Online Feedback for Nested Aggregate Queries with Multi-Threading[†]

Kian-Lee Tan   Cheng Hian Goh[‡]   Beng Chin Ooi
Dept. of Computer Science, National University of Singapore

Abstract

In this paper, we study the progressive evaluation of nested queries with aggregates (i.e., the inner query block is an aggregate query), where users are provided progressively with (approximate) answers as the inner query block is being evaluated. We propose an incremental evaluation strategy to present answers that are certainly in the final answer space first, before presenting those whose validity may be affected as the inner query aggregates are refined. We also propose a *multi-threaded model* in evaluating such queries: the outer query is assigned to a thread, and the inner query is assigned to another thread. The time-sliced across the two subqueries is *nondeterministic* in the sense that the user controls the relative rate at which these subqueries are being evaluated. We implemented a prototype system using JAVA, and evaluated our system. Our results show the effectiveness of the proposed mechanisms in providing online feedback that reduces the initial waiting time of users significantly without sacrificing on the quality of the answers.

1 Introduction

Database management systems are increasingly being employed to support end users in their decision making. Such users have a three-fold requirement: (1) the answers are summary data that characterize the datasets or are derived from or based on summary data, (2) imprecise answers can be tolerated, i.e., "approximately correct" answers suffice, and (3) the answers must be obtained quickly [7]. As such, database support for efficient computation of summary statistics in the form of aggregation queries becomes very important.

Traditionally, aggregation queries are evaluated under a *blocking execution model* (i.e., all data are examined and all operations are performed before a final answer is returned) to obtain precise answers. However, with greater availability of large volumes of data, this is not only computationally expensive, but users would find the waiting time unacceptable. Recently, Hellerstein et. al. [7] proposed a promising approach called *online aggregation* to meet the users requirements. Instead of presenting a final answer to the user (after a long period of waiting), an aggregation query is evaluated progressively: as soon as some data are evaluated, approximate answers with their respective running confidence intervals are presented; as more data are examined, the answers and their corresponding running confidence intervals are refined. In so doing, users are occupied with approximate answers and can terminate the evaluation prematurely if these answers suffice for their decision making. However, the work is restricted to simple non-nested queries.

Nesting of query blocks is a very interesting and powerful feature of SQL. In fact, it has been noted that the nested form is often easier for users to formulate and to understand [12]. Existing research (on nested queries) has sought methods of reducing the evaluation costs, typically by transforming a nested query into a logically equivalent form that can be evaluated more efficiently [2, 3, 10]. However, the class of *nested queries with aggregates* is especially interesting. First, such queries are commonly encountered. For example, to list employees who are *high-earners* where "high-earners" are defined as a certain factor of the average salaries of employees, requires a nested query with an average function. As another example, to list departments whose budgets are less than a factor of the average salaries of employees in those departments is another nested query involving the average function. Second, nested queries with aggregates cannot typically be expressed in a single query without nesting. Third, while some nested queries are transformed into non-nested forms that can be optimized into and eval-

---

[†]This work is partially supported by the University Research Grant RP982694.

[‡]Deceased on 1 April 1999.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

uated as a single plan, nested queries with aggregates still require multiple separate queries. To employ the traditional blocking execution model (i.e., the inner query block is processed before the outer block is processed) to evaluate such nested queries would frustrate the users if they are forced to wait without feedback.

In this paper, we present a novel approach for providing rapid online feedback for evaluating nested queries with aggregates (i.e., nested query where the inner query block is an aggregate query). For simplicity, we restrict our discussion to single-level nesting; the principles discussed can be easily generalized to multiple levels of nesting. Our technique is similar but goes beyond the online aggregation approach presented in [7]. Processing nested queries with aggregates online pose several interesting challenges that non-nested queries do not offer. First, it is not clear what forms the answers to a query and how the results should be interpreted, given that the inner query is an aggregate query. Furthermore, the outer query may or may not involve another aggregate, and the outer and the inner query blocks may or may not be correlated. Second, it is not clear how processing of the query can be *optimally* time-sliced across the outer and inner subqueries. This paper presents our solutions to these two issues. For the first problem, the answer space to the query begins with a superset of the final answers and is refined as the aggregates from the inner query block are refined. For the intermediary answers to be meaningful, they have to be interpreted with the aggregates from the inner query. We also propose an incremental evaluation strategy to present answers that are certainly in the final answer space first, before presenting those whose validity may be affected as the inner query aggregates are refined. For the second problem, we propose a *multi-threaded evaluation model* where the different query blocks are evaluated concurrently in a multi-threaded fashion. The time-sliced across the two query blocks is *nondeterministic* in the sense that the user controls the relative rate at which these subqueries are being evaluated. We implemented a prototype system using JAVA, and evaluated our system. Our results show the effectiveness of the proposed mechanisms in reducing the initial waiting time without sacrificing on the quality of the answers.

The remainder of this paper is organized as follows. In the next section, we present an overview of the proposed approach, together with issues that need to be addressed. In Section 3, we present a multi-threaded model for evaluating nested-queries with aggregates. Sections 4 and 5 present the various mechanisms and algorithms that are needed for the evaluation of the inner and outer query blocks respectively. Section 6 present the design, implementation and evaluation of a prototype system. In Section 7, we review some related work, and finally, we conclude in Section 8.

