Turbo-Charging Estimate Convergence in DBO

Alin Dobra[1], Chris Jermaine[1,2], Florin Rusu[1], Fei Xu[1]
adobra@cise.ufl.edu, cjermain@cs.rice.edu, frusu@cise.ufl.edu, feixu@cise.ufl.edu
[1]University of Florida, [2]Rice University

ABSTRACT

DBO is a database system that utilizes randomized algorithms to give statistically meaningful estimates for the final answer to a multi-table, disk-based query from start to finish during query execution. However, DBO's "time 'til utility" (or "TTU"; that is, the time until DBO can give a useful estimate) can be overly large, particularly in the case that many database tables are joined in a query, or in the case that a join query includes a very selective predicate on one or more of the tables, or when the data are skewed. In this paper, we describe *Turbo DBO*, which is a prototype database system that can answer multi-table join queries in a scalable fashion, just like DBO. However, Turbo DBO often has a much lower TTU than DBO. The key innovation of Turbo DBO is that it makes use of novel algorithms that look for and remember "partial match" tuples in a randomized fashion. These are tuples that satisfy some of the boolean predicates associated with the query, and can possibly be grown into tuples that actually contribute to the final query result at a later time.

1. INTRODUCTION

A common complaint regarding real-world data warehousing installations is that they give the user no meaningful feedback regarding the final result until the query runs to completion. This is problematic for several reasons. For example, it makes debugging queries difficult, because there is no sanity check on the final query result. This is one of the reasons that users often subsample large database tables, then run their queries over the sampled data before running them over the entire database. Another problem related to lack of feedback is that it discourages users from interactively exploring the database data. When the goal is finding unexpected trends or relationships, one may have to try out a large number of exploratory queries, most of which return nothing of interest. A user is unlikely to issue a query over a multi-terabyte warehouse, look at the result, use the result to issue a second query, look at the result, and so on, in interactive fashion, if evaluating each query takes minutes or hours.

A notable line of work that attempted to address this problem is *online aggregation* (OLA), which first began more than ten years ago [6]. In OLA, the answer to a statistical/aggregate database query is estimated from the time that the query fires up. The estimate is bracketed by statistically meaningful confidence bounds of the form, "with probability p, the final answer is between low and $high$." This is achieved by clustering the data in a statistically random fashion on disk, so that sequential processing of database tuples results in a random sample of all of the tuples in the database.

The most recent work on OLA has been undertaken in the context of the DBO database system [15, 9]. DBO is unique in that unlike the original OLA proposals—which can only provide an estimate for the final query result as long as at last one table that is being joined fits entirely in memory—DBO is saleable. DBO is able to provide confidence bounds from start to finish, even for complex query plans requiring external-memory joins of multiple, disk-based tables.

Does DBO Always Converge Quickly? The utility of an OLA system should ultimately be measured by its ability to quickly provide tight bounds on the final query result. That is, DBO should have a short "time 'til utility". If the bounds that DBO supplies are wide (that is, if $\frac{high-low}{est}$ is a large value, even after much of the query plan has been processed) then the ability to provide an estimate will be of little use, because the estimate is of poor quality.

In practice, we have found that DBO often works quite well, though there is a class of queries where DBO's convergence to the actual query result can be too slow for comfort: queries that are highly selective, or that query highly skewed data, or that involve joins of many database tables. Imagine that DBO is used to answer a query of the form:

SELECT SUM($f(t_1 \bullet t_2 \bullet ... \bullet t_n)$)
FROM TABLE1 AS t_1 TABLE2 AS t_2 ... TABLEn AS t_n
WHERE $P(t_1 \bullet t_2 \bullet ... \bullet t_n)$

In this query, $\bullet$ is the concatenation operator, f is an arbitrary function over the tuple created by concatenating t_1 through t_n, and P is some boolean predicate. To guess the answer to such a query, DBO relies on a statistical process to "get lucky" and find various combinations of tuples that happen to be buffered in-memory at a given instant, and are also accepted by the predicate P. If n is large or P is very selective, then finding such combinations in memory at any given instant may be quite unlikely. If $n = 4$ and DBO has enough memory to buffer $1/50$ of each input relation in main memory, then DBO has a $1/(50^4)$ or around 1×10^{-7} chance of being able to construct any given output tuple at a given instant; if there are only thousands of tuples in a result set, then few output tuples will ever be discovered and DBO's accuracy will suffer accordingly.

Turbo-Charging Estimate Convergence in DBO. This paper is based upon the observation that it is possible to "turbo charge" DBO's estimation convergence in precisely those problem cases by

VLDB '09, August 24-28, 2009, Lyon, France

taking advantage of *partial* result tuples—combinations of tuples from subsets of the input relations that cannot satisfy P directly because they are not constructed from all n input relations, but might later be combined with other tuples to eventually satisfy P. For example, imagine that $n = 4$ and DBO happens to find t_1 and t_2 that satisfy the join predicate over TABLE1 and TABLE2, but DBO is not lucky enough to find an associated t_3 and t_4 in memory at the same time. The improved version of DBO described in this paper (called *Turbo DBO*) remembers the partial result $t_1 \bullet t_2$. As Turbo DBO encounters more tuples, it tries to find appropriate matches for $t_1 \bullet t_2$ from TABLE3 and TABLE4 in an incremental fashion. The fact that Turbo DBO searches for early result tuples incrementally greatly increases the chance of finding combinations of tuples such that $P(t_1 \bullet t_2 \bullet t_3 \bullet t_4)$ evaluates to `true`. As a result, estimation accuracy can be radically improved. In many cases, Turbo DBO can produce confidence bounds that are orders of magnitude smaller than the ones produced by the original version of DBO.

The contributions of this paper include:

- Turbo DBO uses partial matches to implement a novel estimation process, which results in a significantly boost to estimation accuracy. This estimation process encompasses several novel techniques, such as an optimized building order for partial matching tuples, a subsampling process to control memory usage, and a tuple "timestamping" abstraction to control the randomization in the system.

- Turbo DBO utilizes a system architecture that could conceivably be added to any database system, where a software component called the *in-memory join* sits outside of the normal data access path, and "snoops" for tuples that happen to contribute to the final query result. This co-processor-like architecture is attractive, in that it need not slow down the rest of the system.

- We benchmark a ground-up implementation of Turbo DBO, and find that for queries joining five large tables, Turbo DBO decreases the "time 'til utility" by almost 80% compared to the original DBO. For a more traditional warehouse workload featuring a central fact table, the reduction is nearly 30%.

Related Work. Sampling and randomized algorithms have a very long history in databases; the best-known early work is the PhD thesis of Olken [14] and a series of papers from Case Western in the early 1990's [7, 8]. Our work on Turbo DBO is a continuation of a line of work on OLA in the database management literature [5, 13, 6, 11, 15, 9]. The focus in Turbo DBO is on the systems-oriented issues that are important when one designs a database system from the ground-up to utilize randomized algorithms. Aside from OLA, there is relatively little work specifically aimed at sampling- or randomization-based systems design. Aside from DBO [9], and the original OLA work at Berkeley and IBM [6], the two most notable projects were the AQUA project from Bell Labs [1] and Derby/S at Dresden [12]. However, the latter two systems do not aim to combine sampling-based approximation with an industrial-strength database engine, as is the goal of this paper.

2. PRELIMINARIES AND SCOPE

The remainder of the paper considers multi-table aggregate queries of the form given in the Introduction. Both the function f and the predicate P can be arbitrary, as long as they are "memory-less"; that is, both f and P operate only over one tuple $(t_1 \bullet t_2 \bullet ... \bullet t_n)$

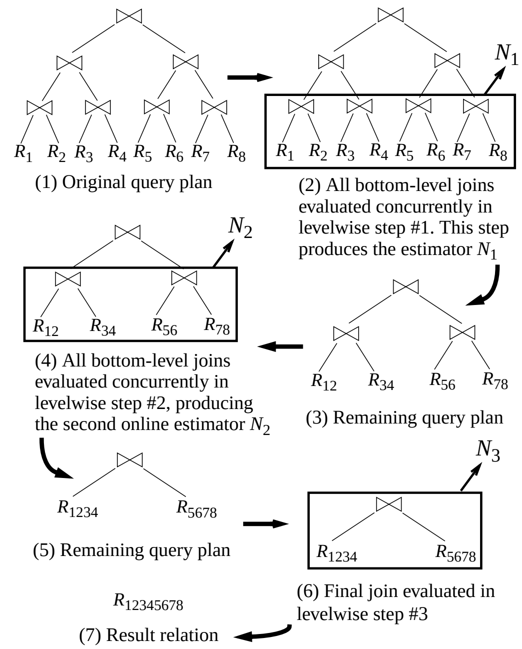

Figure 1: Levelwise query evaluation in DBO.

and no state can be saved across tuples. Fortunately, this is not too restrictive and handles the classic "select-project-join" class of queries, but it does rule out correlated sub-queries in the WHERE clause and a DISTINCT keyword in the SELECT clause. Handling such queries is an interesting research problem in and of itself [4, 11], and beyond the scope of the paper.

While the paper does not explicitly discuss aggregate functions besides SUM, other functions such as COUNT, AVERAGE, STD_DEV and VARIANCE can all be handled easily. COUNT is a special case of SUM where $f(.) = 1$ always. The other functions are handled by answering several aggregate queries simultaneously [3] (for example, AVERAGE is a ratio of a SUM and a COUNT). MIN and MAX require special consideration [17].

