Scalable Approximate Query Processing with the DBO Engine

CHRIS JERMAINE, SUBRAMANIAN ARUMUGAM, ABHIJIT POL,
and ALIN DOBRA
University of Florida

This article 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 answers 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 as soon as they are happy with the query accuracy, and thus encourages exploratory data analysis.

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

General Terms: Algorithms; Performance

Additional Key Words and Phrases: Online aggregation; sampling; randomized algorithms

ACM Reference Format:

Jermaine, C., Arumugam, S., Pol, A., and Dobra, A. 2008. Scalable approximate query processing with the DBO engine. ACM Trans. Datab. Syst. 33, 4, Article 23 (November 2008), 54 pages. DOI = 10.1145/1412331.1412335 http://doi.acm.org/10.1145/1412331.1412335

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.

This article was supported by the National Science Foundation under grants 0347408 and 0612170.

A promising idea is to make randomization the basic database design principle [Hellerstein et al. 1997]. 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 [Hellerstein et al. 1997], and later showed how to evaluate joins so as to give accuracy guarantees during query execution (with the introduction of the ripple join [Haas and Hellerstein 1999]). This work was later extended to a parallel environment [Luo et al. 2002]. However, a problem with this work is that the proposed algorithms are not scalable. As soon as enough data have 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 be unacceptable. In response to this, Jermaine et al. [2005a] 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 greater the number of 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 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 [Haas and Hellerstein 1999; Jermaine et al. 2005a, 2005b] 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.

1.1 New Material in this Article

This article is a substantially expanded version of a previously published conference paper [Jermaine et al. 2007]. The new material includes the following:

—A large part was added to cover use of indexing in DBO. The main contribution in this respect is devising methods to use primary indexing in DBO, a far from trivial problem. The main challenge in incorporating indexes in DBO, which we overcame, is to provide for enough randomization in the use of indexing to allow statistical estimates to be developed. We believe that the resulting methods have significant practical value, since they result in a speedup in terms of the shrinking of the confidence intervals provided.

—The theoretical analysis of sampling estimators is significantly extended and clarified. Due to the lack of space, the analysis in the conference paper is sketchy and does not allow either the verification of the results or, more importantly, the efficient implementation of statistical estimators. All of these shortcomings are remedied in this article through the following contributions:

  (a) A long and careful introduction is provided to the techniques used to analyze sampling estimators. We believe this introduction eases significantly the understanding of the rather complicated theoretical developments.

  (b) The technical result at the core of the general statistical analysis is proved completely. The proof can be used as a pattern for future analysis and is far from straightforward. While the conference paper contained the statement of the technical result, the proof was only hinted, not carried out to any degree.

  (c) The generic analysis is used to identify efficient algorithms to compute the statistical estimators. To this end, we added algorithms that spell out how the computation is to be carried out, since the efficient estimation depends on a number of key ideas that are nontrivial.

—The empirical evaluation was extended to test the indexed DBO as well.

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 [Haas and Hellerstein 1999]. In our discussion of the ripple join (and of all of the algorithms considered in this article), 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 $f(r_1 \bullet r_2 \bullet \ldots)$
FROM R1 as r_1 R1 as $r_2, \ldots,$ Rn 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 tuples in main memory, it becomes necessary to page out one or more tuples to make room for a new tuple, and to page in other tuples in order to check for matches with the new tuple. These I/Os will be random due to the random order of input tuples, so the algorithm causes severe thrashing. Even if each new tuple that is processed requires only a single random disk I/O, the processing rate will be only around 10,000 tuples/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. [2005a] proposed the sort-merge-shrink (SMS) join, a scalable join that is able to maintain online, statistical estimates throughout query execution. The SMS join is closely related to the classic sort-merge join [Shapiro 1986], 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 (R3.c)
FROM R1, R2, R3
WHERE R1.a = R2.a AND R2.b = R3.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 R2 are sorted or hashed on attribute R2.a, then the resulting buckets or runs cannot be joined directly with R3 without re-sorting or re-hashing (because the join with R3 is on the attribute R2.b and the tuples will be sorted on the wrong attribute). If the data from R2 are sorted or hashed on attribute R2.b, then R2 cannot be joined directly with R1. 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 [Dittrich et al. 2003, 2002] to process R1 $\bowtie$ R2, and to pipeline result tuples from the first join into a second SMS join with R3. 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 [Chaudhuri et al. 1999]. 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 this article 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 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 ith 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 11), though this is beyond the scope of this article. Whatever two-phase join algorithm is used, the joins that make up a levelwise step must be carefully

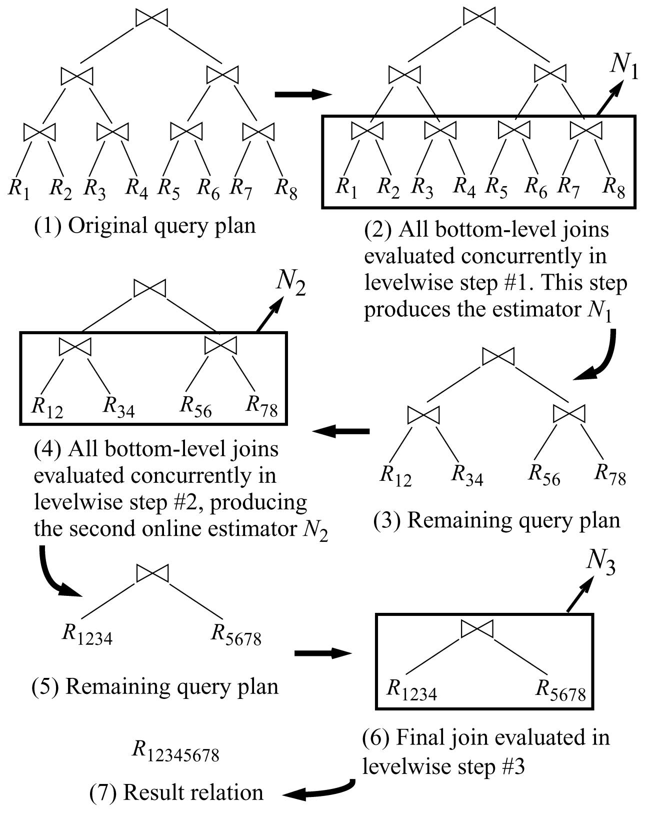

(1) Original query plan
(2) All bottom-level joins evaluated concurrently in levelwise step #1. This step produces the estimator N_1
(3) Remaining query plan
(4) All bottom-level joins evaluated concurrently in levelwise step #2, producing the second online estimator N_2
(5) Remaining query plan
(6) Final join evaluated in levelwise step #3
(7) Result relation
N_1
N_2
N_3
$R_{12345678}$

Fig. 1. Levelwise query evaluation in DBO.

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 ith 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 [Haas and Hellerstein 1999], 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 subsequently, this also implies that 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 tuples from each relation is read into memory from disk (or, since levels are pipelined, the tuples are taken as input directly from the previous levelwise step). Once one run from each input relation is present in memory, all of the tuples 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.

(2) 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 $\texttt{R}_j$ is to be joined on the attribute $\texttt{R}_j\texttt{.key}$, it is not sorted on $\texttt{R}_j\texttt{.key}$ directly. Rather, it is sorted on the value of $H(id, \texttt{R}_j\texttt{.key})$, 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 tuples' 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 tuples from the first run of the first relation are written back to disk, the next set of tuples 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 tuples from the second relation is sorted using H and written out to disk to make room for the second run of tuples 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 $\texttt{R}_1$ is read and processed, then a run from $\texttt{R}_2$ 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 $\texttt{R}_1$ 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

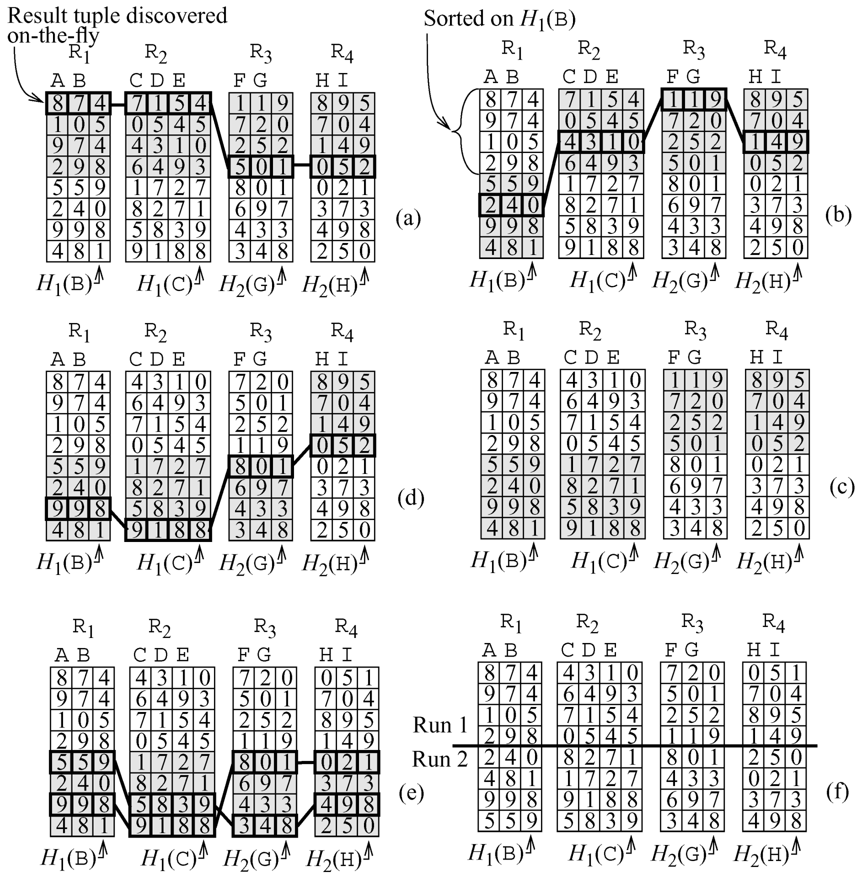
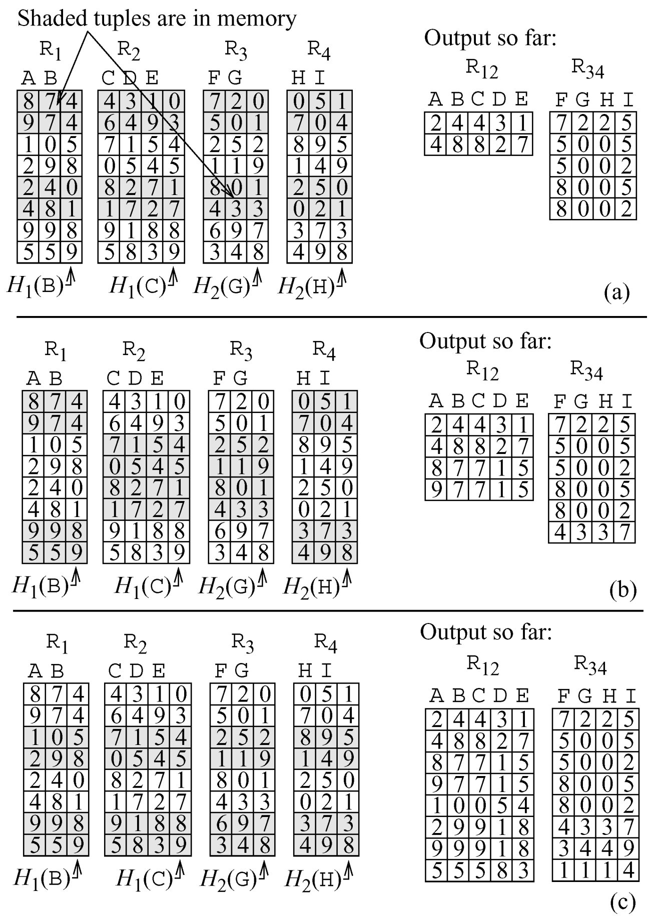


Fig. 2. Scan phase of a levelwise step. In this example, we assume an SQL query having the where clause `WHERE R₁.B = R₂.C AND R₂.E = R₃.F AND R₃.G = R₄.H`, and we assume that the first levelwise step computes the joins $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$. In the scan phase, a run from each input 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).

of all of the joins are run concurrently. That is, the head of each run of each input relation to the ith levelwise step is read into memory, and runs of tuples from each output relation are produced in a round-robin fashion in order to pipeline the result tuples directly into the scan phase of the joins making up the $(i+1)$th levelwise step, without ever writing tuples 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 ith levelwise step. An example merge phase (continuing the example of Figure 2) is depicted in Figure 3.



(a)

R1

A	B	
8	7	4
9	7	4
1	0	5
2	9	8
2	4	0
4	8	1
9	9	8
5	5	9

R2

C	D	E	
4	3	1	0
6	4	9	3
7	1	5	4
0	5	4	5
8	2	7	1
1	7	2	7
9	1	8	8
5	8	3	9

R3

F	G	
7	2	0
5	0	1
2	5	2
1	1	9
8	0	1
4	3	3
6	9	7
3	4	8

R4

H	I	
0	5	1
7	0	4
8	9	5
1	4	9
2	5	0
0	2	1
3	7	3
4	9	8

Output so far:

R12

A	B	C	D	E
2	4	4	3	1
4	8	8	2	7

R34

F	G	H	I
7	2	2	5
5	0	0	5
5	0	0	2
8	0	0	5
8	0	0	2

(b)

R1

A	B	
8	7	4
9	7	4
1	0	5
2	9	8
2	4	0
4	8	1
9	9	8
5	5	9

R2

C	D	E	
4	3	1	0
6	4	9	3
7	1	5	4
0	5	4	5
8	2	7	1
1	7	2	7
9	1	8	8
5	8	3	9

R3

F	G	
7	2	0
5	0	1
2	5	2
1	1	9
8	0	1
4	3	3
6	9	7
3	4	8

R4

H	I	
0	5	1
7	0	4
8	9	5
1	4	9
2	5	0
0	2	1
3	7	3
4	9	8

Output so far:

R12

A	B	C	D	E
2	4	4	3	1
4	8	8	2	7
8	7	7	1	5
9	7	7	1	5

R34

F	G	H	I
7	2	2	5
5	0	0	5
5	0	0	2
8	0	0	5
8	0	0	2
4	3	3	7

(c)

R1

A	B	
8	7	4
9	7	4
1	0	5
2	9	8
2	4	0
4	8	1
9	9	8
5	5	9

R2

C	D	E	
4	3	1	0
6	4	9	3
7	1	5	4
0	5	4	5
8	2	7	1
1	7	2	7
9	1	8	8
5	8	3	9

R3

F	G	
7	2	0
5	0	1
2	5	2
1	1	9
8	0	1
4	3	3
6	9	7
3	4	8

R4

H	I	
0	5	1
7	0	4
8	9	5
1	4	9
2	5	0
0	2	1
3	7	3
4	9	8

Output so far:

R12

A	B	C	D	E
2	4	4	3	1
4	8	8	2	7
8	7	7	1	5
9	7	7	1	5
1	0	0	5	4
2	9	9	1	8
9	9	9	1	8
5	5	5	8	3

R34

F	G	H	I
7	2	2	5
5	0	0	5
5	0	0	2
8	0	0	5
8	0	0	2
4	3	3	7
3	4	4	9
1	1	1	4

Fig. 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` R_1`.B =` R_2`.C AND` R_2`.E =` R_3`.F AND` R_3`.G =` R_4`.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 tuples 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).