2 Motivation and Overview

In this section, we shall present an overview of the proposed approach with an example to reiterate the motivation behind online aggregation, i.e., to provide fast response of (approximate) answers to users. Before that, we shall review the two types of nested queries with aggregates that are relevant to our work.

2.1 Types of Nested Queries with Aggregates

In [10], Kim presented a classification of nested query types, two of which involve aggregates in the inner query block. For illustration, we shall use the example of a large admissions database that records information on persons (of different nationalities) applying to graduate business schools. The database has the following schema:

```
applicant(pid,city,income,gmat)
location(city,country)
```

In the relation `applicant`, we assume that an individual is uniquely identified by `pid`, resides in `city`, earns an annual salary given by `income`, and has a GMAT-score of `gmat`. The relation `location` identifies the country which a given city is in.

2.1.1 Type-A Nesting

A nested query is said to be of `Type-A` nesting if the inner query block Q has no correlation with the outer query block (i.e., it does not contain a join predicate that references a relation in the outer query block) and if the `SELECT` clause of Q consists of an aggregate function over a column in an inner relation. An example is the query that asks for applicants that earn a salary greater than 60,000 and have GMAT scores higher than the average score of applicants from the states. The SQL expression of this query is given by the left expression of Figure 1.

Traditionally, this query is evaluated in two steps. In the first step, the inner query block is computed to determine the average GMAT score of US applicants. In the second step, the outer query is evaluated with the inner query block being replaced by its answer. Unfortunately, step 1 is time consuming as the system needs to examine all result tuples in (`applicant` $\bowtie$ `location`) to compute the average function. Thus, it takes a long period of time before the answers to the (outer) query are returned to the user. This is frustrating to users, and new mode of computing such queries is desirable.

2.1.2 Type JA Nesting

`Type JA` nesting is present when the inner query block and outer query block are correlated (i.e., the `WHERE` clause of the inner query block contains a join predicate that references a relation of an outer query block),

```
Type-A query

SELECT a1.pid, a1.salary
FROM applicant a1
WHERE a1.salary > 60000
AND a1.gmat >
    (SELECT avg(a.gmat)
     FROM applicant a, location l
     WHERE a.city = l.city
     AND l.country = ``USA'');
```

```
Type-JA query

SELECT a1.pid, a1.salary
FROM applicant a1
WHERE a1.salary > 60000
AND a1.gmat >
        (SELECT avg(a2.gmat)
         FROM applicant a2, location l
         WHERE a1.city = a2.city
         AND a2.city = l.city
         AND l.country = ``USA'');
```

Figure 1: SQL expressions of sample Type-A and Type-JA queries.

and the inner `SELECT` clause consists of an aggregate function over an inner relation. An example is the query that asks for applicants that earn a salary greater than 60,000 and have GMAT scores higher than the average score of applicants from the same US city (see the right expression of Figure 1). This query can be evaluated efficiently by transforming it into the following two queries [10]:

Inner Query:

```
(SELECT city, avg(a.gmat)
 FROM application a, location l
 WHERE a.city = l.city
 AND l.country = ``USA''
 GROUP BY city);
```

Outer Query:

```
SELECT a.pid, a.salary
FROM applicant a, tmpAgg t
WHERE a.salary > 60000
AND a.city = t.c1
AND a.gmat > t.c2
```

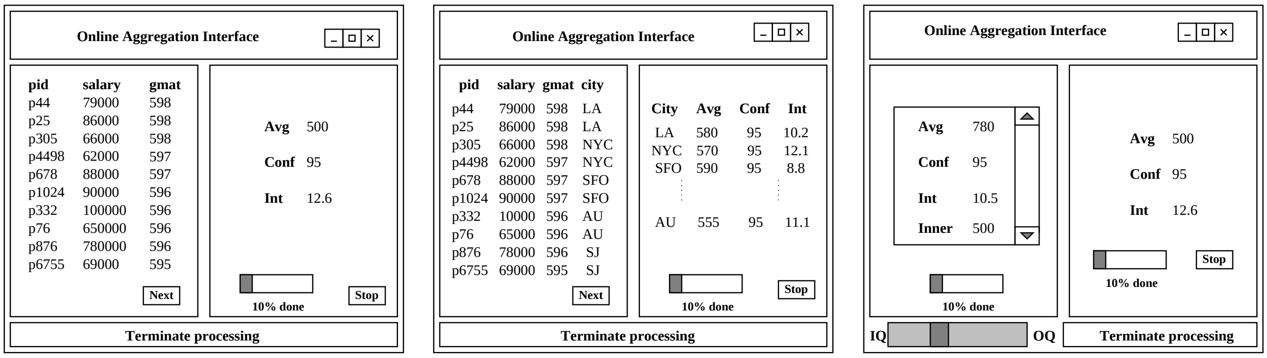
where `tmpAgg` is the temporary relation for the inner query, and `c1` and `c2` are the first and second columns in `tmpAgg` respectively.

Again, under the traditional query evaluation model, the inner query will be completely evaluated before the outer query can proceed. This may be unacceptable to users as the evaluation of the inner query will take a long time.

