Scalable Approximate Query Processing With The DBO Engine

Christopher Jermaine, Subramanian Arumugam, Abhijit Pol, Alin Dobra
CISE Department, University of Florida
Gainesville, FL, USA
{cjermain, sa2, apol, adobra}@cise.ufl.edu

ABSTRACT

This paper describes query processing in the DBO database system. Like other database systems designed for ad-hoc, analytic processing, DBO is able to compute the exact answer to queries over a large relational database in a scalable fashion. Unlike any other system designed for analytic processing, DBO can constantly maintain a guess as to the final answer to an aggregate query throughout execution, along with statistically meaningful bounds for the guess's accuracy. As DBO gathers more and more information, the guess gets more and more accurate, until it is 100% accurate as the query is completed. This allows users to stop the execution at any time that they are happy with the query accuracy, and encourages exploratory data analysis.

Categories and Subject Descriptors

G3 [Probability and Statistics]: Probabilistic Algorithms; H.2.4 [Database Management - Systems]: Query Processing

General Terms

Algorithms, Performance

Keywords

Sampling, Online Aggregation, Randomized Algorithms, DBO

1 INTRODUCTION

Modern database systems are ill-suited to the task of ad-hoc, analytic query processing over massive data sets. For proof of this, one needs only to look at the TPC-H benchmark results, which show that modern hardware and software can still provide dismal, day-long query evaluation times given an ad-hoc analytic processing workload. Such slow speeds render interactive, exploratory data processing an impossibility.

One way to address this performance limitation is to redesign database architecture from the ground up to support intense, analytic workloads. A promising idea is to make *randomization* the basic database design principle [11]. Under such a paradigm, a database relies on randomized algorithms that immediately give an approximate and statistically meaningful guess as to the eventual query result. If the user is satisfied with the accuracy, or the guess shows that the question will likely have an uninteresting answer, then the computation can be terminated. However, if the query is allowed to run, the guess becomes more and more accurate as the database system processes more data. If necessary, the user may simply decide to wait until an exact answer is obtained.

This paper describes the design and implementation of the query processing engine of a prototype database system based on such a design, called *Database-Online* or *DBO*. DBO takes as input a `SELECT-FROM-WHERE-GROUP BY` aggregate SQL query over a number of disk-based, input tables. Like a traditional database system, DBO computes the exact answer to the query in a scalable fashion. However, DBO is designed to make use of novel, randomized algorithms that not only allow it to compute the exact answer to the query, but also allow it to maintain a guess (with accuracy guarantees) as to the final answer to the query at all times.

DBO demonstrates that by modifying certain basic principles of database system design, it is possible to have the best of both worlds: a database system that can process large data sets efficiently, but also supports interactive data exploration through fast and accurate approximation.

An Unsolved Problem: Scalable Online Approximation

The design and implementation of such a system presents a challenging set of research problems. Hellerstein, Haas, and Wang first proposed an idea along these lines in their 1997 paper describing *online aggregation* [11], and later showed how to evaluate joins so as to give accuracy guarantees during query execution (with the introduction of the *ripple join* [6]). This work was later extended to a parallel environment [15]. However, a problem with this work is that *the proposed algorithms are not scalable*. As soon as enough data has been processed that they cannot all be stored in main memory, it is no longer possible to provide statistical guarantees. This is a significant limitation of this early work. As we show experimentally, available memory may be consumed after only a few seconds, and yet the accuracy may still not be acceptable.

In response to this, Jermaine et al. [13] showed how to make online estimation scalable, and described a generalization of the ripple join called the *SMS join* that gives an estimate with statistical accuracy guarantees from startup through completion. However, a problem with the SMS join is that *it is generally only appropriate for joins over two large input tables*. This is problematic because the greater the number of input tables, the more difficult it is to produce an accurate, approximate answer quickly. Each additional input table typically increases the inaccuracy of the obvious estimators in a multiplicative fashion. Thus, the more input tables to a query, the more likely that a scalable algorithm will be required to process enough data to give an accurate answer.

Our Contributions

DBO specifically addresses these limitations and demonstrates a new paradigm for analytic processing. DBO is able to complete the

Material in this paper is based upon work supported by the US National Science Foundation under Grant Nos IIS-0347408 and IIS-0612170.

SIGMOD '07, June 12-14, 2007, Beijing, China.

answer to *arbitrary* select-project-join query plans in a scalable fashion, and can provide for statistical guarantees from startup through completion. There are many technical innovations in DBO's query processing engine, including: (1) a redesign of the traditional query processing engine to facilitate information sharing across relational operations; (2) a novel scheme for producing join tuples in a randomized fashion that facilitates statistical guarantees; (3) a deep mathematical analysis of the engine's statistical properties that generalizes existing analysis [6][13] to different types of randomization and queries; and (4) derivation of unbiased estimators for estimate quality that allow analysis of queries over arbitrary numbers of tables.

2 WHY IS THIS HARD?

The problem of combining scalability and online estimation is difficult. In order to achieve scalability, a database system must rely on careful movement of data between memory and disk. On the other hand, in order to perform statistical inference, a system must rely on randomization. Obviously, these requirements are in direct opposition to one another: How is it possible to achieve careful organization and randomization at the same time? In this Section, we discuss these difficulties in more detail.

2.1 The Ripple Join

The most well-known algorithm for performing online estimation over multi-table queries is the *ripple join* family of algorithms [6]. In our discussion of the ripple join (and of all of the algorithms considered in this paper), we assume a TPC-H-style query of the form:

```
SELECT n.name, SUM (...)
FROM customer c, orders o, lineitem l,
    supplier s, nation n, region r
WHERE c.custkey = o.custkey AND
  l.orderkey = o.orderkey AND ...
GROUP BY n.name
```

Or more generally:

SELECT SUM ($f(r_1 \bullet r_2 \bullet \ldots \bullet r_n)$)
FROM R_1 as r_1, R_1 as r_2, . . . , R_n as r_n

In the above expression, $\bullet$ is the concatenation operation, which appends one tuple to another. f can encode any relational selection or join predicate over the input tuples, and can also encode a GROUP BY by selecting tuples only from a specific group.

The ripple join works by reading an ever-larger sample of each input relation in a sequential fashion, and using those samples to estimate the final query answer. As the sample grows, the algorithm outputs estimates of ever-increasing accuracy. The fact that the portion of the data space used to compute the estimate grows from the lower left to upper right corner of the data space leads to the name "ripple join".

However, the algorithm is not scalable. Assuming a hash ripple join, at the point that the join can no longer buffer all of the sampled records in main memory, it becomes necessary to page out one or more records to make room for a new record, and to page in other records in order to check for matches with the new record. These I/Os will be random due to the random order of input tuples, so the algorithm causes severe thrashing. Even if each new record that is processed requires only a single random disk I/O, the processing rate will be only around 10,000 records/minute/disk (with a 3ms random I/O time), with no easy way to address the problem.

2.2 The SMS Join

In response to this, Jermaine et al. proposed the *sort-merge-shrink join*, a scalable join that is able to maintain online, statistical estimates throughout query execution [13]. The SMS join is closely related to the classic sort-merge join [19], except that during the sort phase of the SMS join, all of the input relations are processed concurrently in order to provide a guess as to the final query result. Unfortunately, the SMS join has problems scaling past two relations. Imagine that we want to answer the query:

SELECT SUM (R_3.c)
FROM R_1, R_2, R_3
WHERE R_1.a = R_2.a AND R_2.b = R_3.b

Like the SMS join, virtually all modern, scalable join algorithms use a two-phase model, where data are first hashed or sorted into buckets or runs and written back to disk. In a second phase, the various buckets or runs are joined. In the above query, it is impossible to use such a two-phase algorithm to compute the answer. If data from R_2 are sorted or hashed on attribute R_2.a, then the resulting buckets or runs cannot be joined directly with R_3 without re-sorting or re-hashing (because the join with R_3 is on the attribute R_2.b and the records will be sorted on the wrong attribute). If the data from R_2 are sorted or hashed on attribute R_2.b, then R_2 cannot be joined directly with R_1. Such a query must be implemented using two *separate* joins, and it is far from clear how two joins can be combined in the SMS framework.

2.3 Fixing the Problem?

Unfortunately, all obvious ideas for addressing this problem encounter difficulties. One idea is to use some sort of scalable "fast first" join algorithm [4][5] to process $R_1 \bowtie R_2$, and to pipeline result tuples from the first join into a second SMS join with R_3. However, there are problems associated with this approach. For example, almost any method for estimating the final query result will require a random input ordering of tuples in order to provide statistical guarantees. However, the output from the first join will not have a randomized ordering, making the statistical properties of such an algorithm very difficult to reason about. It is known that producing such a randomized ordering is difficult [2]. Even if tuples *were* produced in a randomized fashion, it is difficult to pipeline them into another join and use that join to produce an estimate for the answer to the query due to important, unknown constants. For example, a ripple-join-style estimator would need to know the size of the intermediate relation, which would be unknown until the relation is materialized.

3 DBO QUERY EVALUATION: OVERVIEW

Because of such difficulties, designing a database system that provides both scalability and accurate estimation from startup through completion is a daunting task. It appears to be impossible to achieve both goals by simply plugging algorithms directly into a traditional database engine; more fundamental design changes are needed. The remainder of the paper describes query processing in the DBO system, which achieves these goals by making use of some fundamental changes in database system architecture.