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 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 $T(i, j, k) = \mathtt{R}_{j,i} \times \mathtt{R}_{j+1,i} \times \ldots \times \mathtt{R}_{k,i}$ 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 memory 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:

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

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:

$$\beta = \frac{|\,\mathtt{R}_1 \times \ \mathtt{R}_2 \times \ldots \ \mathtt{R}_n|}{\sum_{a=1}^{r} |T(a, 1, n)| + \sum_{a=2}^{p}\sum_{b=1}^{n-1} |T(a, 1, b)||T(a-1, b+1, n)|}.$$

Then, $\alpha\beta$ is an unbiased estimator for the final answer to the query. In the remainder of the article, 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 $\sigma(N_1)$) is considered in Section 5.5 and Section 8.

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 $\mathtt{R}_{34}$. 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 ith 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 [1997] to remove the correlation induced when sampling blocks of tuples rather than tuples. 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 $t_1, t_2, \ldots, t_n$ from intermediate relations 1 through n are actually clumps or sets of tuples, where all 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:

$$f(t_1 \bullet t_2 \bullet \ldots \bullet t_n) = \sum_{t'_1 \in t_1} \sum_{t'_2 \in t_2} \sum_{t'_n \in t_n} f(t'_1 \bullet t'_2 \bullet \ldots \bullet t'_n).$$

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 $H(\mathtt{R.a})$. 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 $H(\mathtt{R.a})$ is from 0 to $(231 - 1)$, 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 tuples 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 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 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 each level and maintained online. The random variable N_1 characterizes the statistical estimator associated with the scan phase of the bottom-most level of the query plan, and N_{d+1} is associated with the tuples output from the merge phase of the top-most 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 associated with 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 8), it follows that for $\{w_1, w_2, \ldots, w_{d+1}\}$, where $\sum_{i=1}^{d+1} w_i = 1$, the following is an unbiased estimate for the final answer to the query:

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

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:

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

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:

$$w_i = \frac{1}{\sigma^2(N_i) \sum_{j=1}^{d+1} \frac{1}{\sigma^2(N_j)}}.$$

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 [Cochran 1977], such as assuming that N is normally distributed (justified by the central limit theorem (CLT) [Shao 1999]), or by using distribution-free bounds such as those provided by Chebyshev's inequality [Hardy et al. 1988]. In our implementation, we use CLT bounds, which, as we show in Section 9, 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 $i = d + 1$, then after r of p partitions have been processed, the variance is computed as:

$$\begin{aligned}\sigma^2(N_{d+1}) &= E\left[N_{d+1}^2\right] - E^2[N_{d+1}] \\ &= E\left[\left(\sum_j \frac{p}{r} X_j f(t_j)\right)^2\right] - E^2\left[\sum_j \frac{p}{r} X_j f(t_j)\right],\end{aligned}$$

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

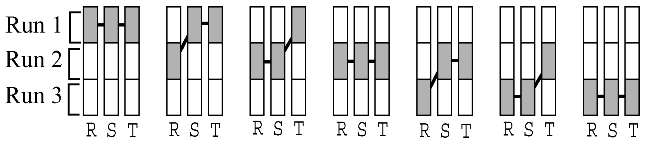


Fig. 4. Using the round-robin method, seven combinations of runs are considered when relations R, S, and T (each broken in to three runs) are sorted during the scan phase of a levelwise step.

pair is independent, so $E[X_iX_j] = r^2/p^2$. Then, simplifying $\sigma^2(N_{d+1})$, we have:

$$\begin{aligned}\sigma^2(N_{d+1}) &= \frac{p^2}{r^2}\left(\sum_j \frac{r}{p} f^2(t_j) + \sum_{j\neq k} \frac{r^2}{p^2} f(t_j)f(t_k)\right) - \frac{p^2}{r^2}\left(\sum_j \frac{r}{p} f(t_j)\right)^2 \\ &= \frac{p^2}{r^2}\left(\sum_j \frac{r}{p} f^2(t_j) - \sum_j \frac{r^2}{p^2} f^2(t_j)\right) \\ &= \frac{p}{r}\left(1 - \frac{r}{p}\right)\sum_j f^2(t_j).\end{aligned}$$

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 p/r 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 $i \neq d+1$ is more complicated, and is left to Section 8.

6. ADDITIONAL CONSIDERATIONS

This section considers some important issues in the design and implementation of the DBO engine. One particularly important issue, indexing, is considered in detail in Section 7.

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 reestimated. 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 of the data space during the scan phase. For example, consider Figure 4, where a levelwise step is computed over three input relations, each broken into three runs. In total, the round-robin approach searches for seven 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 pth 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 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 for the remainder of the levelwise step. Because the set of tuples 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.

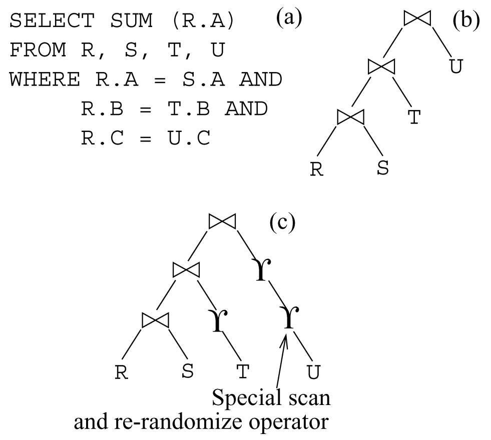
6.3 Handling Data Skew

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 Section 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 joins. 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 8 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).



```
SELECT SUM (R.A)
FROM R, S, T, U
WHERE R.A = S.A AND
     R.B = T.B AND
     R.C = U.C
```

Fig. 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).

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 tuples from each one of the input relations (or at least access temporary relations that contain tuples derived from each of the input relations), the engine cannot provide for an early guess as to the query result.

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. 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 rerandomization 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 rerandomized order. The rerandomization 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 2, 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. INDEXING IN DBO

Indexing is an important consideration in any database system, and DBO is no exception. In order to offer a serious alternative to a classical database system, DBO must employ methods that allow the tuples satisfying very selective relational selection predicates to be located efficiently.

At first glance, it may seem that indexing within a system such as DBO is difficult. DBO depends fundamentally on randomization. However, database indexing structures such as B+-Trees rely on careful organization of data into blocks. Those blocks are in turn carefully organized into larger data structures, so that the index can be traversed efficiently in order to locate all of the tuples that satisfy a given query. Such careful organization is not obviously compatible with the randomization required by DBO. Fortunately, it turns out that despite such considerations, indexing can be used by the system.

Before we begin our in-depth discussion of how indexing is used by DBO, it is worthwhile to mention that indexing is only relevant to the scan phase of the very first (bottom-most) levelwise step, where tuples are read from the randomized relations as query evaluation first begins. This is because at subsequent levels, irrelevant tuples that could be filtered using an index have already been removed, and so an index is of little or no use.

7.1 Secondary Indexing

Using a secondary index within the DBO query processing engine is actually quite straightforward. In secondary indexing, the data contained within the indexed database file are clustered on disk in a way that is totally unrelated to the index. The index itself contains only (*key*, *pointerlist*) pairs, where *pointerlist* is a set of pointers that indicate where in the database file the tuples having search key *key* are located. One example of secondary indexing is the situation where new data are simply appended to the underlying database file in their insertion order, but a B+-Tree is used to index the data using a primary key attribute.

Secondary indexing for "small" queries. The reason that secondary indexing is quite natural within DBO is that the randomized tuple ordering required by DBO can be treated as nothing more than yet another possible tuple clustering, and a secondary index can be constructed over the randomized data. This will have no effect on normal DBO query processing that does not make use of the index.

However, to actually use such an index to help answer a query, some care must be taken. Consider the following query:

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

This query evaluates a potentially complicated, multi-table join over six relations, but the customer relation is only relevant to the extent that it is used to look up a specific customer who is named 'Blah.' In this case, a secondary index over customer.name can be used by DBO to look up the appropriate tuple in customer at the very beginning of query processing. Once this tuple is located, imagine that the corresponding custkey value is found to be 12345. Logically, the customer table can then be removed from the query by instead evaluating an equivalent SQL query that asks for custkey 12345 directly:

```
SELECT SUM (...)
FROM orders o, lineitem l, supplier s, nation n, region r
WHERE o.custkey = '12345' AND l.orderkey = o.orderkey AND ...
```

The resulting query is then evaluated by DBO in the normal fashion. In this way, the index on customer.name is really used no differently than it would be in a classic database engine. Locating the customer named 'Blah' is treated as a preprocessing step that happens before the rest of the query is evaluated—in effect, the selection predicate is pushed down to the bottom of the query plan. The fact that the relevant tuple must be located before normal DBO query evaluation begins, may delay the production of estimates and confidence bounds by the few milliseconds that it takes to perform the index lookup and then chase the pointer into the customer table, but since the first estimates and confidence bounds take a few seconds for DBO to produce anyway, this extra time is insignificant.

Secondary indexing for larger queries. As long as the number of tuples returned by the selection predicates in a query is reasonably small, then the approach of looking up those tuples as a preprocessing step is the preferred option. "Reasonably small" is defined with respect to the time required to retrieve them from disk using the index. The delay incurred in producing the first estimates due to the index lookups should not be noticeable by the user—limiting the time to retrieve all of the required tuples to a second or less. In practice, this allows probably a hundred or so tuples to be retrieved during preprocessing per usable disk, assuming that each random tuple lookup requires a random I/O in the database file and one or less random I/Os in the index file, and that each random I/O requires about 3ms.

If the number of tuples returned by the selection predicates in a query is larger, then the lookups cannot be performed as a preprocessing step. Imagine that the following query is issued:

```
SELECT SUM (...)
FROM orders o, part p, supplier s
WHERE o.date > '1-1-'06' AND o.date < '3-1-06' AND ...
```

In this case, the number of tuples from orders satisfying the range predicate is too large to access all of them using a preprocessing step. To handle this query using an index, DBO will use the following steps:

(1) As it starts up the scan phase of the first levelwise step, DBO first loads a full run from all of the relations that will not be accessed using an index into memory. In our example, this is the relations `part` and `supplier`.

(2) Then, the index that will be used to filter tuples through the selection predicate is used to check how many tuples will be accepted by the selection predicate—let this number be *cnt*. This requires use of a so-called "ranked" indexing structure that augments the (*key*, *pointerlist*) pairs that are stored in the index with an additional value that conts how many tuples in the underlying relation have a key value less than *key*—while this means that we must maintain the ranking information in the index, as we will describe in step (3), a ranked structure is useful anyway in order to provide the required random order of tuples [Olken and Rotem 1989; Antoshenkov 1992].

(3) Once *cnt* is known, we view the subset of tuples from `orders` that satisfy the selection predicate as a new "mini-relation" relation of size *cnt*. This mini-relation is then accessed, one tuple at a time, in randomized order. To access the tuples in randomized order, assuming a B+-Tree index, we generate a random sample i without replacement from 1 to *cnt*, and use the B+-Tree to find the (*key*, *ptr*) pair for the tuple having the ith rank in the mini-relation. This tuple is then retrieved from `orders` and joined with the tuples from `part` and `supplier` that happen to be in memory. The result of the join is then used to update the estimate that is supplied to the user. When computing the estimate as well as its accuracy, the size of `orders` is not used during the computation; since we will only consider tuples from `orders` that satisfy the selection predicate, the size *cnt* of the resulting mini-relation is used instead.

(4) Finally, once all of the tuples satisfying the selection predicate have been retrieved from `orders`, they are pinned in memory for the remainder of the levelwise step, and the scan and merge phases continue in the normal fashion, except for the fact that additional tuples from `orders` are never retrieved—the mini-relation resulting from the selection predicate is always joined in its entirety with tuples from `part` and `supplier`.

The reader may notice that this algorithm is only applicable in the case where all of the tuples from `orders` that satisfy the underlying query's selection predicate can be stored in memory. It would be possible to handle larger mini-relations by breaking them into runs, during the scan phase, that are then merged during the merge phase, just as is done for relations that are not accessed with the help of an index. However, in the case of a secondary index, this makes little sense. If there are so many tuples satisfying a selection predicate that external memory is required to deal with them, then the time to access them via a series of random I/Os using a secondary index will make it impractical to use the index in the first place. For example, accessing 100MB of 100B tuples that are randomly clustered on disk in random order, requires on the order of 10^6 random I/Os—enough to keep an array of 10 disks busy for perhaps 6 minutes. In this time, those same 10 disks could scan a 150GB file from start to finish.

7.2 Primary Indexing in DBO

For relational selection predicates where secondary indexing is not applicable, a standard tactic is to use some sort of primary file organization that clusters tuples together in such a way that those that are frequently accessed at the same time will appear close to one another on the disk. For example, a temporal selection predicate may specify tuples with timestamps in the last year, which may accept too many tuples to make the use of a secondary index a viable option. In this case, a more attractive option would be to sort and then index the tuples based upon the `time` attribute so that once the first tuple in the range has been accessed, the remainder of the tuples in the range can be accessed via a fast sequential scan.

This exact solution is not usable in DBO because tuples cannot be sorted—they must be stored in random order on disk. However, a variation on the idea can be used within the DBO system. Rather than sorting tuples to facilitate the fast evaluation of range predicates, DBO allows the following file organization:

(1) First, tuples are sorted based upon the search key attribute.
(2) Then, the sorted file is partitioned k ways, so that each of the k partitions contains a range of contiguous key values.
(3) Finally, the tuples are randomized *within* each of the k partitions, so that when a given partition is scanned from front to back, the set of tuples that have been scanned always constitutes a random sample of all of the tuples within the partition.

Imagine that the relation $\mathtt{R}_1$ has been partitioned into $\mathtt{R}_{1,1}$, $\mathtt{R}_{1,2}, \ldots,$ $\mathtt{R}_{1,k}$. The net result of this is that the answer to the query:

SELECT SUM $f(r_1 \bullet r_2 \bullet \ldots)$
FROM $\mathtt{R}_1$ as r_1 $\mathtt{R}_2$ as $r_2, \ldots,$ $\mathtt{R}_n$ as r_n

is nothing more than the sum over all i's of the k different queries of the form:

SELECT SUM $f(r_1 \bullet r_2 \bullet \ldots)$
FROM $\mathtt{R}_{1,i}$ as r_1 $\mathtt{R}_2$ as $r_2, \ldots,$ $\mathtt{R}_n$ as r_n.