Finally, GROUP BY queries can be handled using the methods in this paper by simply treating each group as a separate query and running all queries simultaneously; then all of the estimates are presented to the user. For each group, a version of P is used that accepts only tuples from that particular group.

3. QUERY PROCESSING IN DBO

We begin by reviewing at a high level the basic query processing techniques employed by DBO to both (1) process analytic queries efficiently from start-up through completion, and (2) give a statistically meaningful guess as to the final query result the whole way. For brevity, many details are glossed over and can be found in the earlier paper on DBO [9].

3.1 The Levelwise Step

In DBO, a query plan is processed by running all of the relational operations attached directly to the leaves of the query plan tree at the same time, in a carefully choreographed fashion. This is called a *levelwise step*. All of the operations in a levelwise step communicate with one another to look to see if they can piece together tuples that satisfy all selection/join predicates in the underlying query. As such "lucky" tuples are discovered, they are used to update an estimate for the final query result, which is modeled as a random variable N_i. Here, i is the number of the current levelwise step. N_i is provably unbiased, which means that DBO is "correct on average". That is, $E[N_i] = Q$, where Q is the final query result and E denotes the expectation of N_i. As more data are processed, it becomes more likely that the levelwise step will discover "lucky" output tuples. The effect of this is that as the levelwise step progresses, the variance $Var(N_i)$ decreases.

When the levelwise step completes (that is, when all of the operations at the leaf level of the query plan finish), N_i is frozen. At this point, the operations attached to the leaves of the query plan are effectively removed, and replaced with the intermediate relations that they produced. Then the relational operations at the *next* level of the query plan begin operation, and a new levelwise step is begun. This process is repeated for each level of the query plan, until the final, exact query answer is computed. The overall process of executing a query plan in DBO is illustrated in Figure 1.

After l levels of the plan have been completed, the actual estimate given to the user is $N = (\sum_{i=1}^{l} w_i N_i)$ for some set of weights $\{w_1 ... w_l\}$ such that $(\sum_{i=1}^{l} w_i) = 1$. Since each N_i is unbiased it holds that N is unbiased. By choosing the weights carefully, the variance of N can be minimized.

3.2 The Estimation Process in Detail

One of the most important aspects of query processing in DBO is how "lucky" output tuples are discovered during a levelwise step, and how they are used to produce N_i. This is the particular issue that is considered in depth in the remainder of the paper.

The search for "lucky" output tuples in DBO is conducted as follows. The tuples that are input into each relational operation in a levelwise step are always streamed into each operation in statistically random order (for details of how this randomness is achieved, we refer the reader to the original DBO paper). Since tuples are processed in random order, the set of tuples that each relational operation has in memory at a given instant is generally a statistically random subset of all of the tuples that the operation will be asked to process. DBO requires that relational operations such as joins make the tuples that they process visible to the rest of the system. By joining these random samples and scaling up the result, DBO produces an unbiased estimator N_i for Q.

This is best illustrated with an example. Imagine that we are processing a join of four relations R_1, R_2, R_3, and R_4, with the SQL `WHERE` predicate "`WHERE` R_1`.a =` R_2`.a AND` R_2`.b =` R_3`.b AND` R_3`.c =` R_4`.c`"; Q is a a sum over R_1`.b` for all the tuples accepted by the `WHERE` clause. The current levelwise step processes two joins concurrently, where join A is $R_1 \bowtie R_2$, and join B is $R_3 \bowtie R_4$. Both joins are implemented as sort-merge joins, and both joins have enough memory to buffer $p \times 100\%$ of their respective input relations in memory at a given instant.

In DBO, both joins start up by streaming tuples into their internal buffers, just like they would in a classical database system. When all buffers fill, the contents of the buffers are indexed via a DBO software component called the *in-memory join* (IMJ). The IMJ maintains an in-memory, hash-based index that allows it to quickly locate tuples that will contribute to Q. In our example, the IMJ would create a separate index on each of the join attributes in the `WHERE` clause: R_1`.a`, R_2`.a`, R_2`.b`, R_3`.b`, R_3`.c`, and R_4`.c`. After the hash indexes are built, the IMJ uses them to locate combinations of tuples in the buffers that are accepted by the query's `WHERE` clause. After the IMJ finds all such combinations, it sums up R_1`.b` for all of these combinations, and multiplies this sum by $1/p^4$ to obtain N_i. N_i is unbiased for Q since the buffer for each input relation contains $(p \times 100)\%$ of the tuples in the input relation (for $p < 1$). Thus, on expectation, the IMJ will have discovered

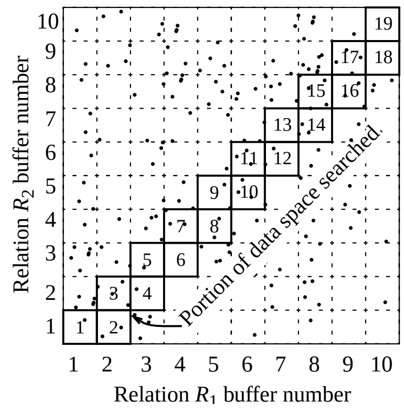

Figure 2: Searching for "lucky" output tuples in a two-relation join in classic DBO. The numbers on the grid cells show the progression of the search.

$(p^4 \times 100)\%$ of Q.

Once the IMJ has searched for these "lucky" output tuples, join A is allowed to replace its R_1 buffer with another $(p \times 100)\%$ fraction of R_1. The IMJ then indexes those new tuples from R_1, and again uses its various indexes to try to discover more "lucky" output tuples that may be found with the new R_1 buffer. With this new search, the IMJ has now doubled its chances of finding any given output tuple, and so by summing up the total aggregate value for R_1`.b` over all discovered output tuples and multiplying the sum by $1/(2 \times p^4)$, it obtains an updated value for N_i.

The process of allowing a join to replenish one of its buffers, indexing the buffer, and looking for "lucky" output tuples is then repeated until the levelwise step has read all of its input data. In our example, the IMJ would then allow join A to replace its buffer for relation R_2. After this is done, the IMJ searches for "lucky" output tuples, and updates N_i by multiplying the total aggregate value seen so far by $1/(3 \times p^4)$. The process continues until all the tuples from each input relation have been pushed through the buffers.

4. CONVERGENCE IN DBO

To understand what governs "time 'til utility' in DBO, and obtain some intuition behind Turbo DBO's improved estimation process, it is critical to understand the key issues governing DBO's convergence speed.

4.1 What Governs Convergence Speed?

A visualization of DBO's randomized search process for a two-relation join $R_1 \bowtie R_2$ with $p = \frac{1}{10}$ is depicted in Figure 2. This figure depicts a two-dimensional grid, where all of the tuples from R_1 are randomly arranged on the x-axis, and all of the tuples from R_2 are randomly arranged on the y-axis. The dots in the grid are "hits", or $t_1 \in R_1$, $t_2 \in R_2$ combinations where $P(t_1 \bullet t_2) =$ `true`. The final answer to an aggregate query over this join is the result of applying the aggregate function to each and every hit in the grid. The lines along the x-axis partition R_1 into $1/p$ different subsets of tuples, where each subset will be buffered at once in its entirety as the levelwise step processes its data. The grid lines along the y-axis show a similar partitioning for R_2.

As a levelwise step progresses, every time that a buffer is refilled, the IMJ searches all buffers for "lucky" result tuples. This has the effect of searching one cell in the grid. For example, as the levelwise step running $R_1 \bowtie R_2$ begins, the first buffer from R_1 will be paired with the first from R_2. In this case, the first grid

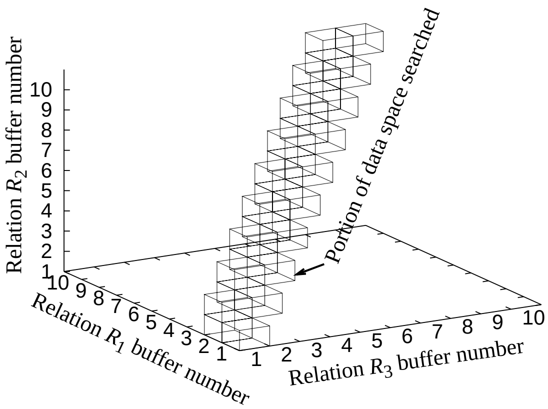

Figure 3: Searching for "lucky" output tuples in a three-relation join in classic DBO.

cell (labeled "1" in Figure 2) is searched. Then the buffer of R_1 is flushed to disk, and the second buffer from R_1 is paired with the first from R_2. This searches the grid cell labeled "2". Then the first buffer from R_2 is flushed and the second buffer from R_2 is paired with the second from R_1. This searches the grid cell labeled "3". This is repeated until both relations have been scanned in their entirety. Since in this example $p = 1/10$, by the time the levelwise step completes, exactly 19 cells have been searched for "lucky" tuples, as depicted in Figure 2.