The problem with traditional database engines in this context lies with the fact that relational operations are treated as "black boxes". This abstraction renders accurate statistical estimation impossible *because it hides intermediate results as well as internal state from the remainder of the system*. If intermediate results are not externally visible, it is impossible for the system to guess the final

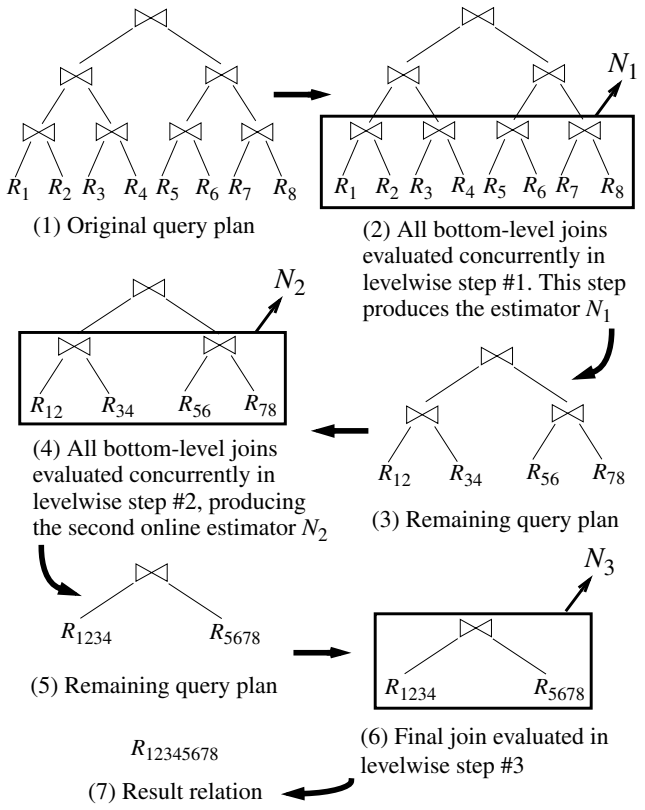

Figure 1: Levelwise query evaluation in DBO.

answer to the query *because no entity has access to information about every input relation*.

In order to provide for accurate online estimation, DBO's execution engine is quite different. All of the operations at a single level of the query plan are *taken together* as the basic query-processing abstraction. The operations executed at a single level in the query plan are together referred to as a *levelwise step*. All of the operations within each levelwise step execute concurrently and share information with one another. The reason for this is simple: assuming for the time being that all leaves of the query plan are at the same level, then by definition, all of the operations at a single level of the query plan have access to enough information to compute the final answer to the query. Actually computing the final answer may take hours or days. But by carefully allowing each operation to share some of its intermediate results with all of the other operations at the same level, it becomes possible to look for preliminary result tuples in order to guess the final query answer.

The process of evaluating a query from startup through completion in DBO for a given query plan is depicted in Figure 1. In this example, DBO's engine begins by executing the first levelwise step, where each operation at the bottom level of the plan is evaluated concurrently. At all times, this step maintains an online estimator N_1 for the final answer to the query by passing information among the various constituent joins. As the levelwise step progresses, N_1 achieves more and more accuracy. Eventually, it becomes frozen as the step completes. The resulting relations are used as input to the second levelwise step, which produces an online estimator N_2. At all times, N_2 is combined with N_1 to produce a single estimate for the final answer to the query. Finally, after the second levelwise step ends, the last levelwise step is executed, which produces an online estimator N_3. Again, as this step progresses, N_3 is combined with both N_1 and N_2 (now both frozen) to produce an estimate for the answer to the query. As the end of query execution approaches, N_3 will approach (and eventually become equal to) the correct query result.

4 THE LEVELWISE STEP

As described above, all of the joins at the *i*th level of the query plan are evaluated concurrently in DBO, and all of the joins that are concurrently executed are collectively referred to as a *levelwise step*. The concurrent evaluation is necessary in order to provide a running estimator for the eventual answer to the query throughout execution, since it ensures that at least some information about every relation is always in memory.

In the DBO prototype, each individual join is implemented as a modified sort-merge join, though use of a sort-merge join is not a fundamental requirement. It would also be possible to modify other scalable, two-phase join algorithms for use (see the discussion in Section 10), though this is beyond the scope of the paper.

Whatever two-phase join algorithm is used, the joins that make up a levelwise step must be carefully coordinated to share information among one another so that an estimate for the final answer to the query can be maintained. This results in the partitioning of a levelwise step into two phases: a *scan* phase and a *merge* phase. These two phases are described now in the context of the sort-merge join employed by the DBO prototype.

4.1 The Scan Phase

The *scan phase* of a levelwise step is analogous to the sort phase of a sort-merge join or the hash phase of a hash join except that the phase is executed concurrently for *all* of the joins that make up the *i*th levelwise step. There are several other key characteristics of the scan phase of a levelwise step:

(1) *Immediate discovery of output tuples*. In a manner similar to the ripple join [6], at all times, the subsets of tuples stored in memory from all relations are checked to see if they can be joined to discover any output tuples immediately, which are then used to guess the eventual answer to the query.

(2) *Randomized sort order.* In order to ensure that the statistical properties of the estimate produced by examining those output tuples are reasonable, the tuples must be input in a randomized order. As we discuss below, this also implies the *output* of the scan phase must be in a randomized order.

(3) *Round-robin processing of runs*. In order to provide for greater accuracy, runs are processed in a carefully choreographed, round-robin fashion. This round-robin processing of runs leads to a "zig-zag" pattern that allows the algorithm to produce a low-variance estimator, as we will discuss subsequently.

The scan phase of a levelwise step is implemented as follows:

(1) As the phase begins, one run of records from each relation is read into memory from disk (or, since levels are pipelined, the records are taken as input directly from the previous levelwise step). Once one run from each input relation is present in memory, all of the records are immediately joined in order to search for any result tuples. As is described in Section 5, these result tuples are used to obtain an unbiased guess for the query answer.

(3) Assuming an arbitrary ordering for the input relations, in the next step, the run from the "first" relation is sorted and written back to disk, just as in a sort-merge join. However, there is one important difference. If R_j is to be joined on the attribute R_j.key,

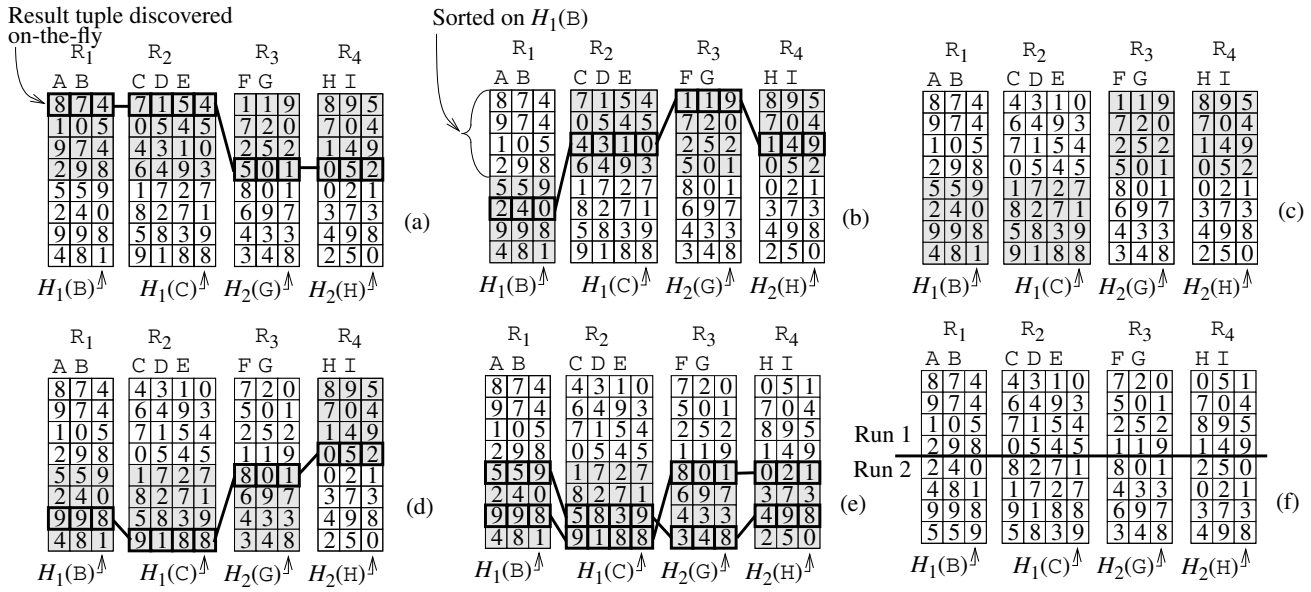

Figure 2: Scan phase of a levelwise step. In this example, we assume an SQL query having the where clause "`WHERE` R_1`.B =` R_2`.C AND` R_2`.E =` R_3`.F AND` R_3`.G =` R_4`.H`", and we assume that the first levelwise step computes the joins $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$. In the scan phase, a run from each input relation is first read into memory. In our example, we have enough memory to hold four tuples from each relation, and the in-memory tuples are shaded. Next, these runs are immediately searched for any tuples that match the final `WHERE` clause, and any such tuples are immediately used to estimate the answer to the query (a). Then, in round-robin fashion, in-memory runs are sorted based on a hash function associated with each join (H_1 for $R_1 \bowtie R_2$ and H_2 for $R_3 \bowtie R_4$) and written to disk; after a run is written to disk, it is immediately replenished with the next run from the appropriate input relation (b)-(e). At all times, any discovered tuples that match the final `WHERE` clause are used to help estimate the final query result. The process is repeated until all input relations have been broken into runs and sorted using the hash function (f).

it is *not sorted on* R_j.key *directly*. Rather, it is sorted on the value of $H(id, R_j.\text{key})$, where H is a randomizing or hash function that takes the value *id* as a seed; in order to make sure that none of the orderings are correlated, a different seed is used for each join. This hashing is performed so it is possible to guarantee a random output order for tuples from the subsequent merge phase: if the sort order is chosen based upon some randomized lexicographic ordering of the input tuples, tuples will be output in an order that is statistically independent of all of the output records' attributes (except for the join attribute). This randomized output order means that the output tuples can then be used as input to a join in the *next* levelwise step.