Since the effect of the partitioning is that we have essentially created k different queries from a single, original query, in DBO the final answer for each of the k "mini" queries can be estimated in tandem during the scan phase of the first levelwise step. The process for doing this is illustrated in detail in Figure 6. Let the k different estimates produced during the first levelwise step be denoted as $N_{1,1}, N_{1,2}, \ldots, N_{1,k}$. Then N_1 (which is the estimate for the final query result that is produced by the first levelwise step) is simply computed as:

$$N_1 = \sum_{i=1}^{k} N_{1,i}.$$

Since the estimate N_1 is nothing more than a sum of individual estimates, the most obvious benefit of this approach is that those partitions that are known not to contain any tuples satisfying the query predicate can be ignored—the resulting estimates will always be zero. This can speed up the scan phase of the

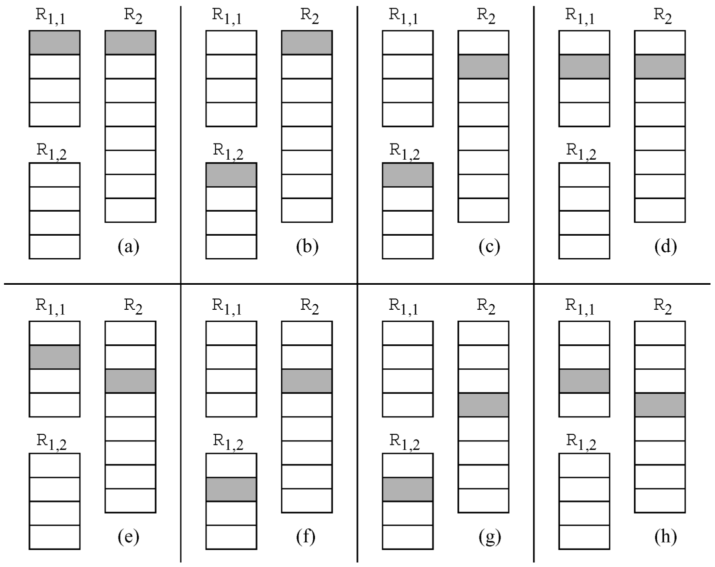


Fig. 6. Producing two estimates after partitioning the relation R_1 in two. The first estimate $N_{1,1}$ is for $R_{1,1} \bowtie R_2$, and the second estimate $N_{1,2}$ is for $R_{1,2} \bowtie R_2$. The process begins in Figure 6(a) by loading the first run of $R_{1,1}$ and the first run of R_2 into memory—the in-memory runs are indicated as the shaded region of the relation. The tuples stored in memory are checked to see if any pairs contribute to the final query result; at this point, the estimate $N_{1,1}$ is first produced. The run from $R_{1,1}$ is sorted and written to disk, and then the first run of $R_{1,2}$ is loaded into memory, as shown in Figure 6(b). The tuples in memory from both R_1 and R_2 are checked to see if any pairs contribute to the final query result; the estimate $N_{1,2}$ is then produced for the first time. Then, the first run of R_2 is sorted and written back to disk, and the second run of R_2 is loaded, joined with the run from $R_{1,2}$, and $N_{1,2}$ is updated. This is depicted in Figure 6(c). This process is repeated until all input relations have been broken into sorted runs and written into disk. The key difference between the partitioned case and the normal DBO scan phase is that when a run from R_1 is processed, the individual partitions of R_1 that are processed rotate or "switch off" in a round-robin fashion. Specifically, the ith time that a run from R_1 is processed, the run is loaded from partition $i \bmod k$, where k is the number of partitions.

levelwise step considerably. This is simply the benefit that one would expect from a useful primary file organization in a classical database system. A less obvious benefit is that, since only the partition or partitions that contain useful data are processed, the confidence bounds on DBO's estimate can be much narrower at a given point in time than if a partitioning has not been used. An even less obvious benefit is that, in the case where several partitions are known to contain relevant data, it becomes possible to shrink the confidence bounds even more quickly by processing the more "important" partitions more aggressively. We consider these issues and others in the following subsections.

Computing the accuracy of N_i. Under the partitioning scheme described in this section, the variance (or inaccuracy) of N_1 is computed as:

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

Each individual $N_{1,i}$ is no different from N_1 in the case of a non-partitioned relation—with the exception that $\texttt{R}_1$ has been replaced with its smaller subset $\texttt{R}_{1,i}$. As a result, the term $\sigma^2(N_{1,i})$ can be computed exactly as $\sigma^2(N_1)$ is computed in Section 8, as long as the input relation sizes and sampling fractions are updated accordingly.

While the process of estimating each $\sigma^2(N_{1,i})$ is really no different than that of a standard DBO variance estimation, the covariance terms do require some consideration. Strictly speaking, each $(N_{1,i}, N_{1,j})$ pair is correlated, since $N_{1,i}$ and $N_{1,j}$ are computed using the same sampling process for the relations other than $\texttt{R}_1$. Consider Figure 6(a) and Figure 6(b). First, the run from $\texttt{R}_{1,1}$ is joined with a run of randomly-selected tuples from $\texttt{R}_2$, and then a run from $\texttt{R}_{1,2}$ is joined with the same run from $\texttt{R}_2$. As a result, some correlation can be induced. However, we have found that in practice this correlation is either quite weak or is quite often negative, which actually results in $\sigma^2(N_1)$ being lower than if there had been no correlation at all. As a result, we simply ignore this correlation. A more nuanced and detailed description of why the covariance can safely be ignored is given in the Appendix.

Keeping the user happy. One difficulty that results from the partitioned approach is that until the first time that each and every $N_{1,i}$ has been computed, it is not possible to give the user any estimates. The reason is quite easy to see when one examines Figure 6. In Figure 6(a), no run has been read from $\texttt{R}_{1,2}$. Since $N_{1,2}$ is undefined, the variance of $N_{1,2}$ is infinite, and thus, so is the variance of N_1 as a whole, so N_1 is not meaningful. This problem can be quite serious in practice, because it can leave the user without an estimate for some time if many partitions have relevant data and must be processed. In this example, there are only two partitions, so $N_{1,2}$ will not remain undefined for long after the levelwise step begins. However, in the general case there may be many partitions. In our experimental section, we partition a relation ten ways, and in this case, the *startup* phase takes around 50 seconds to complete. This is quite a long time for the user to go without any estimates.

Fortunately, there is a reasonably simple solution to this problem. The first run of each partition is broken into a set of k *mini-chunks* of equal size. When the first levelwise step begins over a partitioned relation, rather than reading in the first run of $\texttt{R}_{1,1}$ to begin with, we instead read in the first mini-chunk from each of the k partitions that have data that is relevant to answering our query. These mini-chunks are then combined to form a single run that encompasses all of the partitions. This run is joined in its entirety with the tuples from the other relations that happen to be in memory. Since the run has data from all of the partitions, the union can be treated as a simple random sample from $\texttt{R}_1$ as a whole.[1] As a result, this first estimate is equivalent to a DBO estimate without partitioning, and it can be output to the user immediately. The ith time

[1]Strictly speaking, a simple random sample from $\texttt{R}_1$ as a whole would not always select tuples from each partition precisely in proportion to the size of the partition. To simulate this process exactly, each mini-chunk size should be sampled from a multidimensional hypergeometric random variable. However, the practical difference between the hypergeometric and proportional methods is quite small, and DBO makes use of the latter method for simplicity.

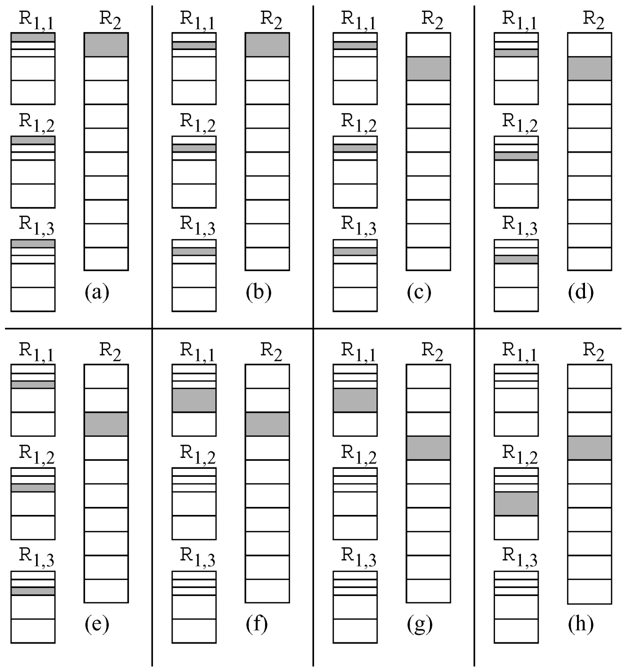


Fig. 7. Using mini-chunks to produce estimates during the beginning of a scan phase over a partitioned relation. The first mini-chunk of the first run of each partition of R_1 is loaded into memory and joined with the first run of R_2. All of the tuples in the mini-chunks are then sorted and written out to disk to be merged in the merge phase of the first levelwise step. The subset of each relation that is initially in memory is depicted as the shaded region of Figure 7(a). All of the mini-chunks are then processed in the same fashion (Figure 7(a)–(e)). Once all of the mini-chunks have been processed, DBO then switches to normal partitioned processing (Figure 7(f)–(h)). The process would then be continued until R_1 and R_2 have been processed in their entirety.

(for $i \leq k$) that a new run is obtained from R_1, it is constructed by combining the ith mini-chunks from each of the k partitions. Then, once all of the k mini-chunks from each partition have been consumed, DBO switches over to the standard partitioned approach described above, and begins pulling runs from single partitions. The process is illustrated in Figure 7.

Optimizing the sampling pattern. More often than not, the k partitions will not be of equal importance for answering a query, even though they may all have relevant data. For example, a given range predicate may entirely contain one partition, though it may only contain part of another partition. Similarly, a query may be issued that lacks a predicate that corresponds exactly to the partitioning scheme, but where the query answer is somehow correlated with the partitioning. For example, a query may ask for the total sales by a given division in a company, and the relation detailing each item sold may be partitioned on the sale date. Due to inflation of prices, or change in the items sold over time, or due to the fact that the division was only created within the last year, it is likely the case that certain partitions are more important for answering this query. In fact, it is not ridiculous to assert that almost every query will be correlated in some way with a partitioning that has been performed using a `time` attribute.

The fact that partitions are likely to not be of uniform importance, can easily be used to great advantage by DBO. As query processing begins, one run from each partition that is relevant to answering the query, is processed using the mini-chunk approach described previously. This allows DBO to give the user an initial set of estimates, and it also allows DBO to collect the statistics that are necessary to estimate each $\sigma^2(N_{1,i})$ (see Section 8). After one run from each partition has been processed, it then becomes necessary to select one of the k partitions to be the source for the next run that will be loaded and joined with all of the other tuples that happen to be in memory. This corresponds to Figure 7(f). While in this example partition $\texttt{R}_{1,1}$ is chosen, there is no reason that this has to be the next partition that is processed. Whenever DBO needs to select another run for processing, it chooses the partition so that the resulting $\sigma^2(N_1) = \sum_{i=1}^{k} \sigma^2(N_{1,i})$ is minimized (subject to the constraint, of course, that once a partition has been processed in its entirety, no runs are selected from it). This simple greedy strategy of always selecting the best partition, has the benefit of being provably optimal. That is, after r runs from $\texttt{R}_1$ have been processed, it is the case that the greedy strategy will produce the lowest possible variance.

To see why the greedy strategy is optimal, we make the observation that $\sigma^2(N_{1,i}) = \frac{\sigma^2(X)}{m}$, where $\sigma^2(X)$ is the variance of a single estimate for partition i, and m is the number of estimators for partition i that have been produced so far (see Section 8.1 for a more precise definition of X). Each $N_{1,i}$ mimics the estimator N_1 in the regular (non-indexed) DBO, where these individual estimates are effectively averaged to produce the final estimate. Since $\sigma^2(X)$ is a constant for estimation purposes (this is the quantity that needs to be determined in the initialization phase of level 1), and $\frac{1}{x}$ is a convex function of x (this can be easily checked), $\sigma^2(N_1)$ is a *separable convex function*. If we consider the optimization problem that consists in allocating r runs to various partitions such that no partition uses more data than available, the resulting problem is a separable convex optimization problem with domain constraints and a single global constraint. This problem is thoroughly treated in Stefanov [2001], where proofs are provided that the greedy strategy of selecting the next partition that minimizes the overall variance provides the optimal solution at every step.

8. STATISTICAL CONSIDERATIONS

In this section, we provide analysis of the accuracy of the various sampling estimators used in this article. At the highest level, this requires us to derive formulas for the variances of the estimators used by DBO, as described in Section 5. To perform the analysis, we develop a general framework that allows uniform treatment for all uniform sampling methods. We carefully present our general analysis method and explain how it can be instantiated to provide analysis for sampling without replacement (which is used in the first levelwise step during DBO query processing) and Bernoulli sampling (which is used in subsequent levelwise steps during DBO query processing).

At the core of the analysis methodology, we develop are a number of technical results that deal with the hardest problem when it comes to analyzing sampling methods over multiple relations: the exponential number of terms that must be manipulated during the analysis. Introducing notation to help express these terms is crucial to analyzing DBO's sampling techniques. The notation helps to reveal the underlying mathematical structure of the results that we prove, and gives unique insight into the inner workings of the sampling methods used by DBO.

While providing mathematical formulas for variances of various estimators is the first step in the analysis of sampling techniques, it is an under-appreciated fact that the actual application of such formulas requires access to the entire database. Since DBO is based upon sampling, the evaluation of such formulas exactly is not practical. If we had access to all of the data in order to evaluate the variance formulas, we could simply evaluate the query exactly. Thus providing unbiased estimators for the variance formulas that operate upon the samples that are available to DBO is crucial, and is the second key problem that we tackle in this section.

Finally, the statistical analysis we present results in rather convoluted estimation formulas. Thus, along with the derivation of variance formulas and associated unbiased estimators, we also present efficient algorithms to compute the unbiased variance estimates from samples. These are the actual algorithms used in the DBO prototype benchmarked in Section 9.

8.1 Individual Estimates and the Levelwise Step

We preface our analysis by pointing out that all of the results described in this section are relevant for computing the accuracy of any one of the estimates obtained during the scan phase of a levelwise step. For example, consider Figure 2. Each time that the cross product of the tuples in memory is searched for any tuple combinations that contribute to the final answer to the query, an individual estimate is produced. This estimate is equivalent to the ripple-join sampling estimate of Haas and Hellerstein [1999]. Figure 2(a) depicts the computation of one such estimate, Figure 2(b) depicts another, and Figure 2(d) depicts yet another. The estimate $\alpha\beta$ from Section 5 essentially averages all of these individual estimates to produce the estimator N_i for the ith levelwise step.