The number of cells that are searched for "lucky" tuples is of critical importance. Ignoring certain messy details, the inaccuracy (variance) of any sampling-based aggregate estimator generally decreases in proportion with the expected number of output tuples that are used in the estimate. Since the expected number of output tuples increases linearly with the area or volume of the data space that has been searched, the number of grid cells searched controls the convergence rate of the estimate produced by DBO. Since each cell in the grid covers the same area (that is, the probability that a given output tuple is in a given cell is p^2 in the case of a two-table join), in DBO the variance of the estimator N_i decreases (approximately) proportionally with the number of cells that have been processed. That is, if σ^2 denotes the variance of the estimate that is produced by searching a single cell, then after m cells have been searched, $Var(N_i) \approx \sigma^2/m$. If central-limit-theorem-based confidence bounds are used, then the width of the resulting confidence bound is proportional to the square root of the variance, and so the bound width is approximately proportional to $1/\sqrt{m}$.

4.2 So What's the Problem?

In the example described above, DBO should converge quite quickly. The fraction p is not too small and only a single join of two relations is considered. By the time the levelwise step has completed, $\frac{19}{100}$ or 19% of the data space has been searched, and so on expectation 19% of the tuples satisfying the predicate P will be discovered. In the realistic case where there are thousands to millions of hits in the entire grid, a 19% sample will obtain hundreds of "lucky" tuples and tend to give a very good result.

The difficulty for DBO is when the expected number of "lucky" output tuples becomes small. This can happen when there are not many tuples to discover (that is, when the underlying query is highly selective), or when the fraction of the data space that is searched is very small. The fraction can be very small for two reasons. First, p can be small, either due to having a very large input data set or a small amount of available memory. In this case, the fraction of the data space that is searched shrinks. Second, for a fixed value of p, the fraction of the data space that is searched decreases exponentially when increasing the number of relations. For example, consider Figure 3. In this case, $p = \frac{1}{10}$, and by the time all of the input streams have been totally processed, only 28 out of the 1,000 cells in the corresponding 3-D search space are checked – or just 2.8%, compared with 19% in the case of a 2-way join using the same value of p. In a four-way join using the same value of p, the fraction of the data space searched decreases to just 0.37%.

5. ESTIMATION IN TURBO DBO

Intuitively, one of classic DBO's biggest problems is that when a relational operation re-fills its buffer with new tuples from one of the input relations, the IMJ simply forgets everything about the older tuples. No state is remembered across buffer flushes.

Turbo DBO uses a very different strategy. Rather than using the IMJ to simply index the content of the various relational operations' buffers and passively look for "lucky" output tuples, in Turbo DBO the IMJ has a memory budget of its own to buffer data. If the current levelwise step is processing a join of n input relations $R_1, R_2, ..., R_n$, the IMJ uses its internal memory to maintain n different buffers. The n^{th} or last buffer contains "lucky" output tuples from $R_1 \times R_2 \times ... \times R_n$ that are accepted by the `WHERE` predicate P, and hence actually contribute to Q. The IMJ also buffers "partial" results, or tuples from a cross product of a subset of the relations that could eventually contribute to the result. In general, the i^{th} buffer contains a set of tuples that belong to $R_1 \times R_2 \times ... \times R_i$ and are accepted by P [1]. We will subsequently refer to these partial results as "chains", since they are strings of tuples chained together using the join predicates encoded by P.

During query processing, tuples that enter into a levelwise step are streamed into the relational operation that is processing them, just as they would be in classic DBO or in any database system. Copies of those tuples are pipelined into the IMJ. As we will describe in detail subsequently, tuples are pipelined into the IMJ in such a way that the timestamp $TS(t)$ denoting tuple t's logical arrival time—$TS(t)$ takes a value from zero (the beginning of the levelwise step) to one (the end of the levelwise step)—can be viewed as statistically random and uniformly distributed from zero to one, with tuples added to the IMJ in ascending order of $TS(t)$.

When the IMJ obtains a tuple t_1 from relation R_1, if $P(t_1) =$ `true`, the IMJ adds t_1 to the first buffer. When the IMJ obtains a tuple t_{i+1} from relation R_{i+1} for $i > 0$, the IMJ goes to its i^{th} buffer, and sees if there is any tuple $(t_1 \bullet t_2 \bullet ... \bullet t_i)$ in this buffer where $t = (t_1 \bullet t_2 \bullet ... \bullet t_i \bullet t_{i+1})$ is accepted by the predicate P. If the IMJ can construct such a t, then t is added to $(i+1)^{th}$ buffer. If $i + 1 = n$, then the IMJ adds $f(t)$ to the total aggregate value seen so far. In the remainder of the paper, we denote this running sum with an upper-case sigma (Σ). Σ is then used to provide an unbiased guess for the query result Q.

Intuitively, the i^{th} buffer contains chains of tuples from each of the first i relations, where each chain in the buffer is accepted by all of the applicable join and selection predicates in P. When a new tuple is accepted by the IMJ, the IMJ tries to attach it to the end of an existing chain in order to grow the chain. If the IMJ is successful, then it buffers the new, longer chain for later use. If the IMJ manages to build a chain that spans all n relations, then it has discovered a new, "lucky" output tuple. Since chains are constructed in order (with relation R_1 first, R_2 second, and so on)

[1] Strictly speaking, P only accepts or rejects tuples from the cross product of *all* of the input relations. For notational simplicity, we assume that P applied to a tuple t that is "missing" one or more attributes returns `true` if and only if it would be possible to satisfy P using t by adding some set of additional attribute values.

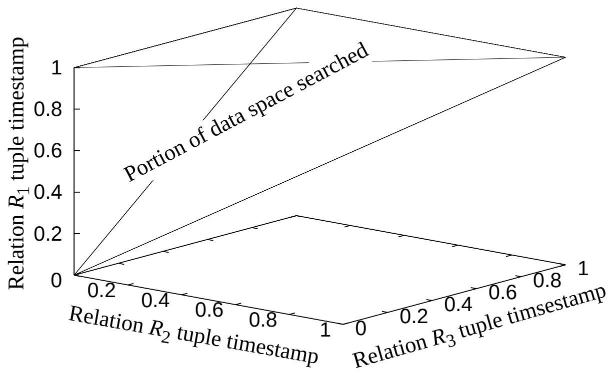

Figure 4: Searching for "lucky" output tuples in Turbo DBO. The fraction of the data space that is searched is much greater than in Figure 3.

a tuple $t = (t_1 \bullet t_2 \bullet ... \bullet t_i \bullet t_n)$ is discovered by the IMJ if and only if $TS(t_1) < TS(t_2) < ... < TS(t_n)$. The reason this technique may be very successful at discovering "lucky" output tuples is that the IMJ does not need to discover them all-at-once. It can build them up slowly, over time, which greatly increases the chance that they will be found.

It is possible to visualize Turbo DBO's search strategy using a plot that is analogous to Figures 2 and 3. In Figure 4, the input tuples from each of the three relations are ordered along each axis based upon their arrival timestamps; just as in the previous figures, the goal is to locate points in the space corresponding to combinations of input tuples accepted by the predicate P. Since an output tuple is discovered by the IMJ if $TS(t_1) < TS(t_2) < ... < TS(t_n)$, the IMJ will eventually discover any tuple falling above and to the left of a triangular plane that connects the point $(0, 0, ..., 0)$ with the point $(1, 1, ..., 1)$. Over time, the volume of this space can dwarf the volume of the space that would be searched along the diagonal by classic DBO. Since the variance of the resulting estimate decreases in proportion to the volume of the space searched, the strategy used by Turbo DBO can greatly increase estimation accuracy.

Of course, this discussion is fairly high level, and ignores many of the key details that must be considered when realizing a practical implementation of these ideas. The following key questions are considered in the remainder of the paper:

- How exactly can Turbo DBO use this search strategy to produce an unbiased estimate for Q, and how can the accuracy of the resulting estimator be quantified in a rigorous fashion?
- How is the search strategy actually implemented by the IMJ, and what data structures are required?
- Chains of tuples of the form $(t_1 \bullet t_2 \bullet ... \bullet t_i \bullet t_n)$ are constructed using a specific order for the input relations. Is this order important, and if so, how can it be chosen?
- What happens when the space required to store all of the chains exceeds the IMJ's memory budget?
- How is the timestamp-based randomization described in this section implemented?

6. BUILDING AN UNBIASED ESTIMATOR

The previous section described a process which computes a random variable Σ. We have argued that this variable can be used to produce a low-variance estimator because it searches a much larger portion of the data space than the estimator used in classic DBO. In this section, we derive an unbiased estimator N_i based upon Σ.