(3) After the records from the first run of the first relation are written back to disk, the next set of records from the first relation are read from disk (or taken directly from the previous merge phase if pipelined). They are immediately joined with all of the other tuples currently in memory. Then, the first set of records from the *second* relation is sorted using H and written out to disk to make room for the second run of records from the second relation. This second run is read in and immediately joined with all of the other tuples in memory. This processing is always performed in a systematic, round-robin fashion: first a run from R_1 is read and processed, then a run from R_2 is read and processed, and so on; after a run from the last relation to be joined at level i is read, the next run from relation R_1 is read, and the cycle begins again. An example is depicted in Figure 2.

4.2 The Merge Phase

The merge phase of the joins making up the ith levelwise step is very similar to the merge phase of a traditional sort-merge join, except that the various merges of all of the joins are run concurrently. That is, the head of each run of each input relation to the i^{th} levelwise step is read into memory, and runs of records from each output relation are produced in a round-robin fashion in order to pipeline the result records directly into the scan phase of the joins making up the $(i + 1)^{th}$ levelwise step, without ever writing records back to disk. Since the scan phases of the joins making up the $(i + 1)^{th}$ levelwise step all run concurrently, so must the merge phases of the i^{th} levelwise step. An example merge phase (continuing the example of Figure 2) is depicted in Figure 3.

5 SCAN PHASE ESTIMATION IN DEPTH

The previous Section described at a high level the algorithm for computing a levelwise step. In this Section, we discuss in detail how to compute online estimates in DBO.

5.1 Estimating the First Time Around

As described previously, a key goal of the scan phase of each levelwise step is to use result tuples discovered on-the-fly to estimate the final answer to the query. In the remainder of this Subsection, we make the assumption that each input relation R_i for i from 1 to n is fully materialized and resides on disk. However, this is only the case in the first levelwise step. In subsequent levelwise steps, the result of the merge phase from the previous step is pipelined into the scan phase. We consider the extension to the pipelined case in the next Subsection.

Let $T(i, j, k) = R_{j,i} \times R_{j+1,i} \times \ldots \times R_{k,i}$ for a given levelwise step. In other words, $T(i, j, k)$ is the cross product of all of the tuples in the ith run of input relations j through k. For example, in Figure 2, $T(1, 1, 4)$ is the cross product of all of the tuples in mem-

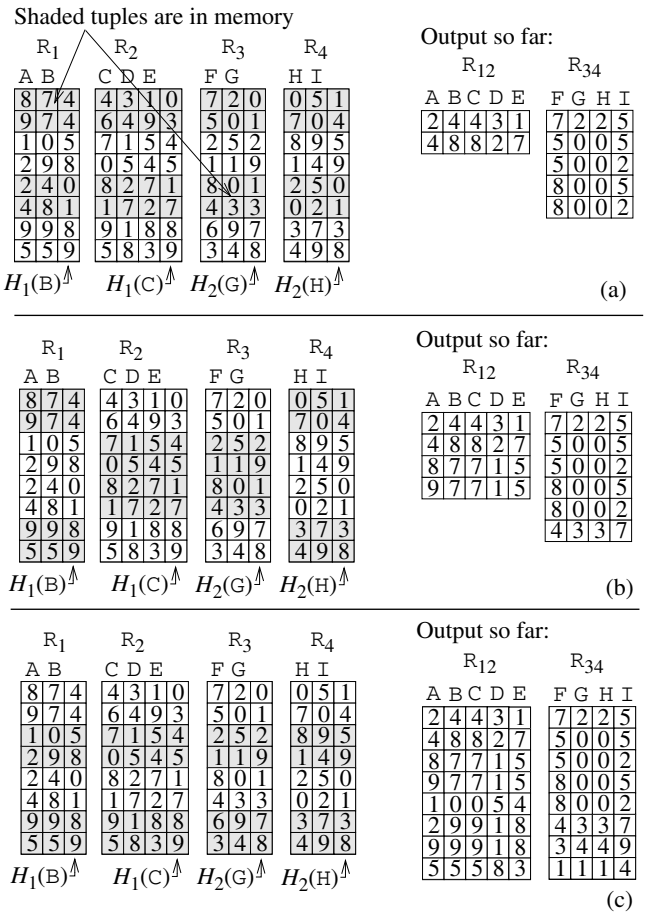

Figure 3: The merge phase of a levelwise step used to compute $R_{12} \leftarrow R_1 \bowtie R_2$ and $R_{34} \leftarrow R_3 \bowtie R_4$ for a query with the `WHERE` clause "`WHERE R1.B = R2.C AND R2.E = R3.F AND R3.G = R4.H`". First, the head of each run produced by the levelwise step's scan phase is read into memory, and all of the in-memory records are joined (a). Note that because tuple processing order is defined by the hash functions H_1 and H_2 associated with $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$, respectively, the output order of tuples to R_{12} and R_{34} is random and independent, except for the clustering of tuples having an identical join key. This allows the output of R_{12} and R_{34} to be pipelined into the scan phase of the *next* levelwise step. When any run's in-memory tuples are exhausted, the next set of tuples is read from disk and joined with those in memory (b). The process is repeated until all of the level's joins have been completed (c).

ory in step (a); since there are four runs in memory and each run has four tuples, $T(1, 1, 4)$ will contain 256 tuples in all. Thus, after r runs have been processed from each relation by a levelwise step, the following is equivalent to the sum of the aggregate function f over all tuples that have been discovered:

$$\alpha = \sum_{a=1}^{r} \left[\sum_{t \in T(a,1,n)} f(t) \right] + \sum_{a=2}^{r} \sum_{b=1}^{n-1} \left[\sum_{t_1 \in T(a,1,b)} \left[\sum_{t_2 \in T(a-1,b+1,n)} f(t_1 \bullet t_2) \right] \right]$$

Since it is equi-probable that any given tuple may be discovered during the scan phase, by simply scaling up, this summation can easily be used to calculate an unbiased guess as to the final answer to the query. Let β be the ratio of the size of the overall data space to the number of tuples considered by the scan phase; that is:

$$\beta = \frac{\left| R_1 \times R_2 \times \ldots \times R_n \right|}{\sum_{a=1}^{p} |T(a,1,n)| + \sum_{a=2}^{p} \sum_{b=1}^{n-1} |T(a,1,b)||T(a-1,b+1,n)|}$$

Then $\alpha\beta$ is an unbiased estimator for the final answer to the query (see the Appendix of the paper for a proof). In the remainder of the paper, we will use the notation N_i to denote the estimator associated with the scan phase of the ith levelwise step.

In general, it is not enough to be able to give an estimate; it is also vital that we be able to characterize the accuracy of the estimate. This characterization via a derivation of the variance of N_i (denoted $\sigma^2(N_i)$) is considered in Sections 5.5 and 7.

5.2 Estimation at Subsequent Levels

The estimation procedure for levelwise steps other than the first one differs for two reasons. First, intermediate tuples are produced by a merge phase only in semi-random order; tuples with the same join key are produced all at one time in a group. For example, consider Figure 3(a); all tuples with join key 0 appear at the same time in relation R_{34}. Second, cardinalities of intermediate input relations are not known as an intermediate levelwise step is computed. This is because the levelwise steps are pipelined: the results from the merge phase of the i^{th} step are used immediately by the scan phase of the $(i + 1)^{th}$ step, before the input relation has been fully materialized. The estimation and variance computation procedures must take into account these properties.

To handle the grouping problem, we use a variation on the idea proposed by Haas to remove the correlation induced when sampling blocks of tuples rather than tuples [7]. We view each group of tuples that all have the same join key and have all been produced by the same merge phase as a *single*, *indivisible* output tuple, which we subsequently refer to as a "clump". Imagine that tuples t_1, t_2, ..., t_n from intermediate relations 1 through n are actually "clumps" or *sets* of tuples, where all $t_i' \in t_i$ have the same join key. Then to compute $f(t_1 \bullet t_2 \bullet \ldots \bullet t_n)$ during both the estimation and variance computation process, we simply use:

$$f(t_1 \bullet t_2 \bullet \ldots \bullet t_n) = \sum_{t_1' \in t_1} \sum_{t_2' \in t_2} \ldots \sum_{t_n' \in t_n} f(t_1' \bullet t_2' \bullet \ldots \bullet t_n')$$

This removes any correlation induced due to the grouping and the clumps are, in fact, produced in random order.