This section focuses on estimating the accuracy of individual estimates, each of which is denoted using the random variable $X_{i,j}$ when we refer to multiple estimates. Given that there are m of these estimates, and given that each relation is partitioned evenly p ways, N_i can be expressed in terms of the properties of the various $X_{i,j}$'s as:

$$\begin{aligned} N_i &= \frac{1}{m}\sum_{j=1}^{m} X_{i,j} \\ \sigma^2(N_i) &= \frac{1}{m^2}\sum_{j=1}^{m}\sigma^2(X_{i,j}) + \frac{1}{m^2}\sum_{j=1}^{m}\sum_{k=1,k\neq j}^{m} Cov(X_{i,j}, X_{i,k}). \end{aligned}$$

In this expression, $Cov()$ denotes the covariance between two of these individual estimates. We ignore this term in the remainder of the section (and in DBO) and focus on the computation of $\sigma^2(X)$, where X is one of the generic $X_{i,j}$. The primary reason that we ignore this covariance is that if every possible $X_{i,j}$ were computed so that effectively no sampling were done (that is, $m = p^n$), then N_i would actually compute the final query result. Since the variance of the final query result is zero, this means that on average, the covariance terms must be negative. Negative covariances actually increase the accuracy of the estimate, so ignoring them should usually not be too much of a problem. For the interested reader, a derivation of the covariance terms is given in Appendix B. Since, outside of the covariance terms, the statistical properties of each $X_{i,j}$ are identical, we focus on how to compute the variance of a single arbitrary X. After estimating $\sigma^2(X)$ for an arbitrary X, once m different estimates have been obtained during a given levelwise step, the variance of N_i is computed by dividing $\sigma^2(X)$ by m.

8.2 Notation and Conventions for Analysis of Estimators

We now introduce notation and conventions we use in the remainder of this section, as well as in the Appendix. Some of the notation is similar to the notation used previously in this article, while the remainder is unique and is required in order to keep the formulas reasonably small.

Let $R_1, \ldots, R_n$ be the n relations that are the arguments of the aggregate query, and let $f(t_1 \bullet \cdots \bullet t_n)$ be the aggregate function that is summed over the cross product of the relations to obtain the answer to the aggregate query. It is assumed that f encodes any selection and join predicates used in the query, so that summing f over the cross product of $R_1, \ldots, R_n$ returns the same result that one would obtain by running the underlying SQL query.

We use tuple variables t_i and $t_{i'}$ to denote tuples from relation R_i, and use the simplifying convention that the argument of the aggregate function $f()$ can be specified as a set of tuples with one tuple from each input relation; the correct ordering and concatenation is then performed automatically by f (the need for this convention will be apparent later).

Since the goal is to analyze estimation of the aggregate over joins using samples of relations, we use the notation R'_i to designate the sample. As in Jrmaine et al. [2005], we will mathematically model the sampling process by introducing 0/1 random variables that indicate whether a given tuple has been selected for inclusion in a sample or not. To this end, we will use the notation X_{t_i} to designate the random variable that takes value 1, when tuple $t_i \in R'_i$ (that is, the tuple is in the sample of R_i), and value 0 otherwise. These random variables allow sums of the form $\sum_{t_i \in R'_i} \mathcal{F}(t_i)$, for any function $\mathcal{F}$, to be rewritten as $\sum_{t_i \in R_i} X_{t_i} \mathcal{F}(t_i)$, which will allow the linearity of expectation to push expectations inside sums.

Since we are dealing with general aggregate queries over cross products that require many levels of nested summations, the notation can become quite complicated. To alleviate this problem we introduce special notation to represent the terms that appear in the analysis. We denote by $\mathcal{P}(n)$ the power set of the set $\{1 : n\}$, that is, the set of all subsets including the empty set and the entire set. For a set $S \in \mathcal{P}(n)$, we denote by $\sum_{\{t_i \in R_i | i \in S\}}$ the multiple sums $\sum_{t_{i_1} \in R_{i_1}} \cdots \sum_{t_{i_k} \in R_{i_k}}$, where $S = \{i_1, \ldots, i_k\}$, that is, the set S consists of indexes $i_1, \ldots, i_k$. This notation is not standard but it will greatly simplify the formulas. With this notation, the final aggregate over the cross product that DBO is trying to estimate is:

$$\sum_{t_1 \in R_1} \cdots \sum_{t_n \in R_n} f(t_1 \bullet \cdots \bullet t_n) = \sum_{\{t_i \in R_i | i \in \{1:n\}\}} f(\{t_i | i \in \{1:n\}\}).$$

8.3 Large Summation Identities

In this subsection, we provide a number of identities involving nested summations that simplify and unify the analysis. The actual analysis then becomes a matter of applying the identities to the two different types of sampling used by DBO—simple without replacement sampling and Bernoulli sampling.

The major difficulty when manipulating the formulas for variances of sampling estimators is the fact that random variables modeling the sampling have many different interactions or correlations among themselves. Let us consider a simple example. Let $X_1, \ldots, X_n$ be independent 0/1 random variables with probability $\frac{1}{2}$ of taking value 0. Let us try to estimate the variance of the random variable $X = \sum_i X_i v_i$, for arbitrary constants v_i, with a direct computation. This variable models Bernoulli sampling from a single set where each item $v_1, \ldots, v_n$ is selected with probability $\frac{1}{2}$. Since by linearity of expectation $E[X] = \sum_i E[X_i v_i] = \sum_i v_i E[X_i] = \frac{1}{2} \sum_i v_i$, it is sufficient to compute $E(X^2)$ in order to obtain the variance of X, since $\sigma^2(X) = E\left[X^2\right] - E[X]^2$. To do so, we have:

$$\begin{aligned} E[X^2] &= \sum_i \sum_j v_i v_j E[X_i X_j] = \sum_i \frac{1}{2} v_i^2 + \sum_i \sum_{j, j \neq i} \frac{1}{2^2} v_i v_j \\ &= \sum_i \frac{1}{2} v_i^2 + \frac{1}{4} \sum_i \sum_j v_i v_j - \frac{1}{4} \sum_i v_i^2 = \sum_i \frac{1}{4} v_i^2 + \frac{1}{4} \left(\sum_i v_i \right)^2, \end{aligned} \qquad (1)$$

where we have used the fact that $X_i^2 = X_i$, since X_i only takes values 0 or 1. The split into two formulas—one involving only one summation over j and a second involving double summation over i and j, excluding the case $i = j$—was necessary, since X_i interacts differently with itself and X_j. The fact that there are multiple cases to consider is not unique to this example. The need for breaking the interactions of the variables into cases will occur time and time again during our analysis, which complicates things considerably.

To alleviate the problem of dealing with formulas involving cases, Dobra [2005] proposed the use of the Kronecker delta symbol: δ_{ij}. δ_{ij} takes value 1 if $i = j$, and value 0 if $i \neq j$. The various cases encountered during variance computation can be encoded using δ_{ij} expressions. In this way, we can avoid restrictions on summation indexes and obtain a purely algebraic treatment of the simplification process. For example, let us analyze simple random sampling without replacement. The key difference between random sampling without replacement, which has fixed size, and Bernoulli sampling, is the nature of the interaction between random variables X_i and X_j. In particular, we have:

$$E[X_i X_j] = \begin{cases} \frac{n/2}{n} = \frac{1}{2} & i = j \\ \frac{n/2}{n}\frac{n/2-1}{n-1} = \frac{n-2}{4(n-1)} & i \neq j. \end{cases}$$

To encode these cases using δ_{ij}, we observe that if we multiply the expression for $i = j$ by δ_{ij}, and the expression for $i \neq j$ by $1 - \delta_{ij}$, and add them up, we obtain a correct expression for $E[X_i X_j]$:

$$E[X_i X_j] = \delta_{ij}\frac{1}{2} + (1-\delta_{ij})\frac{n-2}{4(n-1)} = \frac{n-2}{4(n-1)} + \delta_{ij}\frac{n}{4(n-1)}.$$

To simplify formulas containing δ_{ij}, we use a convenient simplification rule [Dobra 2005; Jermaine et al. 2005a]: for any function $\mathcal{F}(i, j)$ we have:

$$\sum_j \mathcal{F}(i, j)\delta_{ij} = \mathcal{F}(i, i).$$

This is due to that fact that, for all terms inside the sum for which $j \neq i$, $\delta_{ij} = 0$. Thus they do not contribute to the result. The only term that has any contribution is $\mathcal{F}(i, i)$.

With this, we have:

$$\begin{aligned} E[X^2] &= \sum_i \sum_j v_i v_j E[X_i X_j] = \sum_i \sum_j v_i v_j \left(\frac{n-2}{4(n-1)} + \delta_{ij}\frac{n}{4(n-1)}\right) \\ &= \sum_i \sum_j \frac{n-2}{4(n-1)} v_i v_j + \sum_i \frac{n}{4(n-1)} v_i^2 \qquad (2) \\ &= \frac{n-2}{4(n-1)}\left(\sum_i v_i\right)^2 + \frac{n}{4(n-1)}\sum_i v_i^2. \end{aligned}$$

This result immediately gives the formula for variance of X in the case of simple sampling without replacement from a single relation. Notice that the proof is simpler if δ_{ij} is used. An even more important benefit of using δ_{ij} is that the proof can be made generic, and so the previous derivation needs to be performed only once for all sampling methods for which $E[X_i X_j]$ does not depend on the actual value of i or j, and depends only on whether they are equal or not. In particular, if we let $E[X_i X_j] = a + b\delta_{ij}$, for some constants a and b, we immediately have:

$$\begin{aligned} E[X^2] &= \sum_i \sum_j v_i v_j E[X_i X_j] = \sum_i \sum_j v_i v_j (a + b\delta_{ij}) \\ &= a\left(\sum_i v_i\right)^2 + b\sum_i v_i^2. \qquad (3) \end{aligned}$$

By appropriately selecting the constants a and b, we obtain the formulas in Equation 1, for Bernoulli sampling, and Equation 2, for sampling without replacement for the case where $E[X_i] = 1/2$. Just as easily, we can obtain the formulas for the general case where $E[X_i] = p$ and, for sampling without replacement, the size of the sample is n'. Since the derivation of Equation 3 makes no use of the fact that X_i is a 0/1 random variable, it can be applied to the analysis of sampling with replacement as well.

From these examples, two key observations will guide the rest of this section:

(1) First, the details of particular sampling methods can be abstracted and expressed as constants in a generic fashion. Thus derivation of formulas can be performed without any reference to the particular sampling method. In other words, it seems that a technical result like the one in Equation 3 suffices to analyze a very large class of sampling methods.
(2) Second, given the values of v_i, a and b, the last formula for $E[X^2]$ in Equation 3 can be evaluated as efficiently as the expected value $E[X]$ of X. This happens because $\sum_i v_i$ and $\sum_i v_i^2$ can be computed in a single pass with $O(n)$ complexity, and the final value is obtained with extra $O(1)$ effort. The surprising fact is that the efficiency of the computation does not depend on the sampling technique. This observation suggests that computing or estimating variance of sampling estimators might not be expensive.

These two key observations can be combined and formalized in a single lemma, which will serve as a basis for the variance analysis in the remainder of this section:

LEMMA 1. *Let $a_i, b_i, i \in \{1:n\}$ be arbitrary values. Then*

$$\sum_{\{t_i \in R_i | i \in \{1:n\}\}} \sum_{\{t'_i \in R_i | i \in \{1:n\}\}} \left(\prod_{i \in \{1:n\}} a_i + b_i \delta_{t_i t'_i} \right) f(\{t_i | i \in \{1:n\}\}) f(\{t'_i | i \in \{1:n\}\})$$
$$= \sum_{S \in \mathcal{P}(n)} \prod_{i \in S^C} a_i \prod_{j \in S} b_j \sum_{\{t_i \in R_i | i \in S\}} \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i | i \in S\} \cup \{t_j | j \in S^C\}) \right)^2,$$

where S^C is the complement of set S with respect to $\{1:n\}$.

PROOF. The proof is provided in Appendix A. □

We will apply this result to the analysis of the three types of sampling, in subsequent subsections. We end this subsection by making a few closing comments on Lemma 1.

First, when $n = 1$ (that is, we have a single relation) and $f(t_i) = v_i$, the result in the lemma gives the last simplification step in Equation 3. In the much more complicated analysis over multiple relations that is required to compute the variance of DBO's estimates, the lemma will similarly allow us to perform the key simplification step that is required to derive the formula for the second moment ($E[X^2]$) of the estimate.

Second, computing the expression resulting from the simplification suggested by Lemma 1 is quite manageable, and is comparable with the complexity of evaluating a single aggregate over the input relations (which would involve summing up the value of a function over the cross product of relations). Using this lemma will result in variance estimators that are exponential in the number of relations, but linear in the relation size, and quite practical for fewer than a dozen relations. Furthermore, due to the resulting strong resemblance to a database cross product computation, we will be able to derive efficient computation algorithms, in Section 8.5.1, that rely on standard database methods, such as grouping, to render the required computations even more efficient.

8.4 Generic Analysis of Sampling

The generic problem of designing an estimator X for the final answer to the query, and computing $\sigma^2(X)$, can be solved without a direct reference to the type of sampling, since the type of sampling influences only the constants involved, not the actual algebraic manipulations that need to be performed. We will show in subsequent sections how the constants required to apply this analysis can be derived for the two types of sampling used by DBO.

To provide a generic study of sampling methods, we assume the following formulas for the moments of the random variables X_{t_i} (these random variables indicate, for each relation, whether the tuple t_i is included in the sample or not):

$$E\left[X_{t_i}\right] = e_i, \qquad E\left[X_{t_i} X_{t_i'}\right] = a_i + b_i\ \delta_{t_i t_i'}.$$

The generic sampling-based estimate for the aggregate function is:

$$\begin{aligned} X &= \frac{1}{\prod_{i=1}^{n} e_i} \sum_{\{t_i \in R_i' | i \in \{1:n\}\}} f(\{t_i | i \in \{1:n\}\}) \\ &= \frac{1}{\prod_{i=1}^{n} e_i} \sum_{\{t_i \in R_i | i \in \{1:n\}\}} \prod_{i=1}^{n} X_{t_i} f(\{t_i | i \in \{1:n\}\}). \end{aligned}$$

The following result shows that X is an unbiased estimate of the aggregate function:

THEOREM 1. *Using the setup and notation above, we have:*

$$E[X] = \sum_{\{t_i \in R_i | i \in \{1:n\}\}} f(\{t_i | i \in \{1:n\}\},$$

that is, X is an unbiased estimator of the aggregate.