To use Σ to produce an unbiased estimator N_i (that is, one that is correct on expectation), it is necessary to compute the expectation of Σ. To do this, we begin by writing the exact formula for Σ. In the remainder of this section, we simplify the formulation by assuming that the aggregate function f has been altered to incorporate P; that is, if $P(t) = \texttt{false}$, then $f(t) = 0$. Assume that at a given instant in time, the IMJ has processed all tuples with a timestamp less than p. Given this, Σ can be expressed as:

$$\Sigma = \sum_{t_1 \in R_1} \sum_{t_2 \in R_2} ... \sum_{t_n \in R_n} I(TS(t_1) < TS(t_2) < \quad ... < TS(t_n) \\ \wedge TS(t_i) \leq p \text{ forall } i) f(t_1 \bullet t_2 \bullet ... \bullet t_n) \tag{1}$$

In this formula, I is the identity function, returning one if the random variable-valued argument returns `true` and zero otherwise. In this case, I returns one if and only if tuple t_1 is encountered by the IMJ before t_2, which is encountered before t_3, and so on, and all have a timestamp less than p. To use Σ to produce an unbiased estimator N_i, it is necessary to compute the expectation of Σ:

$$E[\Sigma] = \sum_{t_1 \in R_1} ... \sum_{t_n \in R_n} E[I(TS(t_1) < TS(t_2) < ... < TS(t_n) \\ \wedge TS(t_i) \leq p \text{ forall } i)] f(t_1 \bullet t_2 \bullet ... \bullet t_n)$$

In this equation, the $E[.]$ operator can be pushed inside the summation due to the linearity of expectation. Thus, it becomes necessary to consider the expected value of $E[I(.)]$. Since this is a zero-one random variable, its expectation is the probability that it evaluates to one. In other words, its expectation is the probability that the tuples t_1 through t_n arrive in precisely that order, before time p. Since each $TS(t_i)$ is an independent, uniformly distributed variable, any ordering is equally possible. There are $n!$ orderings of n different tuples, and so $Pr[TS(t_1) < TS(t_2) < ... < TS(t_n)] = \frac{1}{n!}$. Furthermore, due to the uniformity of $TS(t_i)$, the fact that all $TS(t_i) < p$ has no effect on this probability—if we know that all t_i arrived sometimes before p, then the conditional distribution is still uniform from 0 to p, which does not affect the fact that each ordering is equally likely. Thus, we have $E[I(.)] = \frac{p^n}{n!}$ and:

$$E[\Sigma] = \sum_{t_1 \in R_1} ... \sum_{t_n \in R_n} \frac{p^n}{n!} f(t_1 \bullet t_2 \bullet ... \bullet t_n)$$

In order to make Σ unbiased (that is, equal to Q on expectation), all we have to do is multiply Σ by $\frac{n!}{p^n}$. Thus, the unbiased estimator associated with the i^{th} levelwise step is $N_i = \frac{n!}{p^n}\Sigma$.

7. IMJ IMPLEMENTATION

The IMJ accepts a stream of input tuples from DBO that are ordered based upon each tuple's timestamp value and uses them to compute Σ. In practice, this means that the IMJ must maintain data structures that efficiently allow the IMJ to accept a tuple t_i from R_i, and join it with any existing chain of the form $(t_1 \bullet ... \bullet t_{i-1})$ to create new tuples of the form $t = (t_1 \bullet ... \bullet t_{i-1} \bullet t_i)$ where $P(t) = \texttt{true}$. Our IMJ implementation described in this section is related to the MJoin algorithm for joining data streams with highly variable and unpredictable rates introduced in [16].

As an IMJ is started up but before it begins accepting tuples, Turbo DBO's query compiler supplies the IMJ with an $n \times n$ matrix

P containing boolean predicates of the form:

$$\begin{pmatrix} \texttt{true} & \texttt{true} & \texttt{true} & ... & \texttt{true} \\ P_{21} & \texttt{true} & \texttt{true} & ... & \texttt{true} \\ P_{31} & P_{32} & \texttt{true} & ... & \texttt{true} \\ P_{41} & P_{42} & P_{43} & ... & \texttt{true} \\ ... & ... & ... & ... & ... \end{pmatrix}$$

An entry P_{ij} in **P** corresponds to the user-supplied equi-join predicate for relations i and j in the original query predicate P, which is suitable for use within a hash join. The query compiler also provides the IMJ with an additional boolean predicate $\bar{P}$. $\bar{P}$ is everything that is "left over" from P that does not appear in **P**; that is, it is everything in P that cannot be captured as an equi-join predicate suitable for hash-based evaluation. For example, if P contains a clause of the form "`e.SAL > s.SAL + 500`", then the clause would appear in $\bar{P}$ since it is not possible to compute this join using standard hashing techniques. Given this, it is always the case that predicate $P = \bar{P} \wedge \bigwedge_{ij} P_{ij}$ [2].

As the IMJ starts up, it creates n different sets, which are initially empty. The i^{th} set will be used to hold the tuples or chains of the form $t = (t_1 \bullet t_2 \bullet ... \bullet t_i)$ from the cross product $R_1 \times R_2 \times ... \times R_i$ where $P(t) = \texttt{true}$. In order to efficiently search for such tuples, the IMJ also creates a hash index on each set. The hash index on the $(i-1)^{th}$ set needs to be able to quickly locate all of the tuples from this set that join with any tuple $t_i \in R_i$ that is passed to the IMJ. More specifically, given a t_i the IMJ needs to quickly locate every chain $(t_1 \bullet t_2 \bullet ... \bullet t_{i-1})$ in the $(i-1)^{th}$ set where the tuple $(t_1 \bullet ... \bullet t_{i-1} \bullet t_i)$ is accepted by all of the predicates in the i^{th} row of **P**. Thus, all of the tuples in the $(i-1)^{th}$ set are indexed by a hash that takes into account the attributes from $R_1, R_2, ..., R_{i-1}$ that appear in the predicates $P_{i1}, P_{i2}, ..., P_{i(i-1)}$.

When a new tuple t_i is processed by the IMJ, it is first hashed on all of the values of attributes from R_i that appear in the predicates $P_{i1}, P_{i2}, ..., P_{i(i-1)}$, and then joined with any chain $(t_1 \bullet t_2 \bullet ... \bullet t_{i-1})$ in the corresponding bucket by applying all of the predicates in the i^{th} row of **P**. If $i = n$ (that is, t_i comes from relation R_n), then the predicate $\bar{P}$ is also checked; if $\bar{P}$ accepts any new chain, then it is treated as a "lucky" output tuple and the aggregate function f is applied to the chain and the result is added to Σ. Finally, the "lucky" output tuple is added to the n^{th} set.

If $i \neq n$, then any new chains resulting from t_i are added directly to the i^{th} set—Σ is not updated, and $\bar{P}$ is not applied. Also in this case, the index on the i^{th} set must be updated to take into account these new chains, so that if a tuple t_{i+1} from R_{i+1} is eventually processed, any matching chains from the i^{th} set can be located quickly using the hash index.

8. ORDERING THE INPUT RELATIONS

The IMJ accepts a stream of input tuples from each of the level-wise step's n input relations, and must map them to $R_1, R_2, ..R_n$, which induces an ordering on the relations. We consider how does this mapping/ordering matter, and how should it be chosen?

First, we argue that altering the ordering of the input relations *has no effect upon the statistical properties of the estimation process*. Consider the formula for the random variable Σ given in Equation 1. Altering the ordering of input relations has only the effect of reordering the summations and of altering identities of the tuples in the expression $TS(t_1) < TS(t_2) < ... < TS(t_n)$ and so it does not affect the value of Σ. Thus, there is no statistical effect.

[2]In this discussion, we ignore any "pure" selection predicates; it is assumed that any tuples not accepted by such a selection predicate would be filtered before they even reach the IMJ, and so such selection predicates are immaterial to this discussion.

While there is no *statistical* effect of the ordering, there is a key practical effect: different orderings require different amounts of memory. The problem of choosing an appropriate ordering is somehow similar to logical query plan optimization (QO), but this optimization problem has a unique structure. If the number of tuples in $R_1 \times R_2 \times ... \times R_i$ that are accepted by the predicate P is m, then by the time the IMJ completes, the number of tuples from this cross product that will be buffered by the IMJ is expected to be $\frac{m}{i!}$. This implies that choosing the identities of the input relations early in the ordering is far more important than choosing the identity of those later in the ordering, and suggests that a greedy strategy is appropriate.

As such, our prototype chooses the first pair of relations in the ordering in an "optimal" fashion, and then orders the remainder greedily. To implement this, the IMJ begins with a short start-up phase lasting a few seconds, when it buffers all of the tuples that it sees. If the largest TS value encountered during this start-up phase is p, then when the start-up phase ends the IMJ has buffered (approximately) a $(p \times 100)\%$ sample of each input relation.

Next, the IMJ chooses R_1 and R_2 by considering all pairs of input relations that have a join predicate in the matrix **P**. For each pair of relations R_a and R_b, the IMJ uses its start-up samples to estimate $bytes_a =$ (size in bytes of R_a), $bytes_b =$ (size in bytes of R_b), and $bytes_{ab} =$ (size in bytes of $R_a \bowtie R_b$). by joining the two samples, and multiplying the size of the join result by $\frac{1}{p^2}$. Once the IMJ has estimated these quantities for each (R_a, R_b) combination, it then chooses the (R_a, R_b) combination that minimizes the quantity $min(bytes_a + bytes_b + bytes_{ab}/2)$.

Then the IMJ chooses R_3 by (a) joining the start-up tuples from each of the remaining relations with the start-up tuples from $R_1 \bowtie R_2$, and (b) selecting R_3 as the relation for which the size (in bytes) of the join result is minimized. Then, it chooses R_4 in a similar fashion. This process is repeated until the relation R_n is selected.

9. REDUCING THE FOOTPRINT SIZE OF THE IMJ

9.1 Subsampling the Relations

In Turbo DBO, the IMJ is given two explicit main memory budgets: a hard upper bound and a soft upper bound on the IMJ's memory footprint. As soon as the amount of storage required by the IMJ exceeds the hard upper bound, one of the relations is chosen to "give up" a fraction of its tuples so that the footprint of the IMJ shrinks below the soft upper bound. How to choose which relation has to "give up" some tuples is discussed subsequently, but once a relation R_i has been chosen in response to a memory overflow, the "giving up" of tuples is implemented by subsampling the relation in a Bernoulli fashion: logically, for each $t_i \in R_i$ that is present in the IMJ, a biased coin is flipped. If the coin comes up "heads", then every chain of the form $(t_1 \bullet ... \bullet t_i \bullet ...)$ is removed from the IMJ and the memory is freed.