2.2 Overview of Multi-threaded and Online Evaluation of Nested Queries

The proposed approach works as follows. Instead of blocking the execution of the outer query block (until the inner query block completes), the outer query block is allowed to proceed as soon as the inner query block produces some estimates for its answers. In other words, the inner query block will be evaluated progressively to provide estimates quickly so that the outer query block can proceed to be evaluated (progressively). In this way, users can have rapid feedback (i.e., approximate answers) to their nested queries. Subsequently, both query blocks can be evaluated concurrently: as the inner query estimates are refined *progressively*, the answers to the outer query block are also refined based on the inner query block's refined aggregates.

The proposed user interface is shown in Figure 2. The interface will appear (almost) immediately after the user submits the query and can begin to display output as soon as the system has examined sufficient data to compute an estimate for the inner query. The interface consists of two panels. The right panel displays the result of the inner subquery: the current estimates of the aggregations together with the confidence and intervals that reflect the probabilistic estimates of the proximity of the current running aggregates to the final aggregate values. This result is updated regularly as more samples are examined. The % done and status bar display provide an indication of the amount of processing remaining before the computation of the aggregates completes. The left panel displays the output of the query, the result of which should be interpreted together with the current running aggregates. The **Stop** button in the right panel allows us to terminate the sampling process in computing the aggregates. The **Next** button in the left panel will retrieve the next answer set that satisfies the query. We note that terminating the aggregate computation does not necessarily mean termination of the query as we can still retrieve answers based on the current estimated aggregate values. For example, Figure 2(a) shows a sample display of the `Type A` nesting query in Section 2.1.1. From the right panel, we note that 10% of the work on the inner query has been performed, and the current running average for GMAT scores is 500 ± 12.6 at 95% confidence. From the left panel, the user can browse through the answer tuples that are retrieved based on the current estimate. In fact, for all the answer tuples that are displayed, the GMAT scores are much higher than the estimate, so much so that the user can be quite certain that these tuples will eventually be in the answer space. This



(a) Type A nesting. (b) Type JA nesting. (c) Outer block with aggregate.

Figure 2: An online nested aggregation interface.

clearly demonstrates the advantage of the proposed approach: *correct results can be returned to the user quickly compared to the traditional blocking execution model.*

As another example, consider the `Type JA` nesting query in Section 2.1.2. The sample display is shown in Figure 2(b). Again, from the right panel, the user can know the running average GMAT scores for the various US cities. The result displayed on the left panel shows those tuples that satisfy the query conditions using the inner query block's running averages.

We note that besides the targeted attributes, the answers include the additional attributes involved in the aggregation (`gmat` for the `Type A` nesting example, and `(city, gmat)` for the `Type JA` nesting example). This additional information allows the user to know which answer tuples are likely to be correct, and which are "approximately" correct (in the sense that future refinement may remove them from the final answer space).

The final example, which is more complex, is essentially a `Type A` nested query with aggregate function in the other query. The query

```
SELECT avg(a1.gmat)
FROM applicant a1
WHERE a1.salary > 60000
AND a1.gmat >
     (SELECT avg(a.gmat)
      FROM applicant a, location l
      WHERE a.city = l.city
      AND country = ''USA'');
```

asks for the average GMAT score of applicants that earn a salary greater than 60,000 and have GMAT scores higher than the average score of applicants from the states. The desired interface with sample results is shown in Figure 2(c). The display on the right panel is similar to the earlier examples. The left panel is essentially similar to the right panel, except that the results are presented differently. There is also the % done and status bar display to indicate the amount of processing remaining before the computation of the outer aggregate completes. We have also introduced a sliding bar to allow users to control the relative rate at which the two query blocks should be evaluated. Instead of displaying only one single aggregate that is based on the current estimate, the interface allows the user to view multiple average values based on different estimates from the inner query block. For example, the value shown represents the query's average when the inner query's average is 500. The confidence and interval represent the proximity of the average (of the query) at 95% confidence given that the inner query aggregate is actually 500. By clicking the up arrow at the top of the *scrollbar*, we will get the answer (query's average with the confidence and interval) for the case when the inner query's average is 499. Similarly, by clicking the down arrow at the bottom of the scrollbar, the answer with the inner query's average being 501 will be displayed. Such an approach allows the user to have a feel of what the average would be had the estimate change (as a result of refining the running aggregate for the inner query).

From the above discussion, we have identified a number of issues that have to be addressed to realize the proposed online evaluation of nested queries:

1. How can we interleave the execution of the inner and outer query blocks, i.e., how can we optimally timeslice the processing time across the two query blocks?

2. **Inner query block**. How much work must be done for the inner query block before the outer query block can proceed to be evaluated? How can the confidences and intervals be determined?

3. **Outer query block**. What are the *answer spaces* of a query (result of the outer query block)? The answer space of a query is the set of answer tuples relevant to the query. Since the inner query results are progressively refined, the answer spaces also change (as the outer query is evaluated based on the inner query's results). How should the user interpret these answers, whether they are likely

to be correct or approximately correct? Are there mechanisms to provide users with answers that are likely to be correct first, before those that are approximately correct?

In this paper, we shall focus on issues 1 and 3. Solutions to the second issue can be borrowed from the work in [4, 7], i.e., by randomly accessing the tuples from the inner query, we can apply the formulas in [4, 5, 7] to obtain the running aggregates and their confidence intervals.