To handle the fact that we do not know the size of the intermediate relations, we note that the tuples output from a join will appear in sorted order, based on the result hash function H(`R.a`). Rather than choosing the size of each run beforehand, we choose the number of runs (or partitions) p and break the output space of H into p contiguous, (approximately) equi-sized ranges of key values. For example, if the range of H(`R.a`) is from 0 to $(2^{31} - 1)$, then we might break the range of H into [0 to $(2^{29} - 1)$], [2^{29} to $(2^{30} - 1)$], [2^{30} to $(2^{29} + 2^{30} - 1)$], and [$(2^{29} + 2^{30})$ to $(2^{31} - 1)$] if $p = 4$. Assuming equi-sized ranges, each "clump" of output tuples produced by the join then has a probability of $1/p$ of falling into a given run, and the sampling performed at all levels except for the first is Bernoulli or "coin-flip" sampling. As a result, the unknown size of the relation is unimportant, since we can scale up any estimate produced using the records in memory by p^n to obtain an unbiased estimate for the eventual query result (this is because we

have a $1/p$ sample of each of the n relations that are input into the levelwise step). Since the summation α used to compute N_i contains $1 + (r - 1)n$ estimates (see Section 6.1), the scaling factor:

$$\beta = \frac{p^n}{1 + (r-1)n}$$

can then be used to produce an unbiased N_i for any scan phase receiving pipelined input tuples.

5.3 Estimation At the Last Level

Tuples output from the join in the very last levelwise step are used as input into a final estimator N_{d+1}, where d is the depth of the query plan. N_{d+1} is computed in exactly the same way as the estimator described in the previous Section. The tuples output from the final join are broken into p partitions, and after p partitions have been processed, the sum α of all tuples discovered is multiplied by $\beta = \frac{p^n}{1 + (r-1)n}$ (which is equivalent to p/r, since $n = 1$ in the case of the final join).

5.4 Estimating the Final Answer to the Query

An unbiased statistical estimator for the eventual answer to the query is associated with the level, and maintained online. The random variable N_1 characterizes the statistical estimator associated with the scan phase bottom-most level of the query plan, and N_d is associated with the tuples output from the merge phase of the topmost levelwise step (that is, we have a query plan that is d levels deep). Thus, at any given moment, there are a number of estimators available, one associated with each level of the plan. Each gives an independent estimate for the final query answer.

Since there are $d + 1$ estimators in all (one associated with each level in the query plan, plus one for the final output) they must be combined to form a single estimate for the final result of the query. Since each N_i is unbiased (see Section 7), it follows that for $\{w_1, \ldots, w_{d+1}\}$ where $\sum_{i=1}^{d+1} w_i = 1$, the following is an unbiased estimate for the final answer to the query:

$$N = \sum_{i=1}^{d+1} w_i N_i$$

Furthermore, since DBO's query evaluation engine computes each N_i so that they are all statistically independent (since each level uses an independent random ordering), it is the case that:

$$\sigma^2(N) = \sum_{i=1}^{d+1} w_i^2 \sigma^2(N_i)$$

In order to minimize the error associated with N, we seek to minimize the variance of N over all possible weights. It can easily be shown using Lagrangian multipliers that $\sigma^2(N)$ is minimized (and hence the accuracy of N is maximized) by choosing:

$$w_i = 1 \Big/ \left\{ \sigma^2(N_i) \sum_{j=1}^{d+1} \frac{1}{\sigma^2(N_j)} \right\}$$

5.5 Providing Confidence Bounds

Once the value of N and $\sigma^2(N)$ have been computed, it is then an easy matter to associate confidence bounds with the quality of the estimate of N using standard techniques [3], such as assuming that N is normally distributed (justified by the central limit theorem (CLT) [20]) or by using distribution-free bounds such as those provided by Chebyshev's inequality [9]. In our implementation, we use CLT bounds, which, as we show in Section 8, seem to give good results. However, what we have ignored thus far is how to compute (or estimate) $\sigma^2(N_i)$ for any given i. If $i = d$, then after r of p partitions have been processed, the variance is computed as $\sigma^2(N_d) =$

$$E[N_d^2] - E^2[N_d] = E\left[\left(\sum_j \frac{p}{r} X_j f(t_j)\right)^2\right] - E^2\left[\frac{p}{r}\sum_j X_j f(t_j)\right]$$

where X_j is a zero/one random variable indicating whether the jth tuple in the topmost relation of the query plan has been found in any of the first r partitions produced by the DBO engine, and t_j is the jth result tuple. Note that $E[X_i] = r / p$, and using the "clumping" strategy of Section 5.2, each X_i, X_j pair is independent and so $E[X_i X_j] = r^2 / p^2$. Then simplifying $\sigma^2(N_d)$, we have:

$$\sigma^2(N_d) = \frac{p^2}{r^2}\left(\sum_j \frac{r}{p} f^2(t_j) + \sum_{j \neq k} \frac{r^2}{p^2} f(t_j) f(t_k)\right) - \frac{p^2}{r^2}\left(\sum_j \frac{r}{p} f(t_j)\right)^2$$

$$= \frac{p^2}{r^2}\left(\sum_j \frac{r}{p} f^2(t_j) - \sum_j \frac{r^2}{p^2} f^2(t_j)\right) = \frac{p}{r}\left(1 - \frac{r}{p}\right)\sum_j f^2(t_j)$$

This value can easily be estimated by simply taking the sum of the square of the aggregate function f applied to each result tuple that has been found thus far, and multiplying the result by $\frac{p^2}{r^2}\left(1 - \frac{r}{p}\right)$ to account for the fact that we have (on expectation) seen r/p of the tuples of the final result relation. Estimating $\sigma^2(N_i)$ for $i \neq d$ is more complicated, and left to Section 7.

6 ADDITIONAL CONSIDERATIONS

6.1 Why Use the Round-Robin Approach?

Recall that the scan phase cycles through the relations. For every relation, the current run is written back to disk, the next run is read in, and the query result is re-estimated. This approach can deliver very high estimation accuracy. The reason is that given n input relations each broken into p partitions or runs, it is possible to search a fraction $(1 + n(p-1))/p^n$ of the data space during the scan phase. For example, consider Figure 4 above, where a levelwise step is computed over three input relations, each broken into three runs. In total, the round-robin approach searches $1 + n(p-1)$ or 7 combinations of runs for result tuples, out of 27 combinations total. This is due to the fact that there is one combination that makes use of the first run from the first relation; there are then n different combinations that make use of each of the second through p^{th} runs of the first relation. On the other hand, if we had searched for result tuples only after a new run had been read from *every* input relation (as the SMS join does), we would have considered only three combinations of runs.

6.2 Choosing the Number of Runs

One problem is how to choose p. The goal is to choose the smallest number of runs possible, because the fewer the number of runs, the more tuples in memory at any given instant, and the better the estimation accuracy. Since one run from each relation must fit into memory, in the first levelwise step, p is chosen by summing each input relation's size, and dividing by the available main memory.

At subsequent levels, choosing p is more difficult because the input relations are not materialized before they are processed, so the size of each input relation is unknown. To handle this, the scan phase at each levelwise step other than the first begins by reading

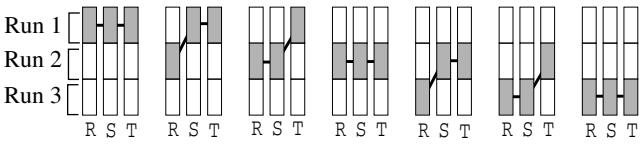

Figure 4: Using the round-robin method, seven combinations of runs are considered when relations `R`, `S`, and `T` (each broken into three runs) are sorted during the scan phase of a levelwise step.

tuples from each input relation so that the range of the hash function H is processed at a constant rate for each run. That is, if there are two relations to be processed, the scan phase should complete the processing of the first k% of H's range for both input relations at roughly the same time, for every k. At the point that the available main memory is (almost) consumed, p is chosen to be $\lfloor 1/k \rfloor$ for the remainder of the levelwise step. Because the set of records from each relation that appears in the first k% of H's range is a k% Bernoulli sample (without replacement) of each relation, and each subsequent k% of H's range is also a k% Bernoulli sample of each relation, their corresponding runs will be (roughly) the same size. Thus, if the first run from each relation fits into memory, the second is likely to as well.

6.3 Handling Data Skew

Like any database system relying on hashing, the DBO system is sensitive to data skew. There are two consequences of this. First, using the method from 6.2 for choosing p, if we are very aggressive and choose a small p, then subsequent partitions may be too large to fit into memory. If this happens, we freeze N_i (as well as its variance estimate) for the remainder of the current scan phase, and run the offending scan phase just as one would run the sort phase of a set of classical, sort-merge join. After the problem scan phase completes, the remainder of the levelwise steps can be executed normally, and updates to the estimates resume. The cost of this is a temporary freeze in updates to the DBO system's estimates.

Second, skew in join key values can also affect the accuracy of the resulting estimator. Fortunately, as long as the method for handling "clumping" from Section 5.2 and the variance estimation methods from Section 7 are used, the DBO system will take into account this drop in accuracy and still report correct confidence bounds; they will simply be wider than if there had been no skew.

6.4 Handling `GROUP BY`s and Other Aggregates

`GROUP BY` queries can trivially be handled within the DBO framework by using a separate "query" for each group. All of these queries can be run concurrently with little additional overhead. A relational selection predicate that accepts only tuples belonging to a given group is added to each query. Other aggregate functions such as `AVERAGE` and `STD_DEV` can also be handled easily, since these are simply functions of multiple `SUM` queries. For example, `AVERAGE` is the ratio of a `SUM` and a `COUNT` (which is itself a `SUM` query).