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 the product of expectations). We have:

$$\begin{aligned} E[X] &= \frac{1}{\prod_{i=1}^{n} e_i} \sum_{\{t_i \in R_i | i \in \{1:n\}\}} \prod_{i=1}^{n} E[X_{t_i}] f(\{t_i | i \in \{1:n\}\} \\ &= \sum_{\{t_i \in R_i | i \in \{1:n\}\}} f(\{t_i | i \in \{1:n\}\}), \end{aligned}$$

since $\prod_{i=1}^{n} E[X_{t_i}] = \prod_{i=1}^{n} e_i$, which cancels the $\frac{1}{\prod_{i=1}^{n} e_i}$ terms, thus giving the desired result. □

We now address the more complicated problem of computing the variance of X, $\sigma^2(X)$. Since $\sigma^2(X) = E[X^2] - E[X]^2$, we need only address the problem of computing $E[X^2]$, since $E[X]$ is given by the above result. To compute $E[X^2]$, we need the technical result provided by Lemma 1:

Proposition 1. *Using the setup and notation above,*

$$E[X^2] = \sum_{S \in \mathcal{P}(n)} \prod_{i \in S^C} \frac{a_i}{e_i^2} \prod_{i \in S} \frac{b_i}{e_i^2} \sum_{\{t_i \in R_i | i \in S\}} \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t_j\}) \right)^2$$

Proof. Using the definition of X, linearity of expectation, and Lemma 1 (in this order), we have:

$$\begin{aligned} E[X^2] &= \frac{1}{\prod_{i=1}^n e_i^2} \sum_{\{t_i \in R_i | i \in \{1:n\}\}} \sum_{\{t_i' \in R_i | i \in \{1:n\}\}} \prod_{i \in \{1:n\}} E\left[X_{t_i} X_{t_i'}\right] f(\{t_i\}) f(\{t_i'\}) \\ &= \sum_{\{t_i \in R_i | i \in \{1:n\}\}} \sum_{\{t_i' \in R_i | i \in \{1:n\}\}} \prod_{i \in \{1:n\}} \left(\frac{a_i}{e_i^2} + \frac{b_i}{e_i^2} \delta_{t_i, t_i'} \right) f(\{t_i\}) f(\{t_i'\}) \\ &= \sum_{S \in \mathcal{P}(n)} \prod_{i \in S^C} \frac{a_i}{e_i^2} \prod_{j \in S} \frac{b_j}{e_j^2} \sum_{\{t_i \in R_i | i \in S\}} \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t_j\}) \right)^2 . \quad \square \end{aligned}$$

Using this, we can obtain the formula for $\sigma^2(X)$:

Theorem 2. *Using the following notation:*

$$y_S = \sum_{\{t_i \in R_i | i \in S\}} \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t_j\}) \right)^2, \quad S \neq \emptyset$$

$$y_\emptyset = \left(\sum_{\{t_i \in R_i | i \in \{1:n\}\}} f(\{t_i | i \in \{1:n\}\}) \right)^2 = E[X]^2,$$

we have:

$$\sigma^2(X) = \sum_{S \in \mathcal{P}(n)} \left(\prod_{i \in S^C} \frac{a_i}{e_i^2} \prod_{i \in S} \frac{b_i}{e_i^2} \right) y_S - y_\emptyset.$$

Proof. The statement follows directly from the fact that $\sigma^2(X) = E[X^2] - E[X^2]$, and from the result in Proposition 1. $\square$

Thus the variance of any sampling estimator is a linear combination of terms of the form y_S. Only the coefficients depend on the particular sampling method.

8.5 Computing Estimates for σ^2 (X)

In the previous subsection, we determined a formula for $\sigma^2(X)$; this is a linear combination of 2^n aggregates of the form y_S with $S \in \mathcal{P}(n)$. By obtaining estimates for each y_S from the samples, an estimate for the variance is readily obtained. A reasonable estimate based on samples $R_1', \ldots, R_n'$ is obtained by compensating for the difference in size of the samples and the original relations. In particular, for simple summations of the form $\sum_{t_i}$, we introduce one factor $\frac{1}{e_i}$, since only an e_i fraction of tuples is included on average. Similarly, for summations that are under the square, we introduce the factor $\frac{1}{e_i^2}$, since they correspond to two sums. This correction will compensate for the most pronounced discrepancy, but it is not expected to produce an unbiased estimator.

With this, the estimate for y_S is:

$$\begin{aligned} Y_S &= \frac{1}{\prod_{i\in S} e_i} \frac{1}{\prod_{j\in S^C} e_j^2} \sum_{\{t_i'\in R_i | i\in S\}} \left(\sum_{\{t_j'\in R_j | j\in S^C\}} f(\{t_i | i \in S\} \cup \{t_j | j \in S^C\}) \right)^2 \\ &= \frac{1}{\prod_{i\in S} e_i} \frac{1}{\prod_{j\in S^C} e_j^2} \sum_{\{t_i\in R_i | i\in S\}} \prod_{i\in S} X_{t_i} \left(\sum_{\{t_j\in R_j | j\in S^C\}} \prod_{j\in S^C} X_{t_j}^{(j)} f(\{t_i\} \cup \{t_j\}) \right)^2 . \end{aligned}$$

The expression for Y_S in terms of samples R_i' is useful in order to provide algorithms for computation of Y_S, while the expression in terms of random variables X_{t_i} is useful for analysis. It is important to notice that Y_S is a biased estimate of y_S. We show how the estimate can be unbiased in Section 8.5.2.

8.5.1 *Efficient Computation of* Y_S. There are two major concerns when it comes to estimation of Y_S: how to efficiently compute each individual term, and how to efficiently extend the computation to all terms by sharing computation among subtasks.

If Y_S were evaluated directly using the formula, the computation would be very inefficient for two reasons:

(1) The formula requires a summation over the join result of all of samples.
(2) Since some parts of the expressions are squared, without some care, much space would be required to maintain each squared term individually in order to subsequently perform the querying and final summation.

Thus, in this section, we will discuss some optimizations used to perform the required computations efficiently.

Let us first address the problem of evaluating the terms that must be squared during computation of Y_S. Specifically, for given $\{t_i\}$, the goal is to efficiently compute:

$$\mathcal{F}_S(\{t_i\}) = \sum_{\{t_j'\in R_j | j\in S^C\}} f(\{t_i, t_j\}).$$

In order to find an efficient way to evaluate this expression, we notice that $f(\cdot)$ encodes not just the aggregate function, but any join predicate that is present in the underlying query as well. Any combination of tuples from the underlying relations that are not accepted by the predicate and are not included in the query result set, will generate a zero value for $f(\cdot)$ and can be ignored. Denote by $\mathcal{D}$ the query result set—that is, the set of tuples discovered in the sample that are accepted by the underlying relational selection/join predicates.

In order to compute the function $\mathcal{F}_S(\{t_i\})$ for given $\{t_i\}$, we need to select from $\mathcal{D}$ only the tuples in the query result set that are accepted by $f(\cdot)$ and are produced by joining with tuples $\{t_i\}$. In this way, each set $\{t_i\}$ defines a group, and to evaluate $\mathcal{F}_S(\{t_i\})$ for each possible $\{t_i\}$, we need to evaluate a `GROUP BY` query over the query result set, with the groups induced based upon the identity of the tuple from relation i that was used to form the tuples in the query result set. We can group all result tuples in $\mathcal{D}$ based on values $\{t_i\}$, and for each group compute $\mathcal{F}_S(\{t_i\})$ in a single pass over the group and then add the squares of these values over the groups to obtain the estimate Y_S.

To perform the grouping efficiently, an identifier that allows us to determine which tuple from each relation was used to produce each value for $f(\cdot)$ is maintained along with the value for $f(\cdot)$. In practice, Y_S is computed only over those tuples that have been sampled by DBO, and so all of the $f(\cdot)$, tuple id combinations can easily be stored in main memory. The required grouping can then be implemented as an in-memory sort on tuple identifiers. After sorting, each group is processed one-at-a-time, and we need only $O(1)$ space to maintain $\mathcal{F}_S(\{t_i\})$ for the current group, and Y_S over all of the groups in the query result set. Thus the extra cost to compute Y_S over a given query result set is only an in-memory sort followed by a single scan. Since there are 2^n different values for Y_S, this implies that the overall cost of computing the estimate of $\sigma^2(X)$ is 2^n times the cost for sorting the query result set, which is quite manageable for reasonable values of n.

Algorithm 1 formalizes the above ideas for computing Y_S. Y_S can be computed for any S by calling procedure EstimateYS(S, $\mathcal{D}$) for each set $S \in \mathcal{P}(n)$.

8.5.2 *Unbiasing Y_S.* Using our prototype implementation, we noticed that unbiased estimates for the variance are usually more statistically stable than biased estimates and, more importantly, they are negative significantly less

Algorithm 1. EstimateYS(S,$\mathcal{D}$)

```
Require: Set $S$ that defines $Y_S$ and the set of matches $\mathcal{D}$.
Ensure: Value of $Y_S$ computed over $\mathcal{D}$
 1: Sort $\mathcal{D}$ based on $\{t_i | i \in S\}$
 2: $Y_S = 0.0$ (overall aggregate)
 3: $F = 0.0$ (partial aggregate)
 4: $t' = \emptyset$ (keep track of the previous tuple)
 5: for all $t \in \mathcal{D}$ in sorted order do
 6:     if $t =_S t'$ (are we still within the same group) then
 7:         $F = F + f(t)$
 8:     else
 9:         $Y_S = Y_S + F^2$; $F = 0.0$; $t' = t$
10:     end if
11: end for
12: $Y_S = \frac{Y_S + F^2}{\prod_{i \in S} e_i \prod_{j \in S^C} e_j^2}$
13: return $Y_S$
```

often. While variance is always positive, the estimate can be negative, with disastrous consequences for estimation, since no analysis can be performed due to the fact that square roots have to be extracted from variances to obtain confidence intervals. Unbiasing a variance estimate usually results in an increase in variance. Quite often, the actual error of the estimate is slightly increased as well. Providing a theoretical characterization of how much the error increases is technically challenging in this case, since it would require a fourth order analysis.

It is important to notice that, due to the fact that covariances are ignored in the analysis of N_i—the levelwise estimator obtained by averaging all individual estimators X_{ij}—unbiased estimates of $\sigma^2\left(X_{ij}\right)$ do not result in a perfectly unbiased estimate of $\sigma^2\left(N_i\right)$.

Computing Y_S over a sample, rather than the actual query result set, results in bias, which means that, on expectation, Y_S is not equal to y_S. In order to unbias Y_S, we first have to compute the bias, that is the difference between the expected value of Y_S and y_s. Then, by designing an unbiased estimator for the bias we can correct Y_S to remove the bias. To this end, we first compute:

$$\begin{aligned} E\left[Y_S\right] &= \frac{1}{\prod_{i\in S} e_i}\frac{1}{\prod_{j\in S^C} e_j^2}\sum_{\{t_i\in R_i \mid i\in S\}}\prod_{i\in S} E\left[X_{t_i}\right] \\ &\quad \times E\left[\left(\sum_{\{t_j\in R_j \mid j\in S^C\}}\prod_{j\in S^C} X_{t_j}^{(j)} f(\{t_i,t_j\})\right)^2\right] \\ &= \frac{1}{\prod_{j\in S^C} e_j^2}\sum_{\{t_i\in R_i \mid i\in S\}} E\left[\left(\sum_{\{t_j\in R_j \mid j\in S^C\}}\prod_{j\in S^C} X_{t_j}^{(j)} f(\{t_i,t_j\})\right)^2\right]. \end{aligned}$$

Now, using the same notation $\mathcal{F}_S(\{t_i\})$ for the squared term as before, we observe that in a manner similar to the computation in Proposition 1, by a direct application of the Lemma 1 we obtain:

$$E\left[\mathcal{F}_S(\{t_i\})\right] = \sum_{T\in\mathcal{P}(S^C)}\prod_{j\in T} a_j \prod_{k\in T^C} b_k \sum_{\{t_j \mid j\in T\}}\left(\sum_{\{t_k \mid k\in T^C\}} f(\{t_i,t_j,t_k\})\right)^2.$$

The complement T^C of T is taken with respect to S^C. By replacing this formula in the expression for $E\left[Y_S\right]$ and observing that the summation over $\{t_i \in R_i \mid i \in S\}$ can be pushed inside the summation over $T \in \mathcal{P}(S^C)$ and grouped with the summation over $\{t_j \mid j \in T\}$, it can be seen that $E\left[Y_S\right]$ takes the form:

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

where the constant $c_{S,T}$ is:

$$c_{S,T} = \prod_{k\in S^C - T}\frac{a_k}{e_k^2}\prod_{j\in T}\frac{b_j}{e_j^2}. \qquad (4)$$

An important observation is the fact that:

$$c_{\emptyset,S} = \prod_{j \in S^C} \frac{a_j}{e_j^2} \prod_{i \in S} \frac{b_i}{e_i^2},$$

thus the formula for $\sigma^2(X)$ in Theorem 2 can be rewritten as:

$$\sigma^2(X) = \sum_{S \in \mathcal{P}(n)} c_{\emptyset,S} \cdot y_S - y_\emptyset. \tag{5}$$

Now, if we let $\hat{Y}_S$ be unbiased estimates for y_S for $T \in \mathcal{P}(S^C) - \emptyset$, and we let

$$\hat{Y}_S = \frac{1}{c_{S,\emptyset}} \left(Y_S - \sum_{T \in \mathcal{P}(S^C) - \emptyset} c_{S,T} \hat{Y}_{S \cup T} \right), \tag{6}$$

we have, using linearity of expectation and the above equations:

$$\begin{aligned} E[\hat{Y}_S] &= \frac{1}{c_{S,\emptyset}} \left(E\,[Y_S] - \sum_{T \in \mathcal{P}(S^C) - \emptyset} c_{S,T} E[\hat{Y}_{S \cup T}] \right) \\ &= \frac{1}{c_{S,\emptyset}} \left(\sum_{T \in \mathcal{P}(S^C)} c_{S,T} \cdot y_{S \cup T} - \sum_{T \in \mathcal{P}(S^C) - \emptyset} c_{S,T} \cdot y_{S \cup T} \right) \\ &= y_S. \end{aligned}$$

Thus indeed $\hat{Y}_S$ is an unbiased estimator for y_S. Since Equation 6 relates $\hat{Y}_S$ and $\hat{Y}_{S'}$ with $S' = S \cup T$ (a strict super-set of S), it can be used to compute $\hat{Y}_S$ recursively without danger of infinite recursion. Also, by using memoization (i.e. caching values of $\hat{Y}_S$ once they are computed), the exponential algorithm can be avoided when $\hat{Y}_S$ is computed for all values of S. By replacing y_S by $\hat{Y}_S$ in Equation 5, an unbiased estimate for the variance of X is obtained. Algorithm 2 uses these ideas to compute an unbiased estimate of Y_S, and Algorithm 3 to compute an unbiased estimate of $\sigma^2(X)$ given all estimates of the form Y_S.

Algorithm 2. UnbiasedEstimateYS(S, $\mathcal{Y}$)

```
Require: Set $S$ and biased estimates $\mathcal{Y} = \{Y_T, \forall T \in \mathcal{P}(n)\}$
Ensure: Unbiased estimate $\hat{Y}_S$ of $Y_S$
 1: if Memoized $\hat{Y}_S$ then
 2:     return $\hat{Y}_S$
 3: end if
 4: $\hat{Y}_S = Y_S$
 5: for all $T \in \mathcal{P}(S^C) - \emptyset$ do
 6:     Compute $c_{S,T}$ using Equation 4
 7:     $\hat{Y}_S = \hat{Y}_S - c_{S,T} * $UnbiasedEstimateYS($S \cup T, \mathcal{Y}$)
 8: end for
 9: Compute $c_{S,\emptyset}$
10: $\hat{Y}_S = \frac{\hat{Y}_S}{c_{S,\emptyset}}$
11: Memoize $\hat{Y}_S$
12: return $\hat{Y}_S$
```

Algorithm 3. UnbiasedVarianceEstimate($\mathcal{Y}$)

```
Require: Biased estimates $\mathcal{Y} = \{Y_T, \forall T \in \mathcal{P}(n)\}$
Ensure: Unbiased estimate of $\sigma^2(X)$
1: $\sigma^2(X) = -$UnbiasedEstimateYS($\emptyset, \mathcal{Y}$)
2: for all $S \in \mathcal{P}(n)$ do
3:     Compute $c_{\emptyset,S}$ using Equation 4
4:     $\sigma^2(X) = \sigma^2(X) + c_{\emptyset,S} *$ UnbiasedEstimateYS($S, \mathcal{Y}$)
5: end for
6: return $\sigma^2(X)$
```

8.6 Analysis of Sampling Without Replacement

What remains to be done is to produce specific instances of the general methods described thus far to handle the two types of sampling used by DBO. The analysis for simple random sampling without replacement is provided in this subsection. This is the sampling used by DBO in the first levelwise step. The analysis for Bernoulli sampling is given in the next subsection, which applies to all levelwise steps past the first one.