3 A Multi-Threaded Nested Query Evaluation Model

Traditionally, query processing is performed under a *sequential* (or single-threaded) model, i.e., one task has to be completed before the next can be initiated. In other words, for nested queries with aggregates, the inner query has to be evaluated completely before the other query can be evaluated. Even if we can interleave multiple tasks (e.g., sample data from inner subquery, evaluate outer query, sample more data from inner subquery, evaluate outer query, etc.), the sequential model would make it cumbersome to facilitate the features discussed in Section 2.

In this section, we propose that the nested query be evaluated using a multi-threaded model. Under the multi-threaded model, two threads are used to evaluate the nested query in a concurrent fashion — thread `IQ` for the inner query block and thread `OQ` for the outer query block:

- Thread `IQ` evaluates the inner query block in phases. In the first phase, the estimates and their corresponding confidence intervals are obtained. In subsequent phases, these are refined.

- Thread `OQ` also evaluates the outer query block in phases (the number of phases is not the same as that of thread `IQ`). In the first phase, some answers are produced quickly based on the estimates obtained from thread `IQ`. Subsequent phases refine the answer spaces and produce refined answers or more answers.

The two threads operate in a *producer/consumer* relationship, where thread `IQ` produces some estimates of the inner query block (with increasing accuracy), which are then consumed by thread `OQ` in its evaluation of the outer query block. We note that the two threads have to be synchronized only once — when thread `IQ` must produce some estimates before thread `OQ` can begin evaluating the outer query. Subsequently, both threads operate concurrently and there is no need to synchronize between the two as thread `OQ` can use the current running aggregates from the inner query block to proceed.

In a single processor environment, the two threads have to be time-sliced. We have adopted the following approach:

- For nested queries that retrieve answer tuples, thread `IQ` is always in a *ready state*, i.e., it is always being processed to refine the inner block aggregates (except when it is being preempted by thread `OQ`, terminated by user or blocked because of I/O operations). On the other hand, thread `OQ` is always *suspended* except when the answers are to be produced. This occurs in the initial phase to produce the first answer set quickly, and subsequently, when the user requests for more answer tuples. In these instances, thread `OQ` resumes processing with a higher priority than thread `IQ`. This allows it to preempt thread `IQ` so that it can return answer tuples to the user rapidly.

- For nested queries that return aggregate values, both threads share equal time-slice by default. However, users may tune the time-slice by adjusting a sliding bar at the user interface (see Figure 2(c)).

In both the above cases, the evaluation of the outer query is "controlled" by the user. Thus, the allocation of time-slice between the two query blocks is essentially nondeterministic. In fact, the answers returned may also vary in the sense that the outer query may be based on different refined estimates at different time. In other words, for a user who takes a longer time to browse through each set of answers, each subsequent answer set will be based on a more accurate estimates (as more data are examined in the inner query). On the contrary, for a user that browses through the answers quickly, most of the answers are based on estimates that are more "crude".

4 Evaluation of Inner Query

In this section, we present the mechanisms and algorithms employed in evaluating the inner query in order to support the proposed online feedback approach.

The inner query is evaluated in phases, each of which produces a set of answer tuples from which the running aggregates can be estimated. At the end of each phase, the updated running aggregates will be reflected in the user interface as well as being passed to the outer query. We note that the aggregates are computed cumulatively. In other words, suppose there are k phases, and phase i ($1 \leq i \leq k$) produces t_i answer tuples; then, the aggregates at phase i are computed from $\sum_{j=1}^{i} t_j$ answer tuples. Moreover, the first phase is the most critical in the sense that sufficient answer tuples must be produced before meaningful estimates can be obtained. For subsequent phases, since the aggregates are computed cumulatively, the number of answer tuples is less of a concern.

There are two issues concerning the inner query evaluation:

1. For the inner query to provide meaningful running aggregates, mechanisms to generate or access random answer tuples (from which the aggregates are computed) are needed. Randomness is crucial since any biases may lead to poor estimates of the aggregate which are far from the actual aggregate values. In this paper, our focus is not on developing such strategies, and hence we opt to employ existing techniques. To access data randomly from a relation, we can employ the *heap scan* for heap files [7], *index scan* when there is no correlation between the attributes being aggregated and the indexed attribute [7] and the pseudo-random sampling schemes for B^+-trees [13]. For join queries, the ripple join algorithms [5] can be applied.

2. For the result (of the full query) to be useful, the estimate for the aggregate has to be meaningful. The proximity of the running aggregate to the actual value can be expressed in terms of a running confidence interval. The width of such a confidence interval serves as a measure of the precision of the estimator. Furthermore, we need to be able to determine the number of samples for the aggregate to be meaningful. In this paper, we restrict our work to *large-sample* confidence intervals based on the *central limit theorems* (CLT's) which contain the final answer μ with a probability approximately equal to p. [4, 7] provided the formulas and efficient methods to do so. In particular, given a confidence parameter p and a confidence interval half-width ϵ_n, let n denotes the size of a random sample required for constructing a large-sample confidence interval for μ that meets the given specification. Then $\epsilon_n^2 = \frac{z_p^2 s^2}{n}$ (see [4, 7]) where z_p is the $(p+1)/2$ quantile of Φ (the cumulative distribution function of an $N(0,1)$ random variable), so that $\Phi(z_p) = (p+1)/2$, and s is the sample variance.

5 Evaluation of Outer Query

In this section, we present the mechanisms and algorithms that the proposed approach adopt in evaluating the outer query. We shall present the evaluation strategies for outer queries without aggregates first, followed by outer queries with aggregates.