The effect of this process is that every relation R_i now has a subsampling rate p_i. That is, there is a probability $(1 - p_i)$ that a given tuple from relation R_i has been removed from the IMJ. If a relation has never given up any tuples, then $p_i = 1$. If a relation has given up some of its tuples, then $p_i < 1$.

The effect of subsampling on the search algorithm implemented by Turbo DBO is illustrated pictorially in Figure 5. In this figure, a random subset of the tuples from one of the input relations is removed, which results in the portion of the data space that is searched being "cut up", where the empty horizontal areas are associated with tuples that have been removed due to the subsam-

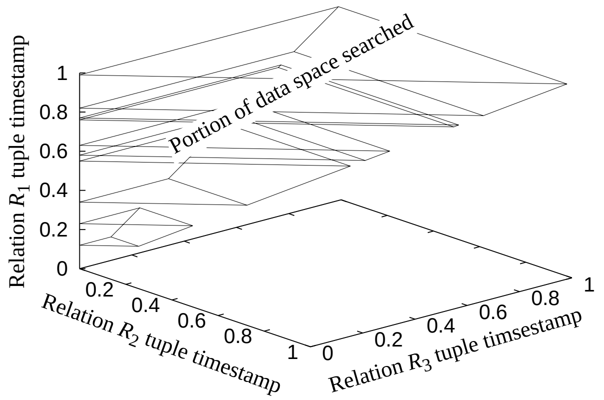

Figure 5: The effect of subsampling on the search for "lucky" output tuples in Turbo DBO.

pling. If more than one relation were subsampled, then there would be cuts or holes along another axis as well. Since the portion of the data space searched is (approximately) proportional to the estimation accuracy, it should be clear that subsampling will have a negative effect on the estimation accuracy—minimizing this negative effect is discussed subsequently.

9.2 Subsampling Implementation

To actually implement the subsampling, the IMJ attaches a value randomly selected from the range zero to one to tuples as they are added to the IMJ. As a tuple $t_i \in R_i$ is added to the IMJ, it is checked to see if its random value exceeds p_i. If it does, then the tuple is discarded without ever entering the IMJ. Also, whenever the subsampling rate p_i for R_i is lowered, the chains that are already stored in the IMJ may need to be deleted as well. For every existing chain of the form $(t_1 \bullet ... \bullet t_i \bullet ...)$, after p_i is lowered, the random value associated with t_i is also checked against p_i; if the random value exceeds p_i, then the chain is deleted. If any chain of the form $(t_1 \bullet ... \bullet t_n)$ is deleted, then its $f()$ value is also subtracted from the sum Σ.

9.3 Statistical Considerations

One effect of this is that the quantity Σ computed by the IMJ actually changes. Mathematically, a random variable X_{t_i} is attached to each tuple $t_i \in R_i$. X_{t_i} takes the value one with probability p_i; otherwise, it takes the value zero. Then:

$$\Sigma = \sum_{t_1 \in R_1} \sum_{t_2 \in R_2} ... \sum_{t_n \in R_n} I(TS(t_1) < TS(t_2) < \quad ... < TS(t_n)$$
$$\wedge\, TS(t_i) \leq p \text{ forall } i) f(t_1 \bullet t_2 \bullet ... \bullet t_n) \prod_i X_{t_i}$$

For this new version of Σ, it becomes necessary to compute the expectation again in order to produce an unbiased N_i. Since $E[\prod_i X_{t_i}]$ is $\prod_i p_i$, we have:

$$E[\Sigma] = \sum_{t_1 \in R_1} ... \sum_{t_n \in R_n} \frac{p^n}{n!} f(t_1 \bullet t_2 \bullet ... \bullet t_n) \prod_i p_i$$

Thus, in order to produce an unbiased estimate for Q, we let $N_i = \frac{n!}{p^n \prod_i p_i} \Sigma$.

9.4 Choosing the Relation to Subsample

When the IMJ exceeds its hard memory budget, it needs to choose which relation to subsample from. This is done as follows:

1. The IMJ considers each input relation, in turn. For each relation R_i, the IMJ estimates the new p_i value (denoted by p'_i) that would be required to shrink the footprint size so that it does not exceed the soft memory budget. This is done by maintaining a set of counters as new data are inserted into the IMJ, one for each relation. For relation R_i, the IMJ maintains a counter B_i, which is the total number of bytes required to store chains containing any tuple $t_i \in R_i$. Assuming that B does not exceed B_i, then if the IMJ needs to shrink its footprint size by B bytes, p'_i can be estimated as $\frac{Bp_i}{B_i}$.

2. For each p'_i, the IMJ computes the variance of the IMJ's estimate that would be obtained if p_i were replaced by p'_i.

3. The relation with the minimum resulting variance is selected. p_i for this relation is replaced with p'_i. All now-defunct chains are removed from the IMJ, and Σ is updated.

10. STREAMING TUPLES INTO THE IMJ

The final systems-oriented issue that we consider is how Turbo DBO actually supplies tuples to the IMJ. Specifically, Turbo DBO needs to stream tuples into the IMJ so that they are sorted upon the order of their randomized timestamp values.

10.1 Simulating Timestamp Ordering

In Turbo DBO (as in regular DBO), all joins are implemented using sort or hash algorithms, where the lexicographic order for the sorting or hashing is based upon a random ordering provided by employing a hash function over the join key. This means that tuples stream out of each join in a statistically random order, except for the fact that tuples with the same join key value appear all at once in a "clump". These clumps do not affect query processing or un-biasedness of Turbo DBO's estimates in any way, except for the fact that clumps tend to increase the variance of the resulting estimate. For the remainder of the discussion, we ignore this clumping of tuples and point out that the statistical issues introduced by the clumping can be handled using methods very similar to those proposed for handling clumping in the original DBO paper [9].

Ignoring clumping, it can be assumed that tuples are streamed out of each levelwise step's join operations in random order. The random order supplied by each individual join needs to be used to stream tuples into the IMJ in a way that is statistically equivalent to first sorting all of the tuples from each and every join on the value of a random timestamp, and then streaming them into the IMJ in the resulting sorted order.

In Turbo DBO, this simulated sorting is facilitated by a special software component called the *Controller*. The Controller controls a set of "valves", where there is one valve placed on each of the output pipes through which all intermediate join results flow as they are pipelined into the operations higher in the query plan. Initially, the Controller turns all of the valves to the "off" position, which blocks any tuples from flowing through the pipes. As the IMJ is ready to start processing a levelwise step, the Controller turns on each of the valves for a period of time. Then, as tuples flow into the operations higher in the query plan, copies of the tuples are also redirected into an in-memory priority queue maintained by the Controller. The required ordering of all of the tuples in the queue is obtained by normalizing each tuple's random hash value to a $[0, 1]$ range—these normalized values are used to supply the random timestamp that will be used to insert tuples into the IMJ.

Once the priority queue has filled, the Controller begins popping tuples off of the front of the queue, and feeding them into the IMJ. Whenever the number of tuples from one of the output pipes that is buffered in the queue becomes too small, the Controller turns on the valve corresponding to that pipe for long enough to replenish the supply of tuples from that pipe.

10.2 Handling the Initial Table Scans

To achieve randomness in the initial table scans, data are stored in random order on disk. To generate a timestamp for disk-based tuples, the table scan maintains a value $t_i.num$ for the i^{th} relation. $t_i.num = 0$ for the first tuple from relation i, 1 for the second, 2 for the third, and so on. The Controller also remembers the value of the last timestamp provided by each relation (call this value TS_i for the i^{th} relation). This number is set to be 0 initially. When the Controller pulls a new tuple t_i from the i^{th} pipe to put into its priority queue, it generates a random number X from a Beta$(1, |R_i| - t_i.num)$ distribution, and then sets the timestamp of t_i to be $TS_i + X(1 - TS_i)$. Without going into details, this process assigns a simulated timestamp and ordering to t_i that is statistically equivalent to the required timestamp and ordering.

10.3 Constant Relations

It is often the case that a table scan or intermediate join result supplies a set of tuples into a levelwise step that is so small that it can be buffered in its entirety. This is labeled as a "constant relation". Each constant relation is buffered by the IMJ in its entirety.

The effect of this is that the value Σ computed by the IMJ is altered so that the timestamps for each constant relation are irrelevant—a tuple t_i from a constant relation is *always* counted towards Σ, no matter what its timestamp. Mathematically, we can represent this by introducing a boolean variable C_i that is true if and only if R_i is constant. Then:

$$\Sigma = \sum_{t_1 \in R_1} \sum_{t_2 \in R_2} \dots \sum_{t_n \in R_n} I(\bigwedge_{i \neq j} (TS(t_i) < TS(t_j) \vee C_i \vee C_j) \wedge TS(t_i) \leq p \text{ forall } i \text{ where } \neg C_i) f(t_1 \bullet t_2 \bullet \dots \bullet t_n) \prod_i X_{t_i}$$

Since Σ is altered, it must be unbiased to compute the IMJ's estimate N_i in a slightly different fashion than if there are no constant relations. Let n' be the number of non-constant relations. Then $N_i = \frac{(n')!}{p^{n'} \prod_i p_i} \Sigma$ is an unbiased estimate for Q.

