left(\sum_{e \in E} f(e, s) \right)^2 \\
c_3 &= \sum_{(e, s) \in E \times S} f^2(e, s) \\
Q &= \sum_{(e, s) \in E \times S} f(e, s)
\end{aligned}$$

A simple corollary of Theorem A.1 is the following.

COROLLARY A.2. EXACT VARIANCE OF AN INDIVIDUAL RIPPLE JOIN. *After an individual ripple join has processed a fraction $1/n$ of E and $1/n$ of S , the variance of its associated estimator is:*

$$\begin{aligned}
\text{Var}(N_i) &= \frac{(n-1)}{(|E|-1)(|S|-1)} \\
&\times \left[(n+1) - |E| - |S| \right) Q^2 + |E|(|S|-n)c_1 + |S|(|E|-n)c_2 + (n-1)|E||S|c_3 \Big]
\end{aligned}$$

Now we consider the covariances.

THEOREM A.3. THE SUM OF THE COVARIANCES.

$$Q_C = \sum_i^n \sum_{j \neq i}^n w_i w_j \text{Cov}(N_i, N_j) = \left(1 - \sum_i^n w_i^2 \right) Q_U$$

where

$$Q_U = \frac{|E||S|}{(|E|-1)(|S|-1)} \left[\frac{(|E|+|S|-1)}{|E||S|} Q^2 - (c_1 + c_2 - c_3) \right]$$

An interesting corollary of this is the following.

COROLLARY A.4 (EFFECT OF RIPPLE JOIN SIZES ON COVARIANCE). *The total covariance among all partitions is independent of the partition sizes that are chosen.*

APPENDIX B. WHAT IF EMP AND/OR SALES ARE NOT CLUSTERED RANDOMLY?

All of the material in the article thus far assumes that a randomized ordering of EMP and SALES is available. Even if such an ordering is not available, the SMS join can still be applied after a few additional considerations have been taken into account.

The most obvious way to allow for an SMS join over nonrandomly-ordered data would be to randomly select tuples from EMP and SALES one-at-a-time and use the resulting randomly-ordered tuples as input into the SMS join. This would allow the SMS join algorithm as described thus far to be used with no

additional modification. For example, one could first randomly permute the numbers from 1 to $|\text{EMP}|$ to obtain a list L . Then, the tuples from EMP would be accessed in the order mandated by L . The first tuple from EMP to be processed would be $L[1]$, the second would be $L[2]$, the third $L[3]$, and so on, up to $L[|\text{EMP}|]$. A similar process could be used for SALES. However, this solution is probably not a good one since it would introduce several additional costs, not the least of which would be the tremendous additional I/O delay incurred due to the access of all of the records from EMP (and SALES) randomly one-at-a-time. This would likely render the SMS join many orders of magnitude slower than a scalable alternative such as the sort-merge join or the hybrid hash join.

A more suitable application of this basic technique would be to logically partition EMP and SALES into a large number of contiguous runs of blocks. The resulting relations would then be viewed as two new relations EMP' and SALES' of size $|\text{EMP}'|$ and $|\text{SALES}'|$, respectively. A run of blocks in EMP is treated as a single tuple in EMP'. The SALES relation is treated similarly. Then, one could run the SMS join over the resulting relations EMP' and SALES' using exactly the technique described in the previous paragraph with each run accessed in a randomized fashion. The only additional change to the SMS join's algorithms and analysis is that since each $e \in \text{EMP}'$ and $s \in \text{SALES}'$ are actually runs of blocks rather than single tuples, in order to compute $f(e, s)$, one must use $f(e, s) = \sum_{\tilde{e} \in e} \sum_{\tilde{s} \in s} f(\tilde{e}, \tilde{s})$, where \tilde{e} and \tilde{s} are tuples from EMP and SALES, respectively.

The reason that this solution is preferable to random access of individual tuples is that, while accessing tuples randomly is very costly, accessing random runs of blocks is much less so because the randomization is amortized over all of the tuples in the run. For example, if one assumes a 5ms random access time and a 50MB/sec sequential read rate, then accessing a large relation as a series 10MB random sequential reads incurs an overhead of less than 3% due to the random I/O, compared to a single sequential scan.

Still, whenever possible, the SMS join should be used over randomly-ordered input relations. The reason for this is that if there is correlation in the ordering of result tuples which tends to result in large $f^2(e, s)$ values for random (e, s) pairs where $e \in \text{EMP}'$ and $s \in \text{SALES}'$, then using a solution such as the one detailed in this appendix may result in confidence bounds that shrink much more slowly compared to the randomly-permuted case.