5.1 Outer Query Block Without Aggregate

Queries whose outer query blocks do not involve an aggregate operation are of the form:

```
SELECT target-list FROM relation-list
WHERE qualification
AND R.A op (inner query with aggregate);
```

where `R` is one of the relations in `relation-list` and `A` is an attribute of `R`, and `op` is one of the operations $>$, $\geq$, $<$, $\leq$, $=$. The sample queries in Section 2.1.1 and Section 2.1.2 are examples of queries in this category.

Traditionally, there is only one answer space (i.e., one unique set of tuples) to the query and the answers are the correct answers. However, under the multi-threaded evaluation model, the answers are based on estimates. Furthermore, as the estimates are refined, the answers may change. We shall first discuss the answer spaces and their interpretations before presenting the evaluation strategies.

5.1.1 Type-A Nested Query

Consider first the case when the nested query is of `Type-A` nesting. Let the set of running aggregates produced for the inner query block in the course of evaluating the nested query be $\overline{\mu}_1 \pm \delta_1$, $\overline{\mu}_2 \pm \delta_2$, ..., $\overline{\mu}_n \pm \delta_n$, where $\overline{\mu}_i \pm \delta_i$ is the running aggregate for phase i, and $\overline{\mu}_n$ is the final (actual) aggregate (and $\delta_n = 0$). Further, let the corresponding answer spaces be $\mathcal{A}_1$, $\mathcal{A}_2$, ..., $\mathcal{A}_n$.

Answer Space and Its Interpretation

Since $\overline{\mu_i}$ may take on different value in different phases, the answer space $\mathcal{A}_i$ is not likely to be the same as $\mathcal{A}_j$, for $i \neq j$. However, if the access to the tuples of the relations are random, we can expect the overlap in the answer spaces to be significant. We note that the concept of answer space is only a logical one and not all answers will be retrieved for the user at a single phase (recall that we only display a set of tuples each time). More specifically, suppose the current running aggregate is $\overline{\mu}-\delta \leq \mu \leq \overline{\mu}+\delta$. Then the corresponding answer space is given as follows.

- If the operation is "$>$" (or "$\geq$") (i.e., `R.A > aggregate` or `R.A` $\geq$ `aggregate`), the answer space includes tuples that satisfy the condition `R.A >` $\overline{\mu} - \delta - \epsilon$ (or `R.A` $\geq \overline{\mu} - \delta - \epsilon$) for some predetermined $\epsilon \geq 0$. In other words, the answer space should (hopefully) contain a superset of the final answers. In this way, we hope not to miss any answer should the actual aggregate be out of the range bound by the running aggregate.

- Similarly, if the operation is "$<$" (or "$\leq$"), the answer space includes tuples that satisfy the condition `R.A <` $\overline{\mu} + \delta + \epsilon$ (or `R.A` $\leq \overline{\mu} + \delta + \epsilon$) for some predetermined ϵ.

- Finally, if the operation is "$=$", the answer space will include all tuples that satisfy the condition $\overline{\mu} - \delta - \epsilon$ `< R.A <` $\overline{\mu} + \delta + \epsilon$.

As the estimate is refined, the current answer space is also refined accordingly — tuples retrieved that no longer satisfy the query will be removed; on the other hand, if too few tuples are in the current set, then a remainder query will have to be generated to retrieve the remaining tuples.

For the tuples to be useful, the user must have some means of ascertaining its quality, i.e., whether it is likely to be correct (in the final answer set), approximately correct (most likely to be in) or likely to be incorrect (likely to be out). Our approach is to include the attribute `R.A` in the target list (if it is not already user specified). In this way, the user knows exactly the value of `R.A`, and hence can take note of those tuples that are "fuzzy".[1] For example, if the operation is $>$, and the estimate has running aggregate of 150 ± 3, and an answer tuple has `R.A = 200`, then the user can be quite sure that this tuple will be in the final answer. Note that if the data are accessed randomly, it is unlikely for the estimator to be way out of the actual aggregate value.

Evaluation Strategy

Since answers are returned in sets (say 10 tuples each time), we propose that answer tuples that are most likely to be correct are returned to the users first. In this way, the users are likely to be browsing through correct answers (while the inner aggregates are being refined), and hopefully by the time they browse answer tuples that are towards the end of the answer space, the actual inner aggregates would have been computed and no (or few) poor quality answers are returned to users. Moreover, as argued in [7], it is not uncommon for users to terminate prematurely (i.e., before all tuples are retrieved), in which case, browsing correct answers at the initial phases would be highly desirable.