11. VARIANCE ANALYSIS

As we explained in Section 5, the IMJ estimator is based on a significantly larger part of the tuple space when compared to the DBO estimator, thus, we expect the IMJ estimator to have a significantly smaller variance. In order to provide meaningful confidence bounds for the IMJ estimator, the variance needs to be estimated accurately enough. The first step is to derive formulas for the variance and then to design an unbiased estimator for the variance.

The starting point for the derivation of the variance of Σ is the analysis developed for the DBO estimator [9]. Surprisingly, the analysis of the DBO estimator depends only on a small degree on the type of sampling; a general analysis was developed in [10] for *any* sampling estimator as long as independent uniform samples, one for each relation, are used to identify matching tuples and form the query result estimator. The main feature of the IMJ estimator is that it does not combine *independent* samples from the relations, but rather computes the estimate using a complicated randomized process. It is not immediately apparent that the analysis in [10] does apply to the IMJ estimator. Fortunately, we discovered that the existing analysis can be generalized to the IMJ estimator. In the rest of this section we first develop this more general analysis and then apply it to the IMJ estimator.

11.1 Generalized Uniform Sampling

The IMJ estimator does not look at all like a sampling estimator. Nevertheless, the IMJ estimator randomly selects tuples from the result tuples of the query and uses them to estimate the result. The selection process seems non-uniform and too complicated to analyze. Fortunately, as we show in the next section, the tuple selection used by the IMJ estimator does belong to the class of methods we call *generalized uniform sampling (GUS).*

DEFINITION 1 (GUS). *A randomized selection process that selects tuples $t = (t_1, \dots, t_n)$ from the cross-product of base relations $R_1, \dots, R_n$ is called* generalized uniform sampling *or GUS if the probability that tuple t is selected is constant and the probability that tuples t and t′ are simultaneously selected depends only on whether tuples t and t′ share the same tuples from the base relations, but not on the content of these tuples.*

For any GUS process, we can define the following constants:

$$P[(t_1, \dots, t_n) \in \mathcal{R}] = a$$
$$P[(t_1, \dots, t_n) \in \mathcal{R} \wedge (t'_1, \dots, t'_n) \in \mathcal{R}] = b_T, T = \{i | t_i = t'_i\}$$

with $\mathcal{R}$ the set of result tuples randomly selected by the specific process. The set T used as a subscript in the constant b_T specifies which of the tuples form the base relations used in the two result tuples are the same. Once this information is provided, the probability is a constant according to the definition of GUS above.

If we denote by $\mathcal{A}$ the true result of the query, the following estimator of $\mathcal{A}$ can be introduced:

$$X = \frac{1}{a} \sum_{(t_1, \dots, t_n) \in \mathcal{R}} f(t_1 \bullet \dots \bullet t_n)$$

The moment analysis of this generic estimator is given by the following result that we provide without proof:

THEOREM 1. *The expected value and the variance of the GUS estimator X are given by:*

$$E[X] = \mathcal{A}, \qquad \sigma^2(X) = \sum_{S \in \mathcal{P}(n)} \frac{c_S}{a^2} y_S - y_\emptyset$$

with

$$y_S = \sum_{\{t_i \in R_i | i \in S\}} \left(\sum_{\{t_j \in R_j | j \in S^C\}} f(\{t_i, t_j\}) \right)^2$$
$$c_S = \sum_{T \in \mathcal{P}(S)} (-1)^{|T| + |S|} b_T$$

The terms y_S are exactly the terms that appear in [10] and can be evaluated using the same strategy.

11.2 Analysis of IMJ estimators

Since all the IMJ estimators do not look at the content of a tuple to decide if the tuple is retained in the sample used for estimation, all the estimators can be characterized using the analysis of the GUS generic estimator. We only need to compute the constants a and $b_T, \forall T \in \mathcal{P}(n)$ for each different type of sampling. While such a computation is nontrivial, it is significantly easier than a *head on* analysis. We provide here only the values of the constants b_T. The values of a ware derived in the previous sections when the estimators were introduced.

Basic IMJ estimator.

$$b_T = \frac{p^{2n-|T|}}{(2n-|T|)!}\prod_{l=0}^{|T|}\frac{[2(i_{l+1}-i_l-1)]!}{[(i_{l+1}-i_l-1)!]^2} \quad (2)$$

where $i_1, i_2, \ldots i_{|T|}$ are the members of the set T, in increasing order (i.e. $i_1 < i_2 < \cdots < i_{|T|}$) and $i_0 = 0$, $i_{|T|+1} = n + 1$.

Subsampling. Subsampling is performed independently of the selection of tuples retained by the basic IMJ estimator. It can be shown that the constants b_T for two independent sampling processes used in sequence are the product of the constants corresponding to the individual sampling. With this

$$b_T = \prod_{i\in T} p_i \prod_{i\in T^C} p_i^2 b'_T$$

where b'_T are the constants for the basic IMJ estimator and p_i is the probability that tuple $t_i \in R_i$ is subsampled.

Constant relations. If we denote by C the set of constant relations, to compute b_T, we form the set $T' = T - C$ and return $b_{T'}$ as computed for the basic IMJ estimator described in Equation 2.

12. EXPERIMENTS

There are four specific goals of our benchmarking:

1. First, we wish to explore how much of an improvement the techniques described in this paper might potentially bring compared to the original version of DBO in the extreme cases where DBO's "time 'til utility" is questionable.
2. Second, we wish to see the degree of improvement these new techniques can achieve in a standard, multi-table query with aggregation and grouping.
3. Third, we wish to verify experimentally the unbiasedness of our estimates and the correctness of the statistical bounds provided by our methods.
4. Finally, we would like to have some idea of the extra expense associated with the estimation algorithms used by Turbo DBO, compared to a traditional database system.

12.1 Basic Setup

The version of Turbo DBO that we benchmark is implemented as approximately 40,000 lines of C++ code. Since our goal is to compare Turbo DBO against the original DBO, we require an implementation of the original DBO as well. The implementation of the original DBO that we benchmark is nothing more than a modification of our Turbo DBO implementation, with the various software components modified so that they implement original DBO's estimation algorithms, rather than Turbo DBO's. Because it is not a ground-up implementation, our "hacked" version of original DBO is quite slow. Thus, for all of our comparisons, the plots are stretched or contracted (normalized) as needed to ensure that both DBO versions begin and end each levelwise step at exactly the same instant on the timeline. This ensures that the comparisons are without an implementation bias.

For each experiment, we are mostly interested in confidence interval width as a function of time. Thus, the vast majority of our plots will have time as the x axis, and the confidence interval width (or relative error) as the y axis, where the width is computed as $\frac{high-low}{est}$, where est is the current estimate, and low and $high$ are 95% confidence bounds on the answer. Thus, if the interval width is 0.3 (for example), then the error is $\pm 15\%$. We note that while the time axis always starts at zero, differences between Turbo and original DBO before 5% of the query has completed are mostly meaningless when comparing the two approaches, because such differences can be attributed largely to system-dependent start-up costs, and not to fundamental differences.

All experiments were conducted with a 20GB instance of the TPC-H benchmark database. Experiments were run on a low-end, eight core Intel server running the Ubuntu distribution of the Linux OS, with the database data striped across four disks. Both versions of DBO make use of 2GB of RAM.

12.2 Nasty Joins, Selective Predicates

Our first set of experiments is designed to compare the two DBO versions in the case where Turbo DBO is likely (by design) to be the most advantageous: when many large database tables are joined, or when a selection predicate of high selectivity is applied to one of the inputs to a join.

Setup. In our first experiment, we run a query of the form:

```
SELECT SUM(l_extendedprice)
FROM lineitem, orders_1, orders_2,...,orders_N
WHERE l_orderkey = o1_orderkey AND
  o1_orderkey = o2_orderkey...
```

In this query, `orders_1`, `orders_2`, and `orders_N` are replicated versions of the TPC-H `orders` relation. Since both `orders` and `lineitem` are large, this query tests the ability of the system to produce narrow bound widths over query plans that join several large relations that cannot fit into memory. The query is run for an `N` value of two, three, and four. Results are plotted in Figure 6.

In our second experiment, we run a query of the form:

```
SELECT SUM(l.l_extendedprice)
FROM lineitem l, orders
WHERE l.l_orderkey = o_orderkey AND PRED(l)
```

In this query, `PRED(l)` is some selection predicate on tuples from `lineitem`. `PRED` is varied so that it accepts 10%, 1%, or 0.1% of the tuples from `lineitem`. The results are plotted in Figure 7.

Discussion. These plots clearly show that as the query gets "nastier", Turbo DBO performs better and better compared to the original version of DBO. Consider Figure 6. For this particular three-table join, both systems give extremely accurate estimates after just 10% of the time required to process the entire query: the relative error is 2.5%, or ± 0.0125. For a four-table join, there begins to be a clear separation between Turbo DBO and original DBO, for the period when between 10% and 60% of the query has been processed. However, one can argue that this difference may not be too significant, because for both systems the relative error is less than 10% after 10% of the query has been processed; this is already quite accurate. However, when another table is added in, original DBO really begins to suffer. After 10% of the query has been processed, Turbo DBO is able to give useful bounds, with error of ± 0.15. However, original DBO does not begin to produce bounds of equivalent quality until nearly 45% of the query has been processed; in this way, the "time 'til utility" (TTU) of the Turbo DBO estimate is 77% smaller that the TTU for original DBO. With additional tables in the join, the gap between the two TTUs will only increase.