6.5 Handling Inconvenient Queries

Thus far, we have assumed that DBO's query processing engine is always used to process queries that have been compiled into a bushy query plan. The reason for this assumption is that unless a levelwise step is able to access a random subset of the records from *each one* of the input relations (or at least access temporary relations that contain records derived from each of the input relations), the engine cannot provide for an early guess as to the query result.

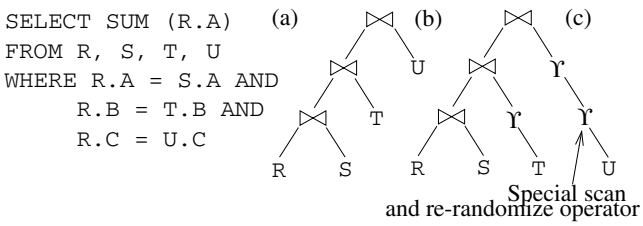

Figure 5: Handling a star-join query. The example query (a) would typically be evaluated using a left- or right-deep plan in a traditional system (b). In DBO, the plan must be augmented with three additional table-scan operations to ensure access to all input data at each levelwise step (c).

However, a query may be processed that cannot be compiled into a bushy tree. This may happen, for example, when a fact table is joined with several smaller, dimension tables. Consider the query of Figure 5(a). If we wish to avoid materializing the result of a cross product, the only plans for this query are linear and non-bushy, because relations `S`, `T`, and `U` must all be joined with `R`.

There are several tactics for dealing with this. In our prototype, we require a scan and re-randomization of *all* of the relations that are "active" during a given levelwise step. This is depicted in Figure 5(b) and (c), where the "normal" query plan for the SQL query of Figure 5(a) has been augmented with three additional operations that do nothing more than read the input tables and write them out in a re-randomized order. The re-randomization is required so that estimators associated with subsequent levelwise steps are not correlated. The result is that `R`, `S`, `T`, and `U` all take part in the scan phase of the first levelwise step, even though `T` and `U` are not joined in this first levelwise step. Their tuples are all read in concurrently, just as in the scan phase depicted in Figure 5, and any result tuples that are discovered are immediately used to produce an estimate.

The obvious cost associated with this technique is that the input relation `T` is processed multiple times. However in practice, this may make little difference. First, this situation is encountered most often in a "star-join" scenario where the table that is repeatedly joined is much, much larger than the others, such as when it is a warehouse fact table. In this case, the additional cost of scanning one or more dimension tables more times than are needed may be negligible. Second, it will often be possible to ensure that the initial scans of relations like `T` and `U` are not wasted, by combining the scans with projection or selection operations found in the query that can substantially reduce the table size.

7 STATISTICAL CONSIDERATIONS

This Section gives a formal, statistical analysis of the estimators associated with each levelwise step, with a particular focus on developing practical, unbiased estimators for $\sigma^2(N_i)$ (that is, the variance of the estimator associated with the ith levelwise step) for the case where $i \neq d + 1$.

7.1 Notation

In this Section we introduce the notation used in Section 7. Let $\mathtt{R}_1, \mathtt{R}_2, \ldots, \mathtt{R}_n$ be the n relations that are the arguments of the aggregate query and let $f(\,)$ be the aggregate function that is summed over each tuple in the cross product of the input relations to obtain the value of the aggregate query, as described in Section 5. We will always use the notation t_i and t_i' to denote tuples from the relation $\mathtt{R}_i$ (that is, if the subscript associated with a tuple is i, then it is assumed that the tuple came from relation i) and we use

the convention that the argument to the aggregate function f can be specified as a set of tuples (one from each relation) in any order; we assume that the ordering and concatenation are performed automatically. We use the notation $\mathtt{R}_i'$ to designate a sample of the relation R differing slightly from the body of the paper.

Following the convention of the paper describing the SMS join [13], we will formally analyze samples from relations by introducing zero/one random variables that indicate whether a tuple belongs to the sample or not. To this end, we will use the notation X_{t_i} to designate the random variable that takes value 1 when $t_i \in \mathtt{R}_i'$ (that is, the tuple t_i is in the sample of $\mathtt{R}_i$), and value 0 otherwise. These random variables allow sums of the form $\sum_{t_i \in \mathtt{R}_i'} f(t_i)$ to be rewritten as $\sum_{t_i \in \mathtt{R}_i} X_{t_i} f(t_i)$.

Since we are dealing with general aggregate queries over joins, the theory will inevitably get complicated. To alleviate this problem we introduce special notation to represent the terms that appear in the analysis. We use $P(n)$ to denote the power set of the set $\{1...n\}$. For a set $S \in P(n)$ we use $\sum_{\{t_i \in \mathtt{R}_i | i \in S\}}$ to denote the multiple sums $\sum_{t_{i_1} \in \mathtt{R}_{i_1}} \sum_{t_{i_2} \in \mathtt{R}_{i_2}} \cdots \sum_{t_{i_k} \in \mathtt{R}_{i_k}}$ where $S = \{i_1, i_2, ..., i_k\}$. With this notation, the aggregate over the cross product we are trying to compute with the DBO system can be written as:

$$\sum_{\{t_i \in \mathtt{R}_i | i \in \{1\ldots n\}\}} f(\{t_i | i \in \{1\ldots n\}\}) = \sum_{t_1 \in \mathtt{R}_1} \sum_{t_2 \in \mathtt{R}_2} \cdots \sum_{t_n \in \mathtt{R}_n} f(t_1 \bullet t_2 \bullet \ldots \bullet t_n)$$

7.2 Analysis of N_1: the First Levelwise Step

We begin with a formal analysis of the bias and variance of the estimator N_1 associated with the first levelwise step. As discussed in Section 5.1, the estimator in the first levelwise step is based on sampling without replacement from the relations (note that a different analysis applies to subsequent steps). We assume that each relation is randomly partitioned into p equi-sized parts. We begin by showing that the estimator described in Section 5.1 is unbiased.

7.2.1 Analysis of Expectation

The idea behind the process used in the scan phase is simple: evaluate the aggregate over the cross-product of the samples and scale up the result to compensate for the difference in size between the samples and relations. In the first levelwise step, every time a new run is loaded into memory, all result tuples present in memory are immediately joined and used to produce such an estimate. N_1 is then essentially an average of all of the estimates that have been produced thus far; since each of these estimates has identical statistical properties, we refer to an arbitrary instance of such as estimate as X. The estimate X can be written formally as:

$$\begin{aligned} X &= p^n \sum_{t_1 \in \mathtt{R}_1'} \sum_{t_2 \in \mathtt{R}_2'} \cdots \sum_{t_n \in \mathtt{R}_n'} f(t_1 \bullet t_2 \bullet \ldots \bullet t_n) \\ &= p^n \sum_{\{t_i \in \mathtt{R}_i' | i \in \{1\ldots n\}\}} f(\{t_i | i \in \{1\ldots n\}\}) \qquad (1) \\ &= p^n \sum_{\{t_i \in \mathtt{R}_i | i \in \{1\ldots n\}\}} \prod_{i=1}^{n} X_{t_i} f(\{t_i | i \in \{1\ldots n\}\}) \end{aligned}$$

In order to analyze X, we first need to specify properties of each X_{t_i}. Since each tuple is sampled independently, the random variables for different values of i are independent, which means that the expectation of the product over X_{t_i} values is the product of expectations. Thus, for any tuples t_i, t_i', we have

$$E[X_{t_i}] = 1/p;\ E[X_{t_i} X_{t_i'}] = \begin{cases} 1/p \text{ if } t_i = t_i' \\ \dfrac{1}{p^2} \dfrac{|\mathtt{R}_i| - p}{|\mathtt{R}_i| - 1} \text{ if } t_i \neq t_i' \end{cases}$$

$$= \delta_{t_i, t_i'} \frac{1}{p} + (1 - \delta_{t_i, t_i'}) \frac{1}{p^2} \frac{|\mathtt{R}_i| - p}{|\mathtt{R}_i| - 1}$$

$$= \frac{1}{p^2 (|\mathtt{R}_i| - 1)} [(|\mathtt{R}_i| - p) + |\mathtt{R}_i| (p-1) \delta_{t_i, t_i'}]$$

where we used the fact that the expectation of a zero-one random variable is equal to the probability that the variable is one. Note that these formulas are versions of the formulas derived in the paper describing the SMS join [13]. $\delta_{t_i, t_i'}$ is the Kronecker delta symbol that is equal to one if $t_i = t_i'$ and zero otherwise; expressing cases using $\delta_{t_i, t_i'}$ transforms an "if" statement into an algebraic expression, and simplifies the analysis.We will use the fact that for any function g, $\sum_j g(i,j) \delta_{ij} = g(i,i)$ since the terms with $i \neq j$ have a zero multiplier. We can now show that X is unbiased:

Theorem 1: *Unbiasedness of X*:

$$E[X] = \sum_{\{t_i \in \mathtt{R}_i | i \in \{1\ldots n\}\}} f(\{t_i | i \in \{1\ldots n\}\})$$

i.e., X is an unbiased estimator for the final answer to the query.