Let the running aggregate be $\overline{\mu} \pm \delta$. Let the nested predicate be `R.A > aggregate`. Furthermore, let the maximum value of `R.A` be `V`. This value is available in the DBMS statistics. The proposed approach splits the answer space into k partitions, each covering a range of $P = \lceil \frac{V-(\overline{\mu}+\delta+\epsilon)}{k} \rceil$. In other words, records in the range $[V-P,\ V]$ are in partition 1, records in the range $[V-2P,\ V-P)$ are in partition 2, and records in the range $[V-kP,\ V-(k-1)P)$ are in partition k. The proposed algorithm returns answers in order of the partition number, i.e., answers are obtained from partition 1, followed by partition 2, and so on. The algorithmic description of the strategy is shown in Figure 3.[2] The algorithm is highly abstracted. The routine **OuterThreadWithoutAggregate** is invoked whenever the user requests for more records (or when the very first set of results is to be displayed). It first obtains the current running aggregate, i.e., $\mu \pm \delta$ (routine **getRunningAggregate()**), and set ϵ to a fraction of the aggregate. Then, it calls the **next()** iterator multiple times (10 in the figure) to display a set of (10) records. The **next()** iterator is the cruz of the algorithm. It returns a record (according to the partition order) each time when invoked. It essentially comprises two fragments. The first fragment evaluates the outer query, returns answer tuples from partition 1, and produces the other partitions (lines 2-14). This is achieved by using another iterator **getNextResult()** that evaluates the outer query based on conventional algorithm, Note that only records that are in the answer space will be returned (for example, if `R.A > aggregate`, then the answers must satisfy $\overline{\mu} - \delta - \epsilon$). The answer tuple from **getNextResult()** is either returned (being an output of **next()**) if it belongs to partition 1 or written to the appropriate partition. The second fragment retrieves answer tuples from the partitions, beginning from partition 2 (lines 15-22). Since the inner query aggregate value may change, the tuples in the partitions have to be checked again with the corresponding running aggregate at the time when it is invoked. We note that if the inner query completes before the outer query completes, then $\delta = 0$ and $\epsilon = 0$.

When the nested predicates involve the operations $>$, $\geq$, $<$, $\leq$, the above strategy can be applied. However, for "=", we adopt the simple strategy of just presenting all tuples whose `R.A` values fall in the range $[\overline{\mu} - \delta - \epsilon,\ \overline{\mu} + \delta + \epsilon]$. This is because the number of tuples are not expected to be many (compared with those involving inequalities). Moreover, it is not common to have nested predicates with the $=$ operation.

A final note before we leave this section: it is "fine" for a displayed answer to fall out of the final answer space as the user would have the value of `R.A` for him to know the validity of the tuple.

5.1.2 Type-JA Nested Query

For `Type-JA` nested queries, the answer spaces, their interpretations and the evaluation strategy are essentially similar to `Type-A` nested queries. However, in each phase, we have multiple answer subspaces each associated with one of the running aggregate values in the inner query block (and multiple ϵ values). Moreover, the target list will include the pair (join attribute, aggregate value) so that user can access the validity of the answer tuples (see Figure 2(b)) for an example).

5.2 Outer Query Block With Aggregate

This category of queries has the form:

```
SELECT aggregate FROM relation-list
WHERE qualification
AND R.A op (inner query with aggregate);
```

[1] Alternatively, we can determine some statistical measures to reflect the confidence intervals of the answer tuples. This method is, in our opinion, less useful since the user is still not clear which tuples are valid.

[2] For ease of presentation, we have omitted the case when a remainder query is needed.

```
1.  OuterThreadWithoutAggregate() {
2.      getRunningAggregate(μ̄, δ, ε)
3.      for (i = 0; i < 10; i++) {
4.          r = next()
5.          if r ≠ ∅
6.              display(r)
7.          else
8.              exit()
9.          }
10.     }
11. }
```

```
1.  next() {
2.      while (more) {
3.          r = getNextResult(μ̄, δ, ε)
4.          if r = ∅ {
5.              more = false
6.              i = 2
7.          } else {
8.              j = determinePartition(r)
9.              if j = 1
10.                 return(r)
11.             else
12.                 cacheAnswer(r, P_j)
13.         }
14.     }
15.     while j ≤ n {
16.         r = getNextPartition(P_j, μ̄, δ, ε)
17.         if r = ∅
18.             j++
19.         else
20.             return(r)
21.     }
22.     if j > n return(∅)
23. }
```

Figure 3: Algorithm for Type-A query without aggregates.

where the outermost query involves an aggregate operation, `R` is one of the relations in `relation-list` and `A` is an attribute of `R`, and `op` is one of the operations $>$, $\geq$, $<$, $\leq$, $=$.

Traditionally, there is only one single answer tuple. Under the new query evaluation model, the outer query produces a set of answer tuples, each of which is a running aggregate, based on the inner query estimates. Consider first `Type-A` nested queries. Suppose the current running aggregate is $\bar{\mu} - \delta \leq \mu \leq \bar{\mu} + \delta$. Then, we compute the running aggregate of the outer query for a number of distinct values in the range $[\bar{\mu} - \delta - \epsilon, \bar{\mu} + \delta + \epsilon]$, for some predetermined ϵ. For example, let $\epsilon = 2$. If the inner query's running aggregate is 500 ± 3, then we can compute 11 running aggregates of the outer query: the first assumes that the actual aggregate value of the inner query is 495, the second for 496, and so on. Thus, the user can have quick feedback on what the outer-aggregate (with its confidence and interval) would be should the inner estimates be refined.

Figure 4 shows an abstracted algorithm. After determining the current running aggregate of the inner query block, the algorithm proceeds to obtain sample result tuples of the outer query (lines 6-9). This is again achieved with the help of the **next()** iterator. The **next()** here is similar to the one used in routine **OuterThreadWithoutAggregate()** except that the result tuples are generated using random access methods and ripple join algorithms (rather than conventional algorithms in iterator **getNextResult()**). The *stopping criterion* depends on how much timeslice is allocated for the thread. Next, assuming that there are n_{agg} aggregates to be computed for the outer query, the aggregates and their confidences and intervals are determined. Two points to note: first, if the inner query block completes execution, then $\delta_{iq} = 0$, $\epsilon = 0$ and n_{agg} will be reset to 1; second, the loop from lines 12-16 is only logical (for ease of presentation) in the sense that the actual implementation exploits the fact that the computation of the aggregates and the corresponding confidences and intervals at a later iteration can reuse results from earlier computation. Thus, evaluating multiple running aggregates is not that costly.