A similar trend is observed in Figure 7. For a two-table join, when 10% of the tuples from the central fact table are accepted by the underlying selection predicate, both systems give extremely tight bounds after only 10% of the query has been processed. For a 1% selection predicate, there is a gap between the two systems, but

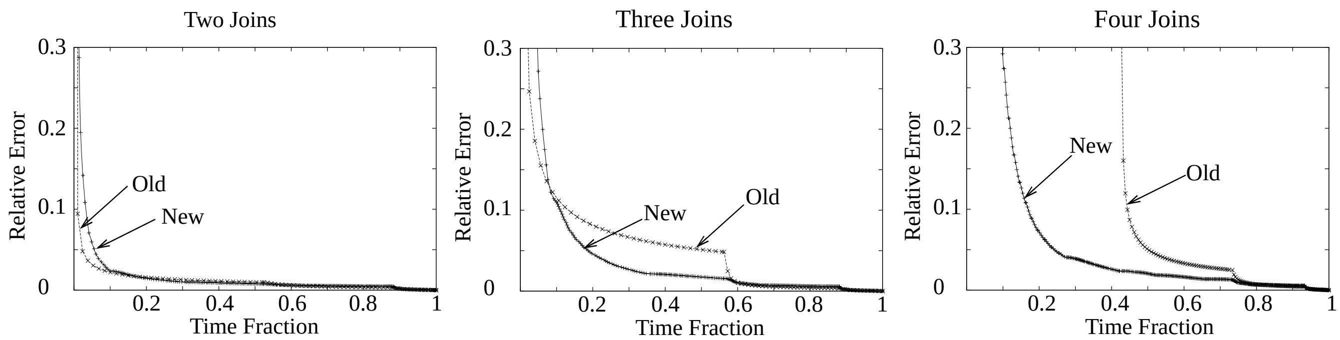

Figure 6: Confidence bound width as a function of time for joins of multiple large relations.

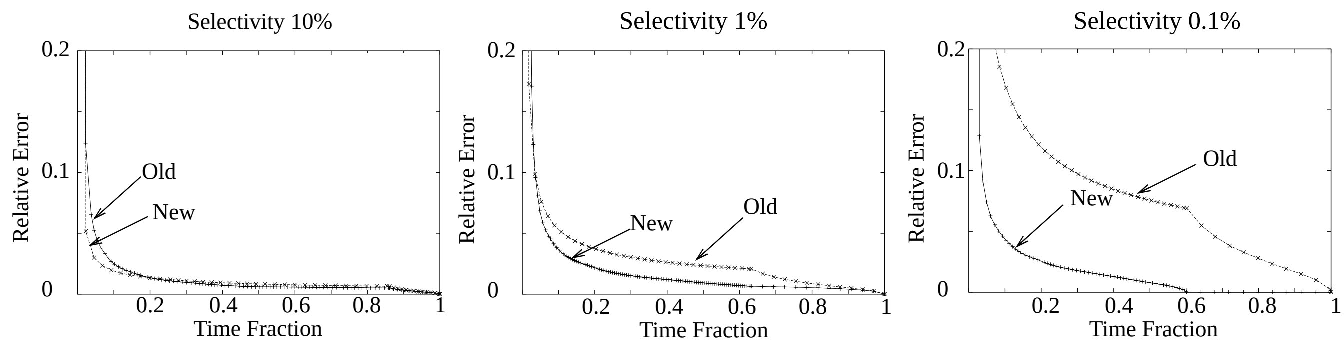

Figure 7: Confidence bound width as a function of time for join queries with relational selection predicates on the central fact table.

it is probably not significant; after all, both systems have around 5% relative error after only 10% of the query has been processed. But when 0.1% of the tuples are accepted, the gap becomes very large. In fact, Turbo DBO has a zero-variance estimate after query processing is only one-half complete, because it treats `lineitem` as a constant relation in this case.

12.3 TPC-H Queries

It is clear that one can construct queries for which Turbo DBO outperforms the original DBO. However, how will the two systems compare on run-of-the mill, TPC-H-style queries?

Setup. To address this, we run the following four queries, each chosen for their "ordinary"-ness as standard, analytic-style queries. The four queries are named Q1, Q2, Q3, and Q4 respectively. SQL code for these four queries follows:

```
SELECT n_name, sum(l_extprice*(1-l_discount))
FROM customer, orders, lineitem, nation
WHERE (fk join conds) AND l_retflag = 'R'
AND o_orderdate < '1994-01-01'
AND o_orderdate > '1993-09-30'
GROUP BY n_name
```

```
SELECT n_name, sum(l_extprice*(1-l_discount))
FROM customer, orders, lineitem,
  supplier, nation, region
WHERE (fk join conds) AND l_disc > 0.08
AND r_name = 'ASIA' AND o_orderdate >
'1993-12-31' AND o_orderdate < '1995-01-01'
GROUP BY n_name
```

```
SELECT n1_name, n2_name, extract(year from
  l_shipdate) as l_yr,
  sum(l_extprice*(1-l_discount))
FROM  customer, orders, lineitem,
  supplier, nation n1, nation n2
WHERE (fk join conds) AND ((n1_name = 'GERM'
AND n2_name = 'FRANCE') OR (n2_name = 'GERM'
AND n1_name =  'FRANCE')) AND l_shipdate <
'1997-01-01' AND l_shipdate > '1995-01-01'
AND l_discount > 0.08
GROUP BY n1_name, n2_name, l_yr
```

```
SELECT sum(1), l_shipmode, extract(year from
  l_shipdate) as l_yr
FROM orders, lineitem
WHERE (fk join cond) AND o_orderpriority >
'1-URGENT' AND l_recdate > '1993-12-31'
AND l_recdate  < '1995-01-01' AND l_comdate
< l_recdate AND l_shipdate < l_comdate
GROUP BY l_shipmode, l_yr
```

We executed each of these queries over the TPC-H database. Results are plotted in Figure 8. For each query, one of the groups returned is arbitrarily chosen, and the confidence interval width as a function of time for that group is plotted.

Discussion. Each of these queries considers at most two large and a few medium-sized database tables, since only `orders` and `lineitem` are too large to fit into the amount of available main memory. Thus, these queries are hardly the sort of workload that Turbo DBO was designed to outperform original DBO on. Still, there is a significant difference between Turbo and original DBO for each of the four queries. While the two curves may look similar in each plot, the key comparison to make is the TTU for both systems on each query — that is, the time required until a usable estimate has been returned. If one considers a confidence bound width of ± 0.15 to be the smallest usable width, then for Q1, Turbo DBO decreases the TTU from 0.3 to 0.26, for a reduction of 13%. For Q2, the decrease is from 0.56 to 0.27, or 41%. For Q3, the

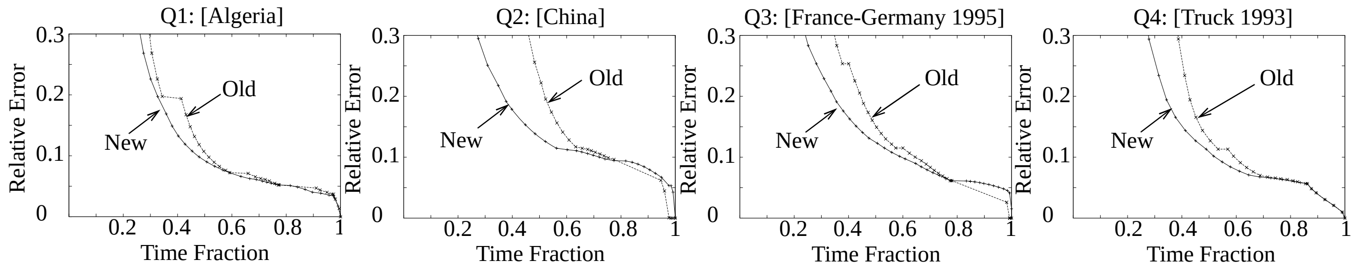

Figure 8: Confidence bound width as a function of time for typical, analytic queries.

decrease is from 0.35 to 0.25, or 29%. For Q4, the decrease is from 0.39 to 0.27, or 31%.

The take-home message from all of this is that it is certainly possible to construct queries for which Turbo DBO decreases the TTU by 70% or more — just join many large tables together. But even for a rather pedestrian, "join the fact table with a bunch of dimension tables" workload, Turbo DBO averaged a 29% reduction in TTU, which we argue is very significant.

12.4 Data Skew

Data skew is always a problem when sampling. If one joins `lineitem` and `orders` but a certain subset of the orders have many more entries in `lineitem` than the others, then sampling can be quite inaccurate if it misses those important orders.

Setup. To test whether Turbo DBO provides some protection against data skew compared to original DBO, we re-run Q1 above. However, this time we modify the TPC-H data generator so that the number of entries in `lineitem` that reference a single record in `orders` is not uniform, but is instead generated via a Zipf distribution. We then generate four versions of `lineitem`, using Zipf parameters 0, 0.5, 1.0, and 2.0. AQ Zipf parameter of 0 should give results that are identical to Q1 in the previous subsection (no skew), whereas a parameter of 2.0 is indicative of extreme skew. "Real life" domains with extreme skew (such as the frequency of use of words in the English language dictionary) tend to top out with a Zipf parameter of around 2. This is very severe skew—in English, the top 20 words (out of more than 100,000) account for nearly a third of the English words in print. For these four data sets, we plot confidence bound width as a function of time in Figure 9.