Proof: The proof uses linearity of expectation and the independence of the random variables X_{t_i} for various values of i (i.e. the expectation of products is product of expectations). We then have:

$$\begin{aligned} E[X] &= p^n \sum_{\{t_i \in \mathtt{R}_i | i \in \{1\ldots n\}\}} \prod_{i=1}^{n} E[X_{t_i}] f(\{t_i | i \in \{1\ldots n\}\}) \\ &= \sum_{\{t_i \in \mathtt{R}_i | i \in \{1\ldots n\}\}} f(\{t_i | i \in \{1\ldots n\}\}) \end{aligned}$$

since $\prod_{i=1}^{n} E[X_{t_i}] = \frac{1}{p^n}$. □

7.2.2 Analysis of Variance

We now address the problem of computing the variance of X, denoted by $\sigma^2(X)$. Since $\sigma^2(X) = E[X^2] - E^2[X]$, we need only address the problem of computing $E[X^2]$ since $E[X]$ is given above. We first need the following technical result:

Proposition 1: *For arbitrary values a_i and b_i, $i \in \{1\ldots n\}$:*

$$\sum_{\{t_i \in \mathtt{R}_i | i \in \{1\ldots n\}\}} \sum_{\{t_i' \in \mathtt{R}_i | i \in \{1\ldots n\}\}} \left(\prod_{i \in \{1\ldots n\}} a_i + b_i \delta_{t_i, t_i'} \right) \times f(\{t_i\}) f(\{t_i'\})$$

$$= \sum_{S \in P(n)} \prod_{i \in S^C} a_i \prod_{j \in S} b_j \sum_{\{t_i \in \mathtt{R}_i | i \in S\}} \left[\sum_{\{t_j \in \mathtt{R}_j | j \in S^C\}} f(\{t_i, t_j\}) \right]^2$$

Proof: The main idea is to introduce the following set of functions:

$$F_k(\{t_l, t_l' | (l \in \{k+1\ldots n\})\}) = \sum_{\{t_i \in \mathtt{R}_i | i \in \{1\ldots n\}\}} \sum_{\{t_i' \in \mathtt{R}_i | i \in \{1\ldots n\}\}} \left(\prod_{i \in \{1\ldots n\}} a_i + b_i \delta_{t_i, t_i'} \right) \times f(\{t_i, t_l\}) f(\{t_i', t_l'\})$$

and to show by induction that:

$$F_k(\{t_l, t_l' | (l \in \{k+1\ldots n\})\}) =$$

$$\sum_{S \in P(n)} \prod_{i \in S^C} a_i \prod_{j \in S} b_j \sum_{\{t_i \in \mathrm{R}_i | i \in S\}} \left[\sum_{\{t_j \in \mathrm{R}_j | j \in S^C\}} f(\{t_i, t_j, t_l\}) \right] \times \left[\sum_{\{t_j \in \mathrm{R}_j | j \in S^C\}} f(\{t_i, t_j, t_l'\}) \right]$$

Details of the proof are omitted due to space constraints. □

This now allows us to prove the following result:

Theorem 2: *Second moment of X*:

$$E[X^2] = \sum_{S \in P(n)} \left[\frac{(p-1)^{|S|}}{\prod_{i \in \{1 \ldots n\}} |\mathrm{R}_i| - 1} \prod_{i \in S} |\mathrm{R}_i| \prod_{i \in S^C} (|\mathrm{R}_i| - p) \sum_{\{t_i \in \mathrm{R}_i | i \in S\}} \left(\sum_{\{t_j \in \mathrm{R}_j | j \in S^C\}} f(t_i, t_j) \right)^2 \right]$$

Proof: The result follows directly from Proposition 1 by observing that, by the linearity of expectation and properties of X_{t_i}, $E[X^2]$ is the expression on the left of the identity in Proposition 1 as long as $a_i = \frac{|\mathrm{R}_i| - p}{|\mathrm{R}_i| - 1}$ and $b_i = \frac{|\mathrm{R}_i|(p-1)}{|\mathrm{R}_i| - 1}$. □

Using this, the variance of X can be readily computed.

To check this result and to illustrate its use, let us consider the situation when $n = 2$, for which the variance expressions are known from the work on the SMS join [13]. In this case we have, by expanding $\sum_{S \in P(2)}$ in the order {}, {1}, {2}, {1, 2} and denoting the first relation by R and the second by S:

$$\begin{aligned} E[X^2] = \frac{1}{(|\mathrm{R}|-1)(|\mathrm{S}|-1)} \Bigg[& (|\mathrm{R}|-p)(|\mathrm{S}|-p) \left(\sum_{t \in \mathrm{R}} \sum_{v \in \mathrm{S}} f(t \bullet v) \right)^2 \Bigg] \\ & + (p-1)|\mathrm{R}|(|\mathrm{S}|-p) \sum_{t \in \mathrm{R}} \left(\sum_{v \in \mathrm{S}} f(t \bullet v) \right)^2 \\ & + (p-1)|\mathrm{S}|(|\mathrm{R}|-p) \sum_{v \in \mathrm{S}} \left(\sum_{t \in \mathrm{R}} f(t \bullet v) \right)^2 \\ & + (p-1)^2 |\mathrm{R}||\mathrm{S}| \sum_{v \in \mathrm{S}} \sum_{t \in \mathrm{R}} f^2(t \bullet v) \end{aligned}$$

By observing that the formula for the variance of X is the same as the formula for $E[X^2]$ above except that the first term in the square brackets has the coefficient $(|\mathrm{R}|-p)(|\mathrm{S}|-p) - (|\mathrm{R}|-1)(|\mathrm{S}|-1) = (p-1)(p+1-|\mathrm{R}|-|\mathrm{S}|)$, the formula we derived here for $\sigma^2(X)$ and the formula in the SMS join paper are identical.

7.2.3 Extending the Analysis to N_1

Using this analysis, we can address the problem of characterizing the variance of the estimator $N_1 = \alpha\beta$ from Section 5.1. First, if each relation is partitioned randomly into p equi-sized parts, then N_1 can be written as the average of a series of the X estimators considered above, each based on one sample from each relation. The expected value and variance of each of these estimators are the same as for X. Since expectation is linear, this implies that N_1 is an unbiased estimator of the aggregate over the cross product.

When considering the variance of N_1, we observe that if $N_1 = \frac{1}{k}\sum_{i=1}^{k} X_i$, for identical X_i's then:

$$\sigma^2(N_1) = \frac{1}{k^2}\sum_{i=1}^{k} \sigma^2(X_i) + \frac{1}{k^2}\sum_{i \neq j}^{k} Cov(X_i, X_j)$$

where $Cov(X_i, X_j)$ denotes the covariance of the two variables. We already know how to compute $\sigma^2(X_i)$; since each X_i has an identical variance, we simply use the formulas above.

The question is: How to compute the covariance terms? In our prototype of the DBO system, we use the same tactic as the SMS join [13] and simply ignore the covariances. Though space precludes presenting it here, we have produced a result similar to Theorem 2 for the value of $E[X_iX_j]$, which directly leads to a formula for $Cov(X_i, X_j)$. However, just as in the case of the SMS join, this covariance is almost always negative. Thus, simply ignoring the $Cov(X_i, X_j)$ in the formula for $\sigma^2(N_i)$ leads to an overestimated variance. The result is that in practice, we may be somewhat pessimistic in our confidence bounds. However, as argued in the SMS join paper, such pessimism may be warranted. The reason is that it is never practical to compute $\sigma^2(N_i)$ directly, and it must always be estimated (an issue we will consider presently). Ignoring the covariance terms may tend to lend an additional margin of error in this estimation process.

7.2.4 Estimating the Variance

In the previous subsection, we determined a formula for $\sigma^2(X)$ as a function of 2^n aggregates over the cross product $\mathrm{R}_1 \times \mathrm{R}_2 \times \ldots \times \mathrm{R}_n$, each taking the form:

$$y_S = \sum_{\{t_i \in \mathrm{R}_i | i \in S\}} \left(\sum_{\{t_j \in \mathrm{R}_j | j \in S^C\}} f(\{t_i, t_j\}) \right)^2$$

with $S \in P(n)$. By obtaining estimates for each of these terms, an estimate for the variance is readily obtained.

A very simple and also reasonable estimate for each of these terms is based on the samples $\mathrm{R}_1', \mathrm{R}_2' \ldots \mathrm{R}_n'$ and is obtained by computing the aggregate over these samples and scaling up the result by a factor of p for each sum. More formally, $Y_S =$

$$p^{2^n - |S|} \sum_{\{t_i \in \mathrm{R}_i | i \in S\}} \prod_{i \in S} X_{t_i} \left(\sum_{\{t_j \in \mathrm{R}_j | j \in S^C\}} \prod_{i \in S} X_{t_j} f(\{t_i, t_j\}) \right)^2$$

While this estimate is reasonable, it will be biased and will actually overestimate the true value y_S. Fortunately, an unbiased estimate can be constructed from Y_S by observing that:

$$\begin{aligned} E[Y_S] &= p^{2^n - |S|} \sum_{\{t_i \in \mathrm{R}_i | i \in S\}} \prod_{i \in S} E[X_{t_i}] \times E\left[\left(\sum_{\{t_j \in \mathrm{R}_j | j \in S^C\}} \prod_{i \in S} X_{t_j} f(\{t_i, t_j\}) \right)^2 \right] \\ &= \sum_{\{t_i \in \mathrm{R}_i | i \in S\}} p^{|S^C|} E\left[\left(\sum_{\{t_j \in \mathrm{R}_j | j \in S^C\}} \prod_{i \in S} X_{t_j} f(\{t_i, t_j\}) \right)^2 \right] \end{aligned}$$