`Type-JA` nested queries are also interpreted and evaluated in a similar manner. The only complexity comes from the fact that the outer query involves an additional join operation.

6 Implementation and Evaluation

To study and validate the effectiveness of the proposed approach, we implemented the proposed approach. In this section, we shall present our findings. We use the initial response time (i.e., the time when the first answer set is presented) as the metric for comparison. The initial response time is taken to be the average value over multiple runs of the same experiment.

OuterThreadWithAggregate() {
  getRunningAggregate($\overline{\mu_{iq}}$, δ_{iq}, ϵ)
  $l_{min} = \overline{\mu_{iq}} - \delta_{iq} - \epsilon$
  $l_{max} = \overline{\mu_{iq}} + \delta_{iq} + \epsilon$
  currSample = $\emptyset$
  repeat
    r = next()
    currSample = currSample $\cup$ r
  until stopping criterion or $r = \emptyset$
  if $\delta_{iq} = 0$
    $n_{agg} = 1$
  for ($i = 0$; $i < n_{agg}$; $i++$) {
    $m_i = l_{min} + (i+1) * \frac{l_{max} - l_{min}}{n_{agg}}$
    computeConfIntervals($\overline{\mu_{oq}}$, δ_{oq}, conf, m_i)
    display($\overline{\mu_{oq}}$, δ_{oq}, conf, m_i)
  }

Figure 4: Algorithm for outer query with an aggregate.

6.1 Implementation and Experimental Setup

The proposed approach is implemented in Java on a SUN UltraSparc2 workstation. Java is the language of choice mainly because of its powerful graphical user interface components and its support for multithreading. The current implementation consists of basic components necessary to facilitate online feedback: an access method that retrieves data randomly, a hash-based ripple join algorithm, a statistical analysis routine for efficiently computing the confidence and interval of an aggregate, and the proposed incremental evaluation strategy. The current system only supports the average function.

We use the example database in Section 2. The database has the following two self-explanatory tables:

```
applicant(pid, fname, lname, live_in, income, gmat)
location(cityid, city, country, region, description)
```

Here, `pid` and `cityid` are keys of the respective tables, and `live_in` is a foreign key (and hence has the same domain as `cityid`). The `applicant` table has 1,000,000 tuples and are generated as follows: `eno` is set by counting 1 to 1,000,000, `income` is randomly picked from [10000,60000], `gmat` is generated using a normal distribution with mean of 550 and is restricted to the range [200,800]. The domain of `live_in` is the same as that of `location.cityid` and will be described shortly. We have fixed 50% of the applicants to be from the US. `fname` and `lname` are simply padded with "garbage" characters to ensure that the `applicant` table is 200 bytes long.

The `location` table contains 10,000 tuples, and is generated in a similar manner as `applicant`: `cityid` is set from 1 to 10,000; `region` is randomly distributed in [1,100] to reflect the region of a country (e.g., Asia, US, Europe, Australia); and `city`, `country`, and `description` are padded with dummy characters to make up a 200-byte location tuple.

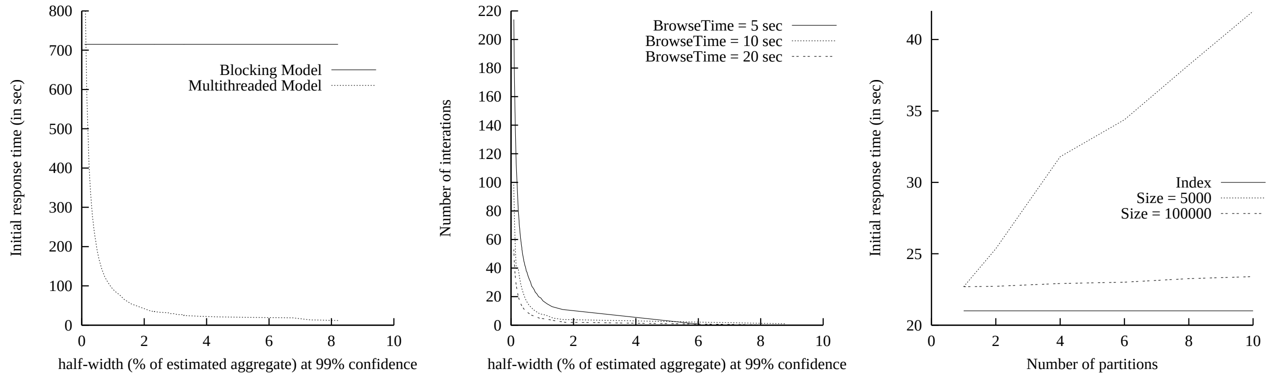
6.2 Experiment 1: Outer Queries Without Aggregates

In our first set of experiments, we study nested queries whose outer queries do not involve aggregates. The experiments are evaluated with the incremental strategy partitioning the answer tuples into two partitions. In all experiments, we set ϵ to be 1% of the estimated aggregate from the inner query block.