REFERENCES

- ACHARYA, S., GIBBONS, P. B., POOSALA, V., AND RAMASWAMY, S. 1999. Join synopses for approximate query answering. *SIGMOD Conference* 275–286.
- ALON, N., GIBBONS, P. B., MATIAS, Y., AND SZEGEDY, M. 1999. Tracking join and self-join sizes in limited storage. *PODS Conference* 10–20.
- ALON, N., GIBBONS, P. B., MATIAS, Y., AND SZEGEDY, M. 2002. Tracking join and self-join sizes in limited storage. *J. Comput. Syst. Sci.* 64, 3, 719–747.
- CHAUDHURI, S., MOTWANI, R., AND NARASAYYA, V. R. 1999. On random sampling over joins. *SIGMOD Conference* 263–274.
- COCHRAN, W. 1977. *Sampling Techniques*. John Wiley and Sons.
- DAS, A., GEHRKE, J., AND RIEDEWALD, M. 2004. Approximation techniques for spatial data. *SIGMOD Conference* 695–706.

- DITTRICH, J.-P., SEEGER, B., TAYLOR, D. S., AND WIDMAYER, P. 2002. Progressive merge join: A generic and non-blocking sort-based join Algorithm. *VLDB Conference* 299–310.
- DITTRICH, J.-P., SEEGER, B., TAYLOR, D. S., AND WIDMAYER, P. 2003. On producing join results early. *PODS Conference* 134–142.
- DOBRA, A. 2005. Histograms revisited: When are histograms the best approximation method for aggregates over joins? *PODS Conference* 228–237.
- DOBRA, A., GAROFALAKIS, M. N., GEHRKE, J., AND RASTOGI, R. 2002. Processing complex aggregate queries over data streams. *SIGMOD Conference* 61–72.
- GANGULY, S., GIBBONS, P. B., MATIAS, Y., AND SILBERSCHATZ, A. 1996. Bifocal sampling for skew-resistant join size estimation. *SIGMOD Conference* 271–281.
- HAAS, P. J. 1997. Large-sample and deterministic confidence intervals for online aggregation. *SSDBM Conference* 51–63.
- HAAS, P. J., NAUGHTON, J. F., SESHADRI, S., AND SWAMI, A. N. 1996. Selectivity and cost estimation for joins based on random sampling. *J. Com. Syst. Sci.* 52, 3, 550–569.
- HAAS, P. J. AND HELLERSTEIN, J. M. 1999. Ripple joins for online aggregation. *SIGMOD Conference* 287–298.
- HELLERSTEIN, J. M., AVNUR, R., CHOU, A., HIDBER, C., OLSTON, C., RAMAN, V., ROTH, T., AND HAAS, P. J. 1999. Interactive data analysis: The control project. *IEEE Comput.* 32, 8, 51–59.
- HELLERSTEIN, J. M., HAAS, P. J., AND WANG, H. J. 1997. Online aggregation. *SIGMOD Conference*, 171–182.
- HOU, W.-C., ÖZSOYOĞLU, G., AND TANEJA, B. K. 1988. Statistical estimators for relational algebra expressions. *PODS Conference* 276–287.
- HOU, W.-C., ÖZSOYOĞLU, G., AND TANEJA, B. K. 1989. Processing aggregate relational queries with hard time constraints. *SIGMOD Conference* 68–77.
- HOU, W.-C. AND ÖZSOYOĞLU, G. 1991. Statistical estimators for aggregate relational algebra queries. *ACM Trans. Datab. Syst.* 16, 4, 600–654.
- IVES, Z. G., FLORESCU, D., FRIEDMAN, M., LEVY, A. Y., AND WELD, D. S. 1999. An adaptive query execution system for data integration. *SIGMOD Conference* 299–310.
- JERMAINE, C., DOBRA, A., ARUMUGAM, S., POL, A., AND JOSHI, S. 2005. A disk-based join with probabilistic guarantees. *SIGMOD Conference* 587–598.
- KOOI, R. P. 1980. The optimization of queries in relational databases, PhD thesis, CWR University.
- LIPTON, R. J. AND NAUGHTON, J. F. 1990. Query size estimation by adaptive sampling. *PODS Conference* 40–46.
- LIPTON, R. J., NAUGHTON, J. F., AND SCHNERDEN, D. A. 1990. Practical selectivity estimation through adaptive sampling. *SIGMOD Conference* 1–11.
- LUO, G., NAUGHTON, J. F., AND ELLMANN, C. 2002. A non-blocking parallel spatial join algorithm. *ICDE Conference* 697–705.
- LUO, G., ELLMANN, C., HAAS, P. J., AND NAUGHTON, J. F. 2002a. A scalable hash ripple join algorithm. *SIGMOD Conference* 252–262.
- LUO, G., ELLMANN, C., HAAS, P. J., AND NAUGHTON, J. F. 2002b. A scalable hash ripple join algorithm. *SIGMOD Conference* 252–262.
- MOKBEL, M. F., LU, M., AND ARET, W. G. 2004. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. *ICDE Conference* 251–263.
- OLKEN, F. 1993. Raridom sampling from databases. PhD thesis, University of California, Berkeley, CA.
- SHAO, J. 1999. *Mathematical Statistics*, Springer-Verlag.
- SHAPIRO, L. D. 1986. Join processing in database systems with large main memories. *ACM Trans. Datab. Syst.* 11, 3, 239–264.
- URHAN, T. AND FRANKLIN, M. J. 2000. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.* 23, 2, 27–33.

Received July 2005; revised May 2006; accepted August 2006