Discussion. The results are striking. Turbo DBO is relatively unaffected by skew until the Zipf parameter rises to 2.0, whereas original DBO is unable to provide meaningful estimates until the query is more than 90% complete, even with the moderate skew resulting from a Zipf parameter of 0.5. The reason for Turbo DBO's robustness to skew is that when Turbo DBO runs out of memory, it is able to intelligently choose which relation to subsample so as to minimize the variance. Since, in this case, certain records from `orders` are very important, Turbo DBO will avoid subsampling `orders` to save space, and will subsample `lineitem` or `customer` instead.

12.5 Correctness

It is useful to experimentally verify the correctness of Turbo DBO's statistical guarantees via a Monte Carlo experiment.

Setup. For each of the four queries from the previous subsection, we repeated the following experiment 100 times. For each experiment, we first randomly shuffle all of the data on disk, so that there is no correlation across experiments. Then, we ran the query from start to finish. For each levelwise step (there are three levelwise steps in each query except for Q1, which has only two), at ten

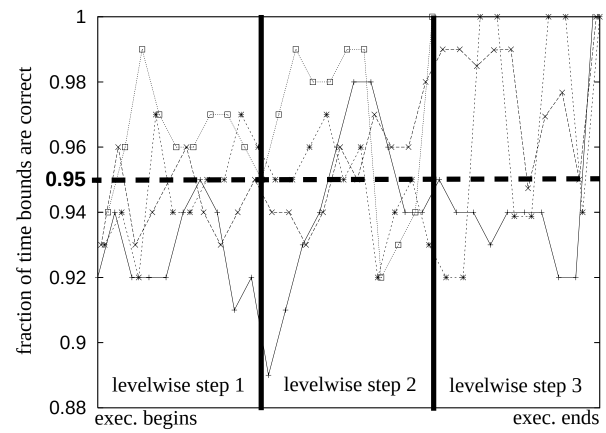

Figure 10: Accuracy of Turbo DBO's confidence bounds. Each line corresponds to one of the four test queries. At ten different intervals during each levelwise step, the fraction of the time that Turbo DBO's 95% intervals contained the actual query result is reported. Ideally, each value should be close to 0.95.

evenly-spaced intervals, we checked the accuracy of the current, 95% confidence bound. If the current confidence bound contained the true query result, we report the trial as a success; otherwise it is failure. Thus, for a three-level query and 100 Monte Carlo trials, we will obtain $3 \times 10 \times 100$ success/failure results. We then group the results for each query by the interval, count the number of successes, and divide that number by 100. If the confidence intervals are in fact accurate, this should result in 30 numbers that are all close to 0.95. The results are plotted in Figure 10.

Discussion. Each of the values in Figure 10 are closely clustered around 0.95, just as one would expect if the bounds were in fact correct. The only somewhat anomalous result is the one 0.89 result for Q1; but even that is not too exceptional, since there is a 5% chance of seeing only 89 correct bounds if in fact the true probability of correctness were 95% (this is a simple binomial probability).

12.6 Speed

The final issue that we consider is the speed of the Turbo DBO system. We were somewhat unsure as to whether including these results in the paper was a good idea. Currently, Turbo DBO is not built for speed. In particular, our implementation of the IMJ could be drastically improved (see below). But in the end, we felt that including some numbers is informative. In particular, we wanted to provide at least some evidence that DBO's co-processor-like architecture, where the IMJ sits outside of the normal data access path and "snoops" for lucky tuples, is not too cumbersome and need not greatly affect performance of a database system.

Setup. In an attempt to measure the cost of the IMJ and the vari-

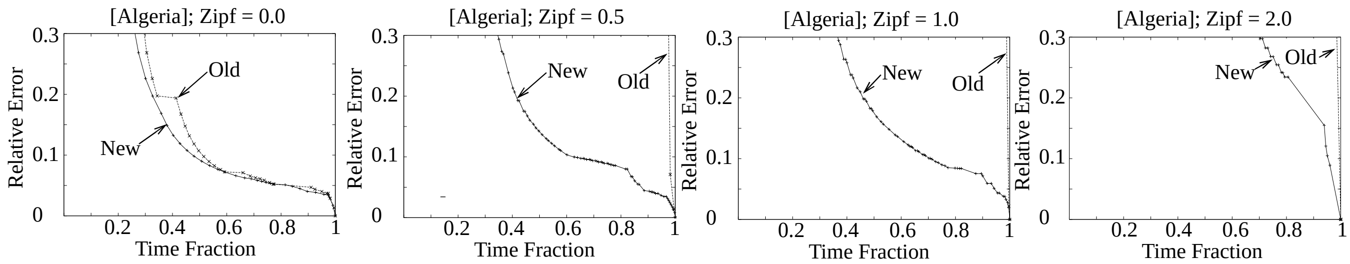

Figure 9: Confidence bound width as a function of time for data with increasing skew.

ous estimation algorithms, we prepared a stripped-down version of Turbo DBO, by ripping out as much of the machinery described in this paper as was possible. Tuples in this modified system are not streamed into the IMJ, and so no extra processing is required compared to what one might expect in a traditional system.

We then ran each of the four queries from the previous subsections in both the original and stripped-down versions of Turbo DBO, and averaged the running times over ten runs. For Q1, the average running times were 862 seconds and 612 seconds, respectively. For Q2, they were 772 and 606 seconds, respectively. For Q3, they were 957 and 613 seconds, respectively. And for Q4, they were 688 and 485 seconds, respectively.

Discussion. In our current implementation, there *is* a significant hit associated with the IMJ and its associated algorithms. By removing these software components from the system, we were able to speed our Turbo DBO implementation by an average of 29% over the four queries. This is significant. However, we feel that it was not too large considering the extent to which the core systems issues associated with the IMJ implementation were overlooked in our Turbo DBO prototype. For example, we used a very naive IMJ implementation which incurred a huge number of cache misses; this coulds be addressed using appropriate methods [2]. Furthermore, our IMJ was implemented as a single thread. We are hopeful that by a much more careful implementation of the IMJ, this extra cost could be taken down to almost zero.

13. CONCLUSIONS

This paper has described Turbo DBO, which is a prototype database system that aims to combine scalable, disk-based query processing with "fast-first" estimation, in order to give a user an immediate idea as to what the final answer to his or her query will be. Turbo DBO owes much of its inspiration to the original DBO system [9], but Turbo DBO contains a number of innovations above and beyond DBO. Specifically, Turbo DBO makes use of a novel estimation strategy that searches for partial chains of tuples that may eventually grow into full tuples that will contribute to the final answer to the query. Since Turbo DBO can make use of partial results to guide its search, it can have a much lower "time 'til utility" (TTU) than the original DBO, where TTU is defined to be the time required until a useful estimate for the final query result can be produced. This is particularly the case for joins of many tables, or when the underlying query is highly selective, or when the data are skewed. For example, in the case where five large database tables are being joined, Turbo DBO's TTU is nearly 80% lower than original DBO's TTU. Even for standard, TPC-H-style queries where only one or two large tables are present in the query, Turbo DBO's TTU averaged nearly 30% less than original DBO without data skew, and 60% less with skew. The net result is that Turbo DBO substantially increases the set of queries that are amenable to fast-first estimation, compared with the current state-of-the-art.

Acknowledgements. Material in this paper was supported by the National Science Foundation under grant no. 0803511.

14. REFERENCES

[1] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *SIGMOD Conference*, pages 487–498, 2000.

[2] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[3] P. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *SSDBM*, pages 51–63, 1997.

[4] P. Haas, J. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB*, pages 311–322, 1995.

[5] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, pages 287–298, 1999.

[6] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.

[7] W.-C. Hou and G. Özsoyoglu. Statistical estimators for aggregate relational algebra queries. *ACM Trans. Database Syst.*, 16(4), 1991.

[8] W.-C. Hou, G. Özsoyoglu, and E. Dogdu. Error-constraint count query evaluation in relational databases. In *SIGMOD Conference*, pages 278–287, 1991.

[9] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. In *SIGMOD*, pages 725–736, 2007.

[10] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. *ACM Trans. Database Syst.*, 33(4), 2008.

[11] C. Jermaine, A. Dobra, A. Pol, and S. Joshi. Online estimation for subset-based sql queries. In *VLDB*, pages 745–756, 2005.

[12] A. Klein, R. Gemulla, P. Rösch, and W. Lehner. Derby/s: a dbms for sample-based query answering. In *SIGMOD*, pages 757–759, 2006.

[13] G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton. A scalable hash ripple join algorithm. In *SIGMOD*, pages 252–262, 2002.

[14] F. Olken. Random sampling from databases. *PhD thesis, UC Berkeley*, 1993.

[15] F. Rusu, F. Xu, L. Perez, M. Wu, R. Jampani, C. Jermaine, and A. Dobra. The dbo database system. In *SIGMOD*, pages 1223–1226, 2008.

[16] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.

[17] M. Wu and C. Jermaine. A bayesian method for guessing the extreme values in a data set. In *VLDB*, pages 471–482, 2007.