We now observe that each of the expectation terms within the sum can be determined using Theorem 2 using S^C instead of $\{1 \ldots n\}$ and by ignoring the dependency of f on $\{t_i\}$. With this and using the more concise notation for these terms, we have:

$$E[Y_S] = \sum_{T \in P(S^C)} c_{S,T} \cdot y_{S \cup T}$$

where the coefficient in front of $y_{S \cup T}$ is:

$$c_{S,T} = \frac{(p-1)^{|T|}}{\prod_{i \in P(S^C)} |\mathbb{R}_i| - 1} \prod_{i \in T} |\mathbb{R}_i| \prod_{i \in T^C} |\mathbb{R}_i - p|$$

where the complement of T is taken with respect to $P(S^C)$. Now, if we let $\hat{Y}_{S,T}$ be an unbiased estimate for $y_{S,T}$ for $T \in P(S^C) - \emptyset$, and we let:

$$\hat{Y}_S = \frac{1}{c_{S,\emptyset}} \left(Y_S - \sum_{T \in P(S^C) - \emptyset} c_{S,T} \hat{Y}_{S,T} \right)$$

we have, using linearity of expectation and the above equations, $E[\hat{Y}_S] = y_S$. Thus, $\hat{Y}_S$ is indeed an unbiased estimate for y_S. The equation that defines $\hat{Y}_S$ can be solved recursively by making two observations. First, $Y_{\{1\ldots n\}}$ is an unbiased estimate for $y_{\{1\ldots n\}}$ (which follows directly from Theorem 1 with f replaced by f^2), thus $\hat{Y}_{\{1\ldots n\}} = Y_{\{1\ldots n\}}$. Second, the equation that defines $\hat{Y}_S$ depends only on Y_S and unbiased estimates of terms $y_{S'}$ where S' is a strict superset of S, and thus the recursion always terminates with the unbiased estimator for $y_{\{1\ldots n\}}$ in at most n steps.

7.3 Analysis of N_i for $i > 1$

As described in Section 5.2, the estimates associated with the levelwise steps after the first one are different, in that each record has a $1/p$ probability of appearing in each partition, and the sampling of each record is independent. This changes the analysis.

In subsequent levelwise steps, the samples are produced by flipping an independent p-faced coin for every tuple in the relation and placing it in one of the p samples depending on the outcome. Using the same approach as in the previous section, the random variables X_i have different behavior. In this case:

$$E[X_{t_i}] = 1/p;\ E[X_{t_i} X_{t_i'}] = \begin{cases} 1/p \text{ if } t_i = t_i' \\ 1/p^2 \text{ if } t_i \neq t_i' \end{cases}$$
$$= \delta_{t_i, t_i'} \frac{1}{p} + (1 - \delta_{t_i, t_i'}) \frac{1}{p^2} = \frac{1}{p^2}[1 + (p-1)\delta_{t_i, t_i'}]$$

Since $E[X_{t_i}] = 1/p$ (just as in the first levelwise step), X is unbiased just as in the first levelwise step. However, the second moment of X *is* changed, along with the variance of X:

Theorem 3: *Second moment of X*:

$$E[X^2] = \sum_{S \in P(n)} (p-1)^{|S|} \sum_{\{t_i \in \mathbb{R}_i | i \in S\}} \left(\sum_{\{t_j \in \mathbb{R}_j | j \in S^C\}} f(t_i, t_j) \right)^2$$

Proof: The proof is similar to the proof of Theorem 2, but here $a_i = 1$ and $b_i = p - 1$. □

Otherwise, not much changes in subsequent levelwise steps. The observations of Section 7.2.3 with respect to the covariance between various trials over the variable X hold, and the process of estimating the variance of each X changes only slightly. To determine unbiased estimates for y_S, the coefficients $c_{S,T}$ have to be taken as $c_{S,T} = (p-1)^{|T|}$. Otherwise, the equations that give unbiased estimates for the variance can be solved as before.

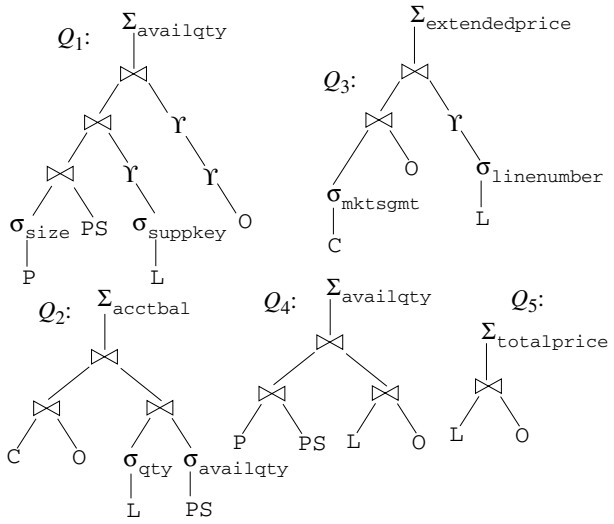

Figure 6: Test query plans.

8 BENCHMARKING

This Section describes a set of benchmarking experiments. Space precludes a detailed benchmark of the DBO engine's performance characteristics; thus, we focus on the goal of answering the following questions:

- How does the width of the confidence bounds produced by the DBO engine decrease in time? Is the decrease rapid and smooth, so that the DBO engine could be used to produce useful results in a short period of time, and more useful results given more time?
- Are the DBO confidence intervals reliable?
- How does the total execution time of the DBO engine compare with the execution time of a traditional database system? Is the overhead incurred by the statistical processing required by the DBO system acceptable?

Experimental Setup. In our experiments, we evaluate five queries over the TPC-H schema. In order to introduce some mild skew into the data to make the evaluation more interesting, we implemented our own TPC-H data generator and generated a database having a scale factor of 10, which creates a database that is approximately 10 GB in size. The queries we run are over the following five tables: (1) `lineitem` (`L`) - 7GB and 60 million rows; (2) `orders` (`O`) - 1.4 GB and 15 million rows; (3) `part` (`P`) - 215 MB and 2 million rows; (4) `partsupp` (`PS`) - 1.4 GB and 8 million rows; and (5) `customer` (`C`) - 240 MB and 1.5 million rows. For more information, see `http://www.tpc.org/`.

To test the width of the confidence bounds produced by the DBO engine and to test total running time, we consider the five queries whose query plans are depicted in Figure 6. The relational selection predicates on `P` and `L` in Q_1 have selectivities of 20% and 60% respectively. Those on `L` and `PS` in Q_2 have selectivities of 99% and 20% respectively. Those on `C` and `L` in Q_3 have selectivities of 99% and 20% respectively. Note that both Q_1 and Q_3 make use of the scan/re-randomize operator.

These query plans were run to completion using the DBO engine. The experimental platform was a 2.4GHz Pentium Xeon machine with 2GB of RAM and dual 10K RPM, 80GB SCSI hard disks. In Figure 7, we plot the relative confidence interval width

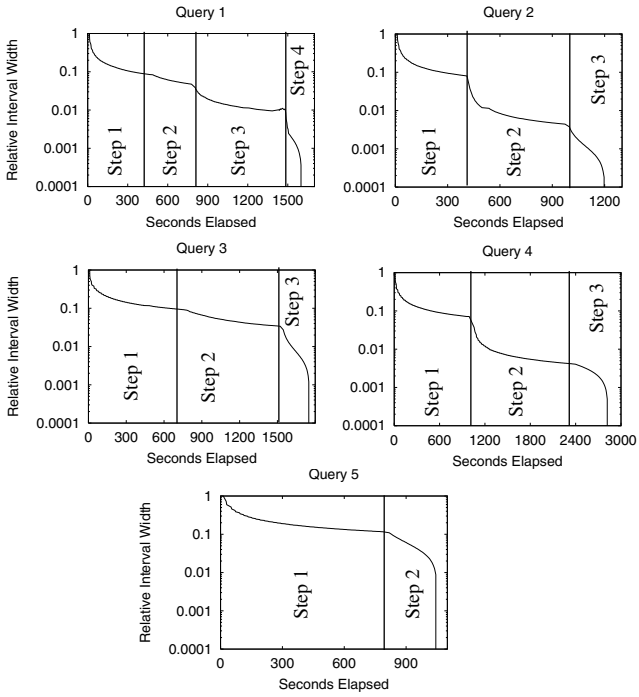

Figure 7: Relative confidence interval width as a function of time for the five test query plans.

produced by DBO as a function of time for these queries (the relative confidence interval width is the ratio between confidence interval width and the current estimate). These CLT-based bounds were produced using a 95% confidence level, meaning that for a calculated variance of σ^2, bounds of approximately $\pm 2\sigma$ around the estimate were used. Thus, a relative interval width of 0.12 means that the width of the 95% confidence bounds is 12% as large as the current estimate.

To test the accuracy of the given confidence intervals, we re-generate the database 100 times and for each instance of the database, we re-run Q_3 and Q_4 to completion. For each query, we consider all of the confidence intervals reported at the end of minute m of the query execution as a group, and for each value of m we compute the fraction of confidence intervals that did, in fact, contain the actual query answer. The results of this experiment are given as Figure 8.

Finally, the time required for completing each query is given as Figure 9. This time is compared with the time required to run the same query to completion on the same machine, using the Postgres system. While we realize that other, widely-used commercial systems such as Oracle are likely to be faster than Postgres, legal restrictions prohibit publishing such a comparison. Still, Postgres is widely used. Thus, this experiment should be seen as testing whether query execution time in DBO is at least "in the ballpark" of what one might expect in terms of completion time from a commercial system.