In order to simplify the analysis, we assume that each relation is randomly partitioned into p parts of equal size. Incorporating variable-size partitioning into the analysis is straightforward, but complicates the analysis unnecessarily, and differing values for p are not used in the DBO prototype.

We begin by explaining how an unbiased estimator for the aggregate over $R_1 \times \cdots \times R_n$ can be constructed using samples $R'_1, \ldots, R'_n$ and how its variance can be estimated. Since the sampling is without replacement (fixed size), we have $|R'_i| = \frac{|R_i|}{p}$.

In order to analyze the random variable X, we first need to specify two properties of the random variables X_{t_i}. First, since each relation is sampled independently, the random variables for different values of i are independent, which means that the expectation of the product is the product of expectations. For each such random variable, and for any tuples t_i, t'_i, we have:

$$\begin{aligned} E[X_{t_i}] &= \frac{1}{p} \\ E[X_{t_i} X_{t'_i}] &= \begin{cases} \frac{1}{p} & \text{If } t_i = t'_i \\ \frac{1}{p^2}\frac{|R_i|-p}{|R_i|-1} & \text{If } t_i \neq t'_i \end{cases} = \delta_{t_i t'_i}\frac{1}{p} + \left(1-\delta_{t_i t'_i}\right)\frac{1}{p^2}\frac{|R_i|-p}{|R_i|-1} \\ &= \frac{1}{p^2(|R_i|-1)}\left[(|R_i|-p) + |R_i|(p-1)\delta_{t_i t'_i}\right], \end{aligned}$$

where we used the fact that the expectation of a 0, 1 random variable is equal to the probability that it takes the value 1.

By setting:

$$e_i = \frac{1}{p}, \qquad a_i = \frac{|R_i|-p}{p^2(|R_i|-1)}, \qquad b_i = \frac{|R_i|(p-1)}{p^2(|R_i|-1)}$$

in the general analysis in Section 8.4, we readily obtain the following:

PROPOSITION 2. *For each $i \in \{1:n\}$, let R'_i be a sample without replacement of size $|R'_i| = \frac{|R_i|}{p}$. Then, the estimate of the aggregate function:*

$$X = p^n \sum_{\{t_i \in R'_i | i \in \{1:n\}\}} f(\{t_i | i \in \{1:n\}\}$$

is an unbiased estimate of the aggregate function and has variance:

$$\sigma^2(X) = \sum_{S \in \mathcal{P}(n)} \left[\frac{(p-1)^{|S|}}{\prod_{i \in \{1:n\}} |R_i| - 1} \prod_{i \in S^C} (|R_i| - p) \prod_{i \in S} |R_i| \right] y_S - y_\emptyset.$$

Moreover, an unbiased estimate of $\sigma^2(X)$ can be efficiently computed using the method in Section 8.4 by setting the values of e_i, a_i, b_i as previously shown.

To check this result and to exemplify its use, let us consider the situation when $n = 2$, for which the expressions of the variance are known from prior work [Jermaine et al. 2005a]. In this case we have, by expanding $\sum_{S \in \mathcal{P}(2)}$ in the order $\{\}, \{1\}, \{2\}, \{1,2\}$ and denoting the first relation by R and the second by S:

$$\begin{aligned} E[X^2] = \frac{1}{(|R|-1)(|S|-1)} \Bigg[&(|R|-p)(|S|-p) \left(\sum_{t \in R} \sum_{v \in S} f(t \bullet v) \right)^2 \\ &+ (p-1)|R|(|S|-p) \sum_{t \in R} \left(\sum_{v \in S} f(t \bullet v) \right)^2 \\ &+ (p-1)|S|(|R|-p) \sum_{v \in S} \left(\sum_{t \in R} f(t \bullet v) \right)^2 \\ &+ (p-1)^2 |R||S| \sum_{t \in R} \sum_{v \in S} f(t \bullet v)^2 \Bigg]. \end{aligned} \tag{7}$$

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

8.7 Analysis of Bernoulli Sampling

We now use Lemma 1 to analyze Bernoulli sampling. To perform Bernoulli sampling, an independent coin with probability of success $1/p$ is flipped for each tuple for each of the n relations, $R_1, \ldots, R_n$. Using the same approach as in the previous section, we need only alter the properties the random variables X_{t_i} have. In this case,

$$\begin{aligned} E[X_{t_i}] &= \frac{1}{p} \\ E[X_{t_i} X_{t'_i}] &= \begin{cases} \frac{1}{p} & \text{If } t_i = t'_i \\ \frac{1}{p^2} & \text{If } t_i \neq t'_i \end{cases} = \delta_{t_i t'_i} \frac{1}{p} + (1 - \delta_{t_i t'_i}) \frac{1}{p^2} \\ &= \frac{1}{p^2} [1 + (p-1) \delta_{t_i t'_i}]. \end{aligned}$$

By setting:

$$e_i = \frac{1}{p}, \qquad a_i = \frac{1}{p^2}, \qquad b_i = \frac{p-1}{p^2}$$

in the general analysis in Section 8.4, we readily obtain the following:

PROPOSITION 3. *For each $i \in \{1 : n\}$, let R'_i be Bernoulli sample obtained by flipping an independent fair p face coin for each tuple. Then, the estimate of the aggregate function:*

$$X = p^n \sum_{\{t_i \in R'_i | i \in \{1n\}\}} f(\{t_i | i \in \{1{:}n\}\}$$

is an unbiased estimate of the aggregate function and has variance:

$$\sigma^2(X) = \sum_{S \in \mathcal{P}(n)} (p-1)^{|S|} \cdot y_S - y_\emptyset$$

Section 8.4 readily provides efficient means to compute an unbiased estimate of $\sigma^2(X)$ for the appropriate values of constants e_i, a_i and b_i.

It is worth pointing out that the results in Section 8.4 can be applied to analysis of sampling with replacement. However, since we do not need this type of sampling for the work in this article, we do not pursue this development here.

9. 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? Is the theory correct?

—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

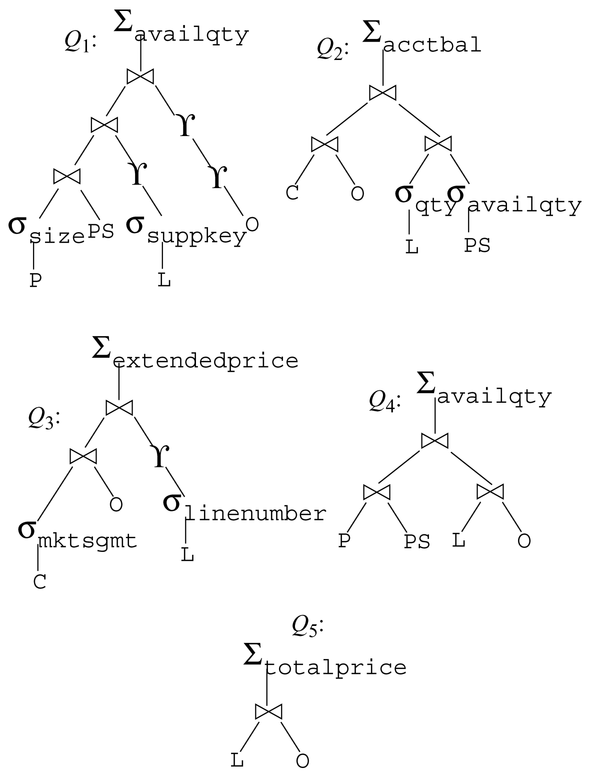

Q_1: Σ_{availqty}
σ_{size} P PS σ_{suppkey} L O
Q_2: Σ_{acctbal}
C O σ_{qty} L σ_{availqty} PS
Q_3: $\Sigma_{\text{extendedprice}}$
σ_{mktsgmt} C O $\sigma_{\text{linenumber}}$ L
Q_4: Σ_{availqty}
P PS L O
Q_5: $\Sigma_{\text{totalprice}}$
L O

Fig. 8. Test query plans.

are depicted in Figure 8. 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/rerandomize operator.

We also test two additional versions of Q_4 that include a range selection predicate on L. In the first case, the selection predicate accepts 5% of the tuples in L, and in the second case, the selection predicate accepts 25% of the tuples in L. In both of these versions, the selection predicate is over the `shipdate` attribute, and L has been partitioned into ten different ranges on this attribute. The engine is given no knowledge of how the selection predicate relates to the partitioning, and the relevant partitions are selected automatically. This is done to illustrate the manner in which “mini-chunk” strategy of Section 7 can correctly select those partitions that contain useful data, and how DBO can develop a sampling plan on-the-fly using the statistical methods described in that section,

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 9, we plot the relative confidence interval width produced by DBO as a function of time for these queries

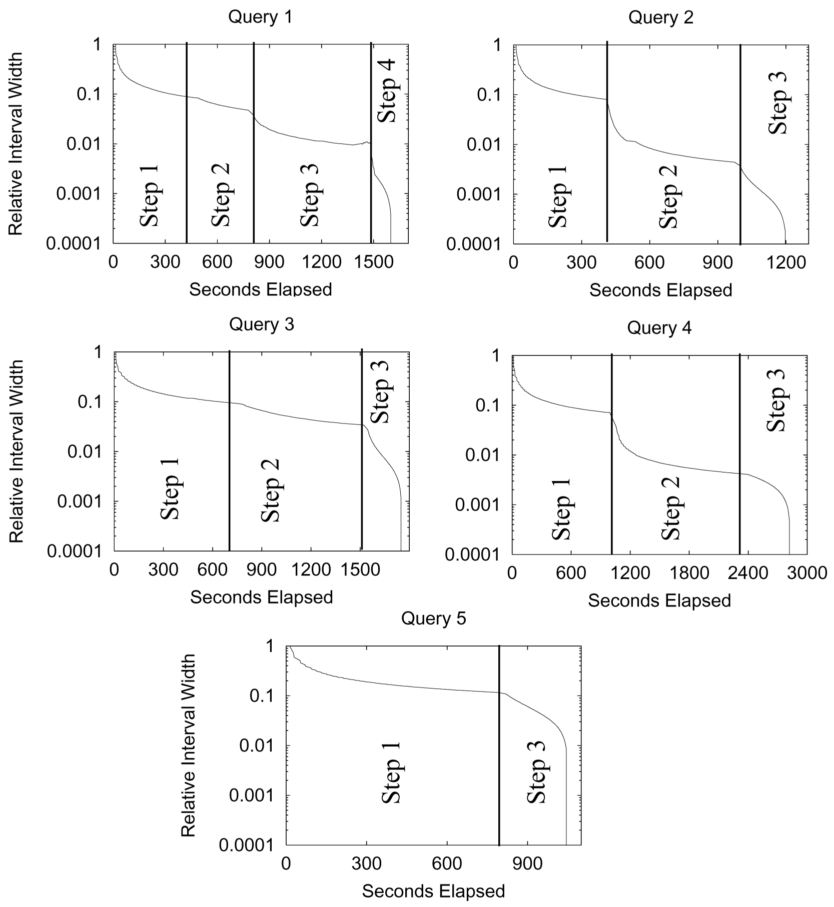
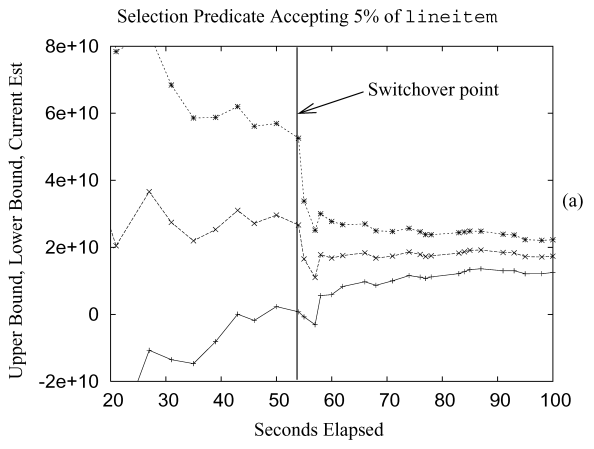


Fig. 9. Relative confidence interval width as a function of time for the five test query plans.

(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 plus or minus 2σ 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.

In Figure 10, we plot the actual confidence intervals given by DBO along with the current estimate as a function of time for the beginning of the first levelwise step, when the query is evaluated using a partitioned L table. The upper plot is for the selection predicate accepting 5% of L, and the lower is for the 25% case. In both the upper and the lower plots, we indicate the *switchover point* where the mini-chunk strategy that is used at the beginning of the query evaluation gives way to the optimized selection of only the relevant partitions.

To test the accuracy of the given confidence intervals, we regenerate 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



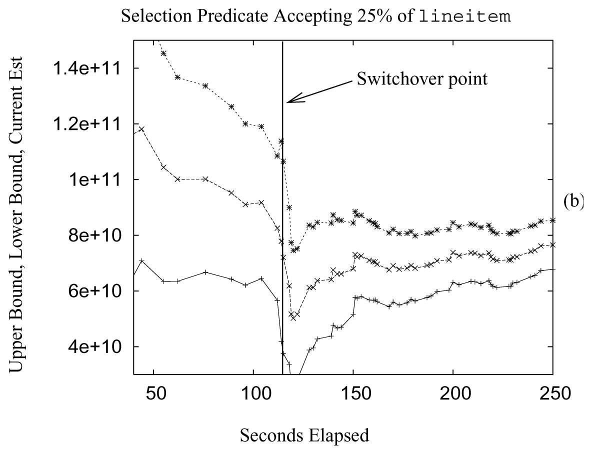


Fig. 10. Confidence intervals and the current estimate as a function of time for the two partitioned versions of Q_4.

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 in Figure 11.