Discussion. It is possible to draw a few conclusions from these results. First, there does not appear to be much of a hit in terms of additional execution time with the DBO engine as compared to a traditional database system. Our experiments show that DBO is actually significantly faster than Postgres in evaluating each of these particular queries. This does not imply that DBO would be faster than any commercial system, especially since Postgres is

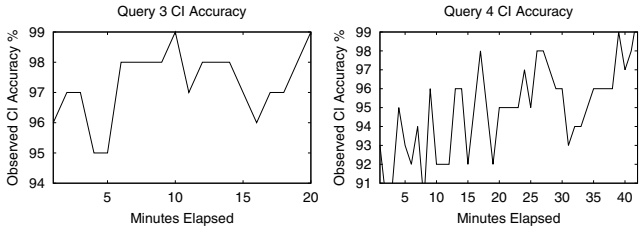

Figure 8: Observed 95% interval accuracy over 100 independent query executions.

surprisingly CPU-bound for this particular workload. However, these results do strongly indicate that algorithms underlying DBO do not incur much of an overhead, validating our claim that the statistical analysis provided by DBO does not come at too high a cost.

Second, these results show that the engine is able to consistently narrow confidence intervals throughout execution. At the beginning of each level, the intervals tend to narrow very quickly (since the estimators associated with each subsequent level are far more accurate than the estimators associated with previous one), but the intervals narrow consistently within each level as well.

Furthermore, these results show that scalability is an absolute necessity in this type of online approximation. In our experiments, DBO fully consumed main memory in 15 to 20 seconds from the start of query processing. Up until this time, the DBO estimate would be identical to the estimate provided by a hashed ripple join, which must be terminated when the main memory is consumed. From Figure 7, it is clear that after such a short time period, the estimates obtained can be far from accurate. For example, in Q_2 the estimate starts out with a 95% confidence interval width that is almost wider than the magnitude of the estimate itself. But by the end of the first levelwise step, DBO is able to shrink that width to less than 10% of the estimate; by the end of the second levelwise step, the width is less than 1% of the estimate. Given the extreme narrowness of the confidence intervals observed after one or two levels in every case, it is reasonable to claim that for many application-specific accuracy requirements, DBO query processing can be terminated early with a satisfactory answer.

Finally, Figure 8 gives strong evidence that the variance calculations described in the paper and the CLT-based bounds we use are in fact valid. Using the binomial distribution, it can easily be calculated that if the true confidence interval probability were 95%, over 100 trials we would expect a 96% chance of observing between 91 and 99 "correct" confidence intervals. From Figure 8 we observe that for the 100 query repetitions tested over Q_3 and Q_4, only three of the 62 minutes have less than 91 correct intervals or more than 99. Significantly, (62 - 3)/62 = 95.2%, which is very close to the 96% that one would expect given 62 sets of 100 tests over true, 95% confidence intervals. Granted, this is not irrefutable evidence of correctness. Only two queries were tested (since each test requires several days) and the 62 minutes reported are not independent (a correct interval in one minute makes it more likely to observe a correct interval in the next). But this certainly is a strong argument that our derivations are in fact valid.

9 RELATED WORK

As discussed previously in the paper, the work most closely related to the DBO engine is the previous work on online aggregation [6][7][10][11] and the SMS join [13]. Online aggregation has its roots in early work linking approximation with processing time [12]. This paper takes inspiration from, and extends, both. For

Query Execution Time

Query	DBO	Postgres
Q_1	26m42s	43m47s
Q_2	20m08s	34m27s
Q_3	29m12s	37m40s
Q_4	47m05s	88m28s
Q_5	17m28s	46m31s

Figure 9: Completion time of DBO vs. Postgres.

example, the statistical results given in Section 7 extend the results of Haas et al. [6][7][8] by extending their analysis to the different types of finite-population sampling without replacement required by the DBO engine, and extend the results of Jermaine et al. [13] by considering Bernoulli (coin-flip) sampling and arbitrary numbers of relations. The algorithms used by DBO clearly have their roots both in the ripple join and in the SMS join, but dramatically extend the applicability of both to the point where the DBO engine may actually be competitive with traditional query-processing methodologies, thereby giving online estimates and accuracy guarantees "for free".

There is a body of relevant work in the database literature on sampling-based algorithms for approximate query processing. Olken's work, summarized in his PhD thesis [16], is well-known. The two papers most closely related to this one describe join synopses [1], and Chaudhuri et al.'s work discusses important issues associated with sampling from joins [2]. However, neither of these papers has the systems-oriented focus of our work, where the goal is to build a system that can run a query from start-up through completion. Join synopses provide a single, fixed precision estimate and are limited to foreign key joins, and it is not clear how to scale Chaudhuri et al.'s work so that *all* of the tuples resulting from a multi-gigabyte join can be sampled in a scalable fashion.

10 FUTURE WORK AND CONCLUSION

This paper has described how the DBO query execution engine can process `SELECT-FROM-WHERE-GROUP BY` aggregate SQL queries over multiple input relations in a scalable fashion, and give statistically rigorous accuracy guarantees from start-up through completion of the plan. This has required significant algorithmic innovation, as well as an extensive statistical analysis of the properties of our new algorithms. The focus of the paper was specifically directed towards query processing (both algorithmic and statistical issues). To keep the paper's scope at a manageable level, other important questions must be deferred to future work. These questions include the following:

- *How should query optimization be performed in the DBO system*? This will be a challenging task, because DBO has two competing optimization goals: running the query to completion quickly, and giving accurate estimates that converge quickly. We plan to use user input to specify the relative importance of the two goals.
- *Are there other join algorithms suitable for use within DBO*? Our preliminary work has focused only on a variant of the sort-merge join. It may be desirable to give DBO the ability to use other joins (such as the hybrid hash join) during the computation of a levelwise step.
- *How must indexing change in the DBO system*? Current sampling-based indexing methodologies [17][18] are likely not useful within the DBO system, because they are targeted towards small samples and require random disk I/Os to sample from a relational selection predicate. Developing indexing and file organizations that support fast sampling from selection predicates is important.
- *How can the randomized data ordering be maintained during data update*? DBO requires a random clustering of data on disk. Developing new, easily-maintained randomized file organizations that support fast updates will be a priority.
- *Can DBO be extended past joins containing equality conditions*? Other operations such as relational subtraction, non-equi-join queries, and duplicate removal are important. There has been some initial work in this area [14], but more effort is needed to allow for truly scalable processing.

References

[1] S. Acharya, P. Gibons, V. Poosala, S. Ramaswamy: Join Synopses for Approximate Query Processing. *SIGMOD* 1999: 275-286.

[2] S. Chaudhuri, R. Motwani, V.R. Narasayya: On Random Sampling over Joins. *SIGMOD* 1999: 263-274

[3] W. Cochran: *Sampling Techniques*. Wiley and Sons, 1977

[4] J.-P. Dittrich, B. Seeger, D.S. Taylor, Peter Widmayer: On producing join results early. *PODS* 2003: 134-142

[5] J.-P. Dittrich, B. Seeger, D.S. Taylor, P. Widmayer: Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm. *VLDB* 2002: 299-310

[6] P.J. Haas, J.M. Hellerstein: Ripple Joins for Online Aggregation. *SIGMOD* 1999: 287-298

[7] P.J. Haas: Large-Sample and Deterministic Confidence Intervals for Online Aggregation. *SSDBM* 1997: 51-63

[8] P.J. Haas, J. F. Naughton, S. Seshadri, A. N. Swami: Selectivity and Cost Estimation for Joins Based on Random Sampling. *J. Com. Syst. Sci.* 52(3): 550-569 (1996)

[9] G. H. Hardy, J. E. Littlewood, and G. Polya. *Inequalities*. Cambridge University Press, 1988.

[10] J.M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, P.J. Haas: Interactive Data Analysis: The Control Project. *IEEE Computer* 32(8): 51-59 (1999)

[11] J.M. Hellerstein, P.J. Haas, H.J. Wang: Online Aggregation. *SIGMOD* 1997: 171-182

[12] G. Özsoyoglu, K. Du, S.G. Swamy, W.-C. Hou: Processing Real-Time, Non-Aggregate Queries with Time-Constraints in CASE-DB. *ICDE* 1992: 410-417

[13] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, A. Pol: A Disk-Based Join with Probabilistic Guarantees. *SIGMOD* 2005: 456-467.

[14] C. Jermaine, A. Dobra, A. Pol, S. Joshi: Online Estimation for Subset-Based SQL Queries. *VLDB* 2005: 745-756.

[15] G. Luo, C. Ellmann, P.J. Haas, J.F. Naughton: A scalable hash ripple join algorithm. *SIGMOD* 2002: 252-262

[16] F. Olken: *Random Sampling from Databases*. PhD Thesis, U. of California, Berkeley, 1993

[17] F. Olken, D. Rotem, P. Xu: Random Sampling from Hash Files. *SIGMOD* 1990: 375-386

[18] F. Olken, D. Rotem: Random Sampling from B+-Trees. *VLDB* 1989: 269-277

[19] L.D. Shapiro: Join Processing in Database Systems with Large Main Memories. *ACM TODS* 11(3): 239-264 (1986)

[20] J. Shao: *Mathematical Statistics*. Springer-Verlag, 1999.