Correct funtioning of DBO depends crucially on the correctness of the theoretical formulas and estimators developed in Section 8. The above check of the correctness of the confidence intervals is a macroscopic check that might still allow for small imperfections in the formulas for variance. To experimentally check the formulas in Section 8, we designed a large scale Monte-Carlo simulation consisting of 100,000 independent runs, designed to pinpoint the most minute discrepancies between theory and actual behavior in practice. The dataset used consists in 3 relations: R(A,B), S(B,C), T(C,D). The query executed is: `SELECT SUM(A*D) FROM R,S,T WHERE R.B=S.B AND S.C=T.C`. Attributes B and C are primary keys in relations R and T, respectively, and they were given values 1 to 1000 for each of the 1000 tuples. Attribute A of R was set to B,

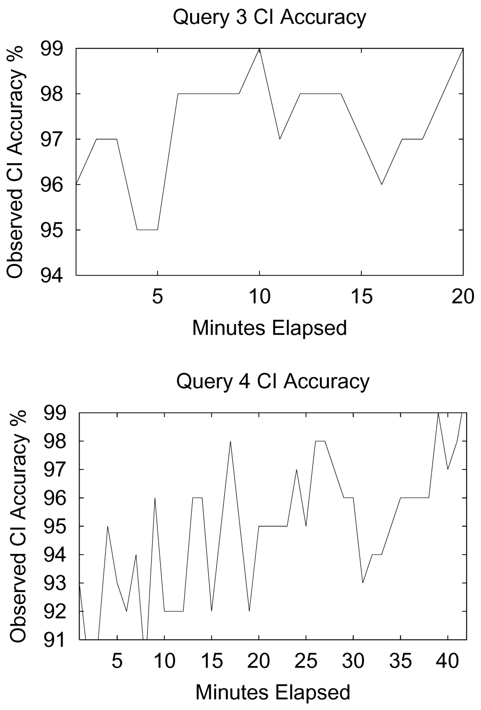


Fig. 11. Observed 95% interval accuracy over 100 independent query executions.

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

Fig. 12. Completion time of DBO vs. Postgres.

and attribute D to $\sqrt{C}$. The tuples (B,C) of S were generated using the random generator in Vitter and Wang [1999] that produces multidimensional skewed distributions—the generated relation had 7114 tuples. The setup for this experiment is not designed for a realistic scenario, but for a tough scenario in which the smallest discrepancy between theory and actual behavior will surface. The simulation results are depicted in Figure 13. The *theoretical variance* is the variance predicted using the exact y_S terms, the *average predicted variance* is the average of the variances predicted by the theory using the unbiased estimators $\tilde{Y}_S$ of y_S, and the *experimental variance* is the variance measured from the estimators themselves. From the experimental results, it is immediately apparent that there is no discrepancy between the theory and actual behavior.

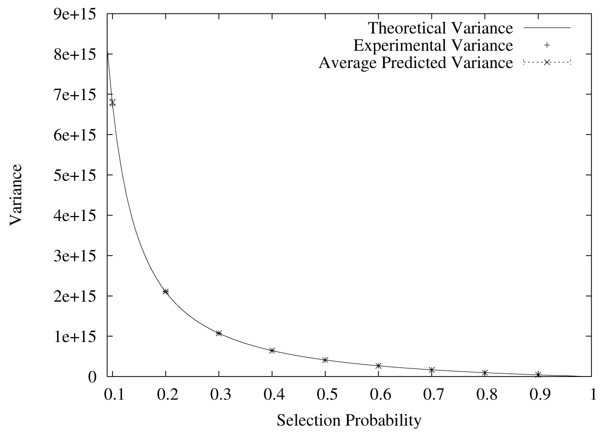


Fig. 13. Comparison of theoretical and experimental variances.

Finally, the time required for completing each query is given in Figure 12. 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 with 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 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.

It is also interesting to note the benefit of partitioning one or more of the input relations, if the partitioning can be done in such a way that relational selection predicates that are embedded in the queries will tend to use tuples in certain partitions more often than others. Consider Figure 10(a). In this plot, during the first 55 or so seconds of query execution, DBO is in the mini-chunk regime, where it is sampling all partitions evenly. At the point that it has read one full run from each relation, DBO switches over and begins sampling the partitions according to their importance. Since the vast majority of the 5% of the tuples accepted by the selection predicate are in a single partition, beyond 55 seconds DBO begins sampling that partition heavily, and the confidence intervals shrink dramatically. By the time 100 seconds have elapsed, the bounds are only 1/5 as wide as they were at 50 seconds.

It is perhaps even more interesting to consider Figure 10(b). Since the predicate is not as selective in this case, the bounds do not shrink as dramatically once DBO beings sampling the more important partitions heavily. However, note that the real answer to this query is at the lower end of the reported confidence interval range before the switchover point. On the other hand, beyond the switchover point, not only do the bounds begin to narrow very quickly, but the current estimate plunges downward toward the correct query answer. This illustrates in a concrete fashion the statistics behind the idea of a confidence bound. Once DBO has modified its sampling plan to take into account the partitioning, the confidence bounds narrow significantly, indicating that DBO becomes much more sure in the accuracy of the estimate. At the same time DBO becomes more sure in the quality of its estimate, the estimate itself becomes much more accurate, and DBO suddenly homes in on the correct answer.

Another significant finding is that these results generally 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 9, 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 11 gives strong evidence that the variance calculations described in this article, 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 11 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.

10. RELATED WORK

As discussed previously in this article, the work most closely related to the DBO engine is the previous work on online aggregation [Haas and Hellerstein 1999; Haas 1997; Hellerstein et al. 1999; Hellerstein et al. 1997] and the SMS join [Jermaine et al. 2005a]. Online aggregation has its roots in early work linking approximation with processing time [Özsoyoglu et al. 1992]. This article takes inspiration from, and extends, both. For example, the statistical results given in Section 8 extend the results of Haas et al. [Haas and Hellerstein 1999; Haas 1997; Haas et al. 1996] 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. [2005b] 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 with little cost.

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 [Olken 1993], is well known. The two papers most closely related to this article describe join synopses [Acharya et al. 1999], and Chaudhuri et al.'s work discusses important issues associated with sampling from joins [Chaudhuri et al. 1999]. 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.

11. FUTURE WORK AND CONCLUSION

This article 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 this article was specifically directed toward query processing (both algorithmic and statistical issues). To keep the article's scope at a manageable level, other important questions must be deferred to future work. These questions include the following:

(1) 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.

(2) 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.

(3) 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.

(4) 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 [Jermaine et al. 2005a], but more effort is needed to allow for truly scalable processing.

APPENDIX

A. PROOF OF LEMMA 1

In this section, we prove Lemma 1 by showing that for arbitrary a_i and b_i:

$$\sum_{\{t_i \in R_i | i \in \{1n\}\}} \sum_{\{t'_i \in R_i | i \in \{1n\}\}} \left(\prod_{i \in \{1n\}} a_i + b_i \delta_{t_i t'_i} \right) f(\{t_i\}) f(\{t'_i\})$$

$$= \sum_{S \in \mathcal{P}(n)} \prod_{i \in S^C} a_i \prod_{j \in S} b_j \sum_{\{t_i \in R_i | i \in S\}} \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t_j\}) \right)^2 .$$

This lemma is rather technical and needs a somewhat sophisticated proof by induction. A proof by induction is strictly necessary, since the left and right side of the statement are so different that there is no direct way to transform one into the other. Part of the difficulty is the fact that a special grouping that saves computation is performed at the same time as the removal of the Kronecker delta symbols, which makes the proof even more difficult.

A direct proof by induction would not work, since there is no direct connection between the formulas that appear on the right hand side for n and $n + 1$ relations. The fact that there are n relations appears throughout the formula either explicitly, in the case of $\mathcal{P}(n)$, or implicitly in S and S^C, since $S \in \mathcal{P}(n)$. To avoid these difficulties, we first slightly generalize the formulas for simplication, prove by induction the generalization, and then show that one of these more general formulas coincides with the left hand side (lhs) of the statement of the proposition. The simplified version of this formula will coincide with the right hand side (rhs) of the statement, thus giving us the desired result.

The main idea is to introduce the following set of functions that allow some of the sums that appear on the left hand side of the statement to be dropped:

$$F_k(\{t_l, t'_l | l \in \{k+1:n\}\})$$
$$= \sum_{\{t_i \in R_i | i \in \{1:k\}\}} \sum_{\{t'_i \in R_i | i \in \{1:k\}\}} \left(\prod_{i \in \{1:k\}} a_i + b_i \delta_{t_i t'_i} \right) f(\{t_i, t_l\}) f(\{t'_i, t'_l\}).$$

Notice that the function is parametrized by the tuples for which the summations are removed from the lhs of the statement. We cannot simply drop the sums and not keep the tuples as the parameters, since we cannot apply the aggregate functions f anymore because we do not have enough arguments. The index k of the functions indicates how many summations are kept (the number of summations is $2 * k$, since we always have two summations per relation). In order for these functions to be useful to prove the result at hand, it is important to notice that $F_n()$ is exactly the lhs of the statement.

We can now define another function $F'_k()$ as:

$$F'_k(\{t_l, t'_l | l \in \{k+1:n\}\}) = \sum_{S \in \mathcal{P}(k)} \prod_{i \in S^C} a_i \prod_{j \in S} b_j$$
$$\times \sum_{\{t_i \in R_i | i \in S\}} \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t_j, t_l\}) \right) \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t_j, t'_l\}) \right).$$

Notice that the shape of the simplification we want to prove is very similar to the rhs of the statement, with the difference that the sets S are subsets of $\mathcal{P}(k)$ instead of $\mathcal{P}(n)$, and two sums are multiplied in the inner part instead of a sum being squared. If we set $k = n$, this expressions become precisely the rhs of the statement, since the product of the two innermost sums becomes a square (the sums are identical).

In order for the above definitions to be useful for an inductive proof, a simple connection between functions with different indexes needs to be established. Inspecting the definition of F_k and comparing it with the definition for F_{k-1}, we can easily see that:

$$F_k(\{t_l, t'_l | i \in \{k+1:n\}) = \sum_{t_k \in R_k} \sum_{t'_k \in R_k} (a_k + b_k \delta_{t_k t'_k}) F_{k-1}(\{t_l, t'_l | l \in \{k:n\}), \quad (8)$$

since the term $(a_k + b_k \delta_{t_k t'_k})$ depends only on t_k and t'_k but not the rest of the tuples, so the term can be pulled out of all except the first two summations. What is left inside the double sums is exactly F_{k-1}.

Now, to establish a relationship between $F'_k()$ and $F'_{k-1}()$, we inspect the definition of $F'_k()$. The key to establishing a simple relationship is to rewrite sums that depend on $S \in \mathcal{P}(k)$, as sums that depend on $S \in \mathcal{P}(k-1)$. First, we observe that, to get the sets $\mathcal{P}(k)$, we have to include all sets in $\mathcal{P}(k-1)$ and all sets obtained by the union of $\{k\}$ with the sets in $\mathcal{P}(k-1)$. With this observation, for some function $\mathcal{F}(S, S^C)$, we can use the rewriting rule:

$$\sum_{S \in \mathcal{P}(k)} \mathcal{F}(S, S^C) = \sum_{S \in \mathcal{P}(k-1)} \mathcal{F}(S, S^C \cup k) + \sum_{S \in \mathcal{P}(k-1)} \mathcal{F}(S \cup \{k\}, S^C).$$

We need to have the complement S^C of S as an argument, since the set with respect to which the complement is taken is different for $S \in \mathcal{P}(k)$ and $S \in \mathcal{P}(k-1)$ (i.e., it is $\{1, \ldots, k\}$ and $\{1, \ldots, k-1\}$, respectively). Now, using this transformation and the definition of F'_k, we have:

$$\begin{aligned}
&F'_k(\{t_l, t'_l | l \in \{k+1:n\}\}) \\
&\quad = \sum_{S \in \mathcal{P}(k)} \prod_{i \in S^C} a_i \prod_{j \in S} b_j \\
&\qquad \times \sum_{\{t_i \in R_i | i \in S\}} \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t_j, t_l\}) \right) \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t_j, t'_l\}) \right) \\
&\quad = \sum_{S \in \mathcal{P}(k-1)} \prod_{i \in S^C \cup \{k\}} a_i \prod_{j \in S} b_j \\
&\qquad \times \sum_{\{t_i \in R_i | i \in S\}} \left(\sum_{\{t_j \in R_j | j \in S^C \cup \{k\}\}} f(\{t_i, t_j, t_l\}) \right) \left(\sum_{\{t_j \in R_j | j \in S^C \cup \{k\}\}} f(\{t_i, t_j, t'_l\}) \right) \\
&\qquad + \sum_{S \in \mathcal{P}(k-1)} \prod_{i \in S^C} a_i \prod_{j \in S \cup \{k\}} b_j \qquad\qquad (9) \\
&\qquad \times \sum_{\{t_i \in R_i | i \in S \cup \{k\}\}} \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t_j, t_l\}) \right) \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t_j, t'_l\}) \right) \\
&\quad = \sum_{t_k \in R_k} \sum_{t'_k \in R_k} a_k \sum_{S \in \mathcal{P}(k-1)} \prod_{i \in S^C} a_i \prod_{j \in S} b_j \\
&\qquad \times \sum_{\{t_i \in R_i | i \in S\}} \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t_j, t_l\}) \right) \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t'_j, t'_l\}) \right) \\
&\qquad + \sum_{t_k \in R_k} b_k \sum_{S \in \mathcal{P}(k-1)} \prod_{i \in S^C} a_i \prod_{j \in S} b_j \\
&\qquad \times \sum_{\{t_i \in R_i | i \in S\}} \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t_j, t_l\}) \right) \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t_j, t'_l\}) \right) \\
&\quad = \sum_{t_k \in R_k} \sum_{t'_k \in R_k} a_k F'_{k-1}(\{t_l, t'_l | l \in \{k:n\}\}) + \sum_{t_k \in R_k,\ t'_k = t_k} b_k F'_{k-1}(\{t_l, t'_l | l \in \{k:n\}\}).
\end{aligned}$$

By establishing these two equations linking $F_K()$ and F_{k-1}, and $F'_k()$ and F'_{k-1}, respectively, we are ready to do the actual proof by induction.

The precise statement we want to prove by induction on k is:

$$\forall \{t_l, t'_l | l \in \{k+1:n\}\}, \quad F_k(\{t_l, t'_l | l \in \{k+1:n\}) = F'_k(\{t_l, t'_l | l \in \{k+1:n\}).$$

Proving that $F_k() = F'_k()$ for all tuples that can be passed as arguments is crucial. Without this, the induction cannot be performed. Strictly speaking, the equality has to hold only for tuples from the corresponding relations, but the proof does not simplify with this assumption.

For the base case of the induction, we observe that:

$$\begin{aligned} F_0(\{t_l, t'_l \mid l \in \{1:n\}\}) &= f(\{t_l) f(t'_l) \\ F'_0(\{t_l, t'_l \mid l \in \{1:n\}\}) &= f(\{t_l) f(t'_l), \end{aligned}$$

since all the summations disappear because $\mathcal{P}(0) = \{\emptyset\}$. Immediately, we have $F_0() = F'_0()$ for all arguments, which proves the base case.

In the induction step, we assume that $F_{k-1}() = F'_{k-1}()$ for all possible values of arguments. We want to show that, using this assumption, the statement $F_k() = F'_k()$ can be proved for all arguments. Let $\{t_l, t'_l \mid l \in \{k+1:n\}\}$ be arbitrary tuples. Then, using the induction step, the recursive relationships in Equations 8 and 10, and properties of the Kronecker delta, we have:

$$\begin{aligned} & F_k(\{t_l, t'_l \mid i \in \{k+1:n\}) \\ &\quad = \sum_{t_k \in R_k} \sum_{t'_k \in R_k} (a_k + b_k \delta_{t_k t'_k}) F_{k-1}(\{t_l, t'_l \mid l \in \{k:n\}) \quad \text{(By Equation 8)} \\ &\quad = \sum_{t_k \in R_k} \sum_{t'_k \in R_k} a_k F_{k-1}(\{t_l, t'_l \mid l \in \{k:n\}) + \sum_{t_k \in R_k,\ t'_k = t_k} b_k F_{k-1}(\{t_l, t'_l \mid l \in \{k:n\}\}) \\ &\qquad\qquad \text{(Using } \delta_{t_k t'_k} \text{ simplif when interacting with } \textstyle\sum_{t_k} \sum_{t'_k}) \\ &\quad = \sum_{t_k \in R_k} \sum_{t'_k \in R_k} a_k F'_{k-1}(\{t_l, t'_l \mid l \in \{k:n\}) + \sum_{t_k \in R_k,\ t'_k = t_k} b_k F'_{k-1}(\{t_l, t'_l \mid l \in \{k:n\}\}) \\ &\qquad\qquad \text{(By induction Hypothesis)} \\ &\quad = F'_k(\{t_l, t'_l \mid l \in \{k+1:n\}\}) \quad \text{(By Equation 10).} \end{aligned}$$

By establishing this result, we conclude the proof by induction of statement $F_k() = F'_k()$. Since simple inspection reveals that $F_n()$ is the lhs of the statement and $F'_n()$ is the rhs of the statement, the equality in the statement of the proposition is proved.

B. COVARIANCE OF ESTIMATORS CONSIDERATIONS

For a strict statistical analysis of estimators in DBO, the covariance between various estimates that are combined needs to be considered as well. As mentioned in the text, DBO ignores the covariances, and estimates the variance of the overall estimate using only variances of individual estimates. These variances are computed using the formulas in Section 8. In this section, we provide an in-depth analysis of why ignoring the covariance term will not significantly change the confidence intervals provided by DBO. There are two different situations we have to consider, with different arguments required. The first is the behavior of covariance in the simple DBO (no partitioning), and the second is the behavior of covariance in the DBO with partitioning.

B.1 Simple DBO Covariances

The elementary estimates that are combined to produce the DBO estimates are based on one run of data from each of the relations. While the run membership is random, the runs cover the entire relation and do not overlap. For a given random partitioning into runs of the relations, when the estimates of all possible combinations of runs are summed, the precise result of the query is recovered, thus the variance of the sum of all these estimators is zero. Since the variance of each individual estimator is a positive quantity, the sum of all covariance terms is negative. This immediately suggests that, on average, the covariance terms between estimators is negative. This is confirmed by the results in Jermaine et al. [2005a], where a detailed analysis for the case when only two relations participate in the query reveals that most covariance terms are negative.

Using this argument, the overall variance would actually be smaller than the variance predicted by summing up variances of individual estimators. Ignoring the covariance, however, adds a margin of error for the variance estimation, making the confidence intervals slightly safer. Unless a large number of estimators are used—this is not the case in DBO, since the number of estimates is the number of runs per relation times the number of relations, vs. the number of runs to the power of the number of relations—the negative covariance will be small, since only a small, fraction of possible covariance terms participate.

It is worth mentioning that Lemma 1 is powerful enough to allow the exact computation of all covariance terms. Without going into details, the formula for covariances would depend on the same statistics Y_S as variance depends on. Since, as we argued previously, it is better to ignore these covariances to add the extra margin of error, we do not pursue the detailed analysis of covariances here.

B.2 Partitioned DBO Covariances

We have argued that the covariance between estimates for the regular DBO is negative, or a negligible positive. When one of the relations is partitioned, the same argument does not apply. Moreover, in Section 7.2, we have seen an example in which the estimates are positively correlated. In this section, we investigate the relationship between the covariance of two partitioning estimators and their variances. The goal is to argue that, in most situations, the covariances can be ignored.

The first question we ask is: between what types of terms might the covariance not be negligible or negative? Since, for each partition, we essentially run a regular DBO estimate, between estimates corresponding to the same partition, the covariance is negative. If estimates for different partitions do not share any runs, the covariance is again negative, since there is no covariance (independence) due to the runs coming from the relation being partitioned, and the runs from the other relations being negatively correlated. The terms that will have the strongest correlations are the terms corresponding to different partitions of the first (partitioned) relation, and the same runs for the other relations. When there is less sharing of runs, the negative correlation between runs of nonpartitioned relations will start to dominate.

In general, the number of covariances involved is quadratic, thus there is a worry that even small covariances have a big influence. For the estimates produced by partitioned DBO, since two estimates will share all the runs for the nonpartitioned relations only when the estimates are produced in succession, only a small number of covariance terms will contribute to the total covariance. In particular, a covariance contribution is incurred only when producing the first estimate with a new run from the partitioned relation that belongs to a different partition. In this situation, the first estimate will use the runs already in memory for all other relations. If the covariance term thus introduced is small when compared to the variances of the two terms, then the contribution can be neglected. In the rest of this section we analyze these particular covariance terms.

Instead of providing a general analysis for the covariance of two partitioning estimators, we provide an analysis only for the case when the query involves just two relations, an analysis similar to the two relation analysis for SMS estimates provided in Jermaine et al. [2005a]. While two relation queries are not very interesting for general DBO, they should suffice to get an idea of whether the covariance terms are negligible. Multiple relations would only complicate the analysis and, we conjecture, will be unlikely to change the result qualitatively. It is worth mentioning, however, that the methods we have introduced in this article can be used to analyze the covariance terms in their full generality. A generalization of Lemma 1 would be required to establish this result.

In order to analyze the covariance of partitioning estimators, we first establish the mathematical formula for the covariance, then we make some observations on the formulas to gain insights into the behavior of the covariance terms. The theoretical treatment in this section will closely follow notation and conventions in Section 8.6.

To analyze the covariance estimator, we let R_1 and R_2 be two partitions of relation R. Furthermore, we let S be a second relation. Samples without replacement from this relation would be denoted by R_1', R_2' and S'. An important observation is the fact that the samples R_1' and R_2' are independent, and the sample S' will be shared by the two estimators. With this, we can introduce sampling estimators for each partition of the form:

Proposition 4. *With the above notation, the sampling estimator:*

$$Z_i = p^2 \sum_{t \in R_i'} \sum_{u \in S'} f(t \bullet u)$$

has the property:

$$\begin{aligned} Cov(Z_1 Z_2) \;=\; & \frac{p-1}{|S|-1} \left[|S| \sum_{u \in S} \left(\sum_{t_1 \in R_1} f(t_1 \bullet u) \right) \left(\sum_{t_2 \in R_2} f(t_2 \bullet u) \right) \right. \\ & \left. - \left(\sum_{t_1 \in R_1} \sum_{u \in S} f(t_1 \bullet u) \right) \left(\sum_{t_2 \in R_2} \sum_{u' \in S} f(t_2 \bullet u') \right) \right]. \end{aligned}$$

Proof. First, we have:

$$\begin{aligned} Z_i \;&=\; p^2 \sum_{t \in R_i'} \sum_{u \in S'} f(t \bullet u) \\ &=\; p^2 \sum_{t \in R_i} \sum_{u \in S} X_{i,t} Y_u f(t \bullet u), \end{aligned}$$

where we make use of random variables $X_{i,t}$ and Y_u to indicate whether tuples in relations R_i and S are present in the respective samples. Since Cov $(Z_1, Z_2) =$

$E[Z_1Z_2] - E[Z_1]E[Z_2]$. We firs estimate $E[Z_1Z_2]$:

$$E[Z_1Z_2] = p^4 \sum_{t_1\in R_1}\sum_{t_2\in R_2}\sum_{u\in S}\sum_{u'\in S} E\left[X_{1,t_1}X_{2,t_2}\right] E[Y_uY_{u'}]\, f(t_1 \bullet u) f(t_2 \bullet u'),$$

where we use the linearity of expectation and the fact that the samples of R_i and S are independent (thus the expectation factorizes). Now, since the sampling of each partitioning is independent (guaranteed by the sampling process), we have $E[X_{1,t_1}X_{2,t_2}] = E[X_{1,t_1}]E[X_{2,t_2}] = \frac{1}{p^2}$. As in Section 8.6, for random variables Y_u, we have:

$$E[Y_uY_{u'}] = \frac{1}{p^2(|S|-1)}[(|S|-p) + |S|(p-1)\delta_{uu'}].$$

Replacing all of these equations in the above formula for $E[Z_1Z_2]$ we have:

$$\begin{aligned}
&E[Z_1Z_2]\\
&= p^4 \sum_{t_1\in R_1}\sum_{t_2\in R_2}\sum_{u\in S}\sum_{u'\in S} \frac{1}{p^2}\frac{1}{p^2(|S|-1)}[(|S|-p)\\
&\quad + |S|(p-1)\delta_{uu'}] f(t_1\bullet u) f(t_2\bullet u')\\
&= \frac{|S|-p}{|S|-1}\sum_{t_1\in R_1}\sum_{t_2\in R_2}\sum_{u\in S}\sum_{u'\in S} f(t_1\bullet u) f(t_2\bullet u')\\
&\quad + \frac{|S|(p-1)}{|S|-1}\sum_{t_1\in R_1}\sum_{t_2\in R_2}\sum_{u\in S} f(t_1\bullet u) f(t_2\bullet u)\\
&= \frac{|S|-p}{|S|-1}\left(\sum_{t_1\in R_1}\sum_{u\in S} f(t_1\bullet u)\right)\left(\sum_{t_2\in R_2}\sum_{u'\in S} f(t_2\bullet u')\right)\\
&\quad + \frac{|S|(p-1)}{|S|-1}\sum_{u\in S}\left(\sum_{t_1\in R_1} f(t_1\bullet u)\right)\left(\sum_{t_2\in R_2} f(t_2\bullet u)\right).
\end{aligned}$$

Now, by observing that, by linearity of expectation,

$$E[Z_i] = \sum_{t\in R_i}\sum_{u\in S} f(t\bullet u),$$

and with simple algebraic manipulations, we immediately get the result in the statement. □

Based on the formula for the covariance in Proposition 4, we can make the following observations:

(1) If the attribute used to produce the partitioning of R is correlated with the join attributes in the query, the covariance will tend to be negative. This is the case, because the first term in the covariance formula will have the tendency to be small, since for a particular value of u, either $f(t_1 \bullet u)$ or $f(t_2 \bullet u)$ will have the tendency to be 0, since matching tuples would be found primarily in either R_1 or R_2 due to the correlation between the join attribute and the partitioning attribute.

(2) If the aggregate function is relatively uniformly distributed with respect to the join attribute, the two large terms in the formula for the covariance will

have roughly the same value and will cancel each other out overall, and the covariance will be, at most, a small positive quantity.

(3) Covariance can be large when the distribution of the aggregate function is skewed with respect to the join aggregate, and when the partitioning is not correlated with the join attribute. In this case, the covariance can be as large as the variances of Z_1 and Z_2.

Based on this discussion, it seems that, in most situations, ignoring the covariance terms will have no adverse effect. In the pathological situation where multiple partitions contain the only few tuples that match tuples from the nonpartitioned relations, the covariance can be significant. Even in this case, the mistake in ignoring the covariance terms is not very large, since the number of such covariance terms is, as we previously explained, small.

REFERENCES

Acharya, S., Gibbons, P. B., Poosala, V., and Ramaswamy, S. 1999. Join synopses for approximate query answering. *SIGMOD Rec. 28*, 2, 275–286.

Antoshenkov, G. 1992. Random sampling from pseudo-ranked b+ trees. In *Proceedings of the 18th International Conference on Very Large Data Bases.* (VLDB'92), Vancouver, Canada. Morgan Kaufmann, San Francisco, CA, 375–382.

Chaudhuri, S., Motwani, R., and Narasayya, V. R. 1999. On random sampling over joins. In *Proceedings of the 1999 SIGMOD International Conference (SIGMOD'99)*. Philadelphia, PA. ACM, New York, NY, 263–274.

Cochran, W. G. 1977. *Sampling Techniques, 3rd Edition*. John Wiley.

Dittrich, J.-P., Seeger, B., Taylor, D. S., and Widmayer, P. 2002. Progressive merge join: a generic and non-blocking sort-based join algorithm. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB'02)*, Hong Kong, China, VLDB Endowment, 299–310.

Dittrich, J.-P., Seeger, B., Taylor, D. S., and Widmayer, P. 2003. On producing join results early. In *Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. San Diego, CA. ACM, New York, NY, 134–142.

Dobra, A. 2005. Histograms revisited: when are histograms the best approximation method for aggregates over joins? In *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. Baltimore, MD. ACM, New York, NY, 228–237.

Haas, P. J. 1997. Large-sample and deterministic confidence intervals for online aggregation. In *Proceedings of the 9th International Conference on Scientific and Statistical Database Management (SSDBM)*. IEEE Computer Society, 51–63.

Haas, P. J. and Hellerstein, J. M. 1999. Ripple joins for online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Philadelphia, PA, ACM, New York, NY, 287–298.

Haas, P. J., Naughton, J. F., Seshadri, S., and Swami, A. N. 1996. Selectivity and cost estimation for joins based on random sampling. *J. Comput. Syst. Sci. 52,* 3, 550–569.

Hardy, G., Littlewood, J., and Polya, G. 1988. *Inequalities*. Cambridge University Press.

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. *Computer 32,* 8, 51–59.

Hellerstein, J. M., Haas, P. J., and Wang, H. J. 1997. Online aggregation. *(SIGMOD). Rec. 26*, 2, 171–182.

Jermaine, C., Arumugam, S., Pol, A., and Dobra, A. 2007. Scalable approximate query processing with the DBO engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Beijing, China. ACM, New York, NY, 725–736.

Jermaine, C., Dobra, A., Arumugam, S., Joshi, S., and Pol, A. 2005a. A disk-based join with probabilistic guarantees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Baltimore, MD. ACM, New York, NY, 563–574.

Jermaine, C., Dobra, A., Pol, A., and Joshi, S. 2005b. Online estimation for subset-based sql queries. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*. Trondheim, Norway. VLDB Endowment, 745–756.

Luo, G., Ellmann, C., Haas, P. J., and Naughton, J. F. 2002. A scalable hash ripple join algorithm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, New York, NY, 252–262.

Olken, F. 1993. Random sampling from databases. Ph.D. thesis, U. of California, Berkeley.

Olken, F. and Rotem, D. 1989. Random sampling from b+ trees. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB)*. Amsterdam, The Netherlands. Morgan Kaufmann, San Francisco, CA, 269–277.

Özsoyoglu, G., Du, K., Swamy, S. G., and Hou, W.-C. 1992. Processing real-time, non-aggregate queries with time-constraints in case-db. In *Proceedings of the 8th International Conference on Data Engineering (ICDE)*, IEEE Computer Society, 410–417.

Shao, J. 1999. *Mathematical Statistics*. Springer-Verlag.

Shapiro, L. D. 1986. Join processing in database systems with large main memories. *ACM Trans. Database Syst. 11,* 3, 239–264.

Stefanov, S. 2001. *Separable Programming*. Applied Optimization, vol. 53. Kluwer Academic Publishers.

Vitter, J. S. and Wang, M. 1999. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, New York, NY, 193–204.

Received October 2007; revised April 2008; accepted July 2008