For the first experiment, we examine a `Type-A` nested query that selects the pid of applicants whose salary is greater than 59500 and the GMAT score is greater than the average GMAT-score of applicants from the US region. The resultant answer space has about 5000 tuples. Figure 5(a) shows the half-width of the confidence interval over the initial response time of the query. We note that the half-width is measured in terms of percentage with respect to the estimated aggregate value. As shown in the figure, the conventional blocking model takes more than 700 sec before the first set of tuples starts to appear on the display. For the proposed approach, the half-width of the interval is obtained with 99% confidence, and the evaluation of the `applicant` table in the outer query is based on a sequential scan of the table. We note that for the proposed approach, we can have the first set of tuples appearing as early as 10 sec when the half-width is about 8% of the estimated aggregate. As shown, even if we were to wait till the half-width is about 1% before we start evaluating the outer query, the first set of tuples would appear before 100 sec, which is far shorter than that of the conventional approach. Our investigation also shows that the answer tuples retrieved from partition 1 are all correct answers, i.e., they are in the final answer space. Thus, the result demonstrates that the proposed approach can provide quick and correct answers to users.

We also study the effect of user browsing time. As users take longer time to browse the result, subsequent requests will be based on better running aggregates. On the other hand, a shorter browsing time will imply that the results are likely to be based on running aggregates generated by fewer samples. We study three different user browsing time, t = 5, 10, and 20 sec, i.e., after every 10 tuples are retrieved, the user waits for t sec before requesting for the next set of tuples. Figure 5(b) shows the result of the experiment. As shown, in almost all cases, by the time the user has browsed through 20 answer sets, the half-width is already reduced to 1%. Moreover, these 20 answer sets are all from partition 1 which contains all correct answers.

Finally, we also study the effect of number of partitions on the incremental strategy. In this experiment, we fixed the half-width to be 5%, i.e., the outer



(a) Initial response time. (b) Effect of browsing time. (c) Effect of answer size.

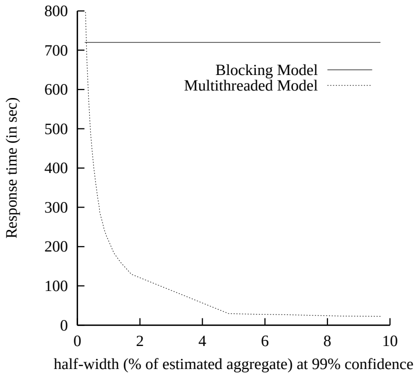
Figure 5: Type-A nested query where the outer query does not involve an aggregate.

query will be evaluated only after the half-width for the aggregate of the inner query has reached 5% of the estimated aggregate value. We vary the number of partitions from 1 to 10. The result is shown in Figure 5(c). We shall look at the curve with result size of 5000 tuples. As shown, it turns out that the number of partitions can be crucial (when the number of answer tuples is small, i.e., 5000 out of 1000000 tuples). For small number of partitions, the proportion of answer tuples falling into these partitions is large. As a result, it takes a shorter time to find a set of tuples in an arbitrary partition (as table `applicant` is sequentially scanned). On the contrary, for large number of partitions, each partition contains a small number of answer tuples; and hence finding a set of tuples in a partition will take a longer time (more tuples in `applicant` has to be scanned before we can obtain a set of tuples). The same figure also shows another curve with result size of 100000 tuples. This is produced by changing the selection predicate on salary to retrieve applicants who earn greater than 50000. However, we note that the number of partitions is hardly a factor in this case. The reason is because for large answer sizes, it does not take a long time to find a set of tuples to display. On the contrary, when the answer size is small, it will take a longer time to produce a set of tuples, and the time will be lengthened with smaller number of partitions. Thus, depending on the answer size, different number of partitions may be employed for optimal performance. We have also included a third curve which shows the initial response time had the `applicant` table been indexed. The response time to produce the first set of tuples is the same regardless of the answer size, and is clearly better than sequentially scanning the `applicant` table.

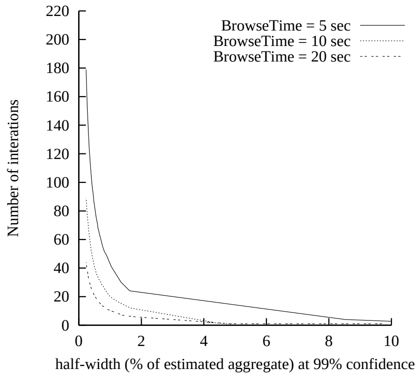
We also repeated the experiment for a `Type-JA` nested query. The query selects the pid of US applicants whose salary is greater than 50000 and the GMAT score is greater than the average GMAT-score of applicants from the same US city. Figure 6(a) shows the half-width of the confidence interval over the initial response time of the query for one city only (since the inner query generates an estimate for a city, there will be as many half-width as the number of cities). In our work, all cities have about the same number of matching tuples, and so, one is sufficient to serve as a representative example. As before, the half-width is measured in terms of percentage with respect to the estimated aggregate value. Once again, the conventional blocking model takes a long time before the first tuple is produced. However, we note that the initial response time is also slightly above 700 sec despite that the query is more complex now. The reason for this is because the number of US cities is about 100, and the intermediate results are being stored in the main memory instead of writing out to disk. To produce the first answer set to the query is thus not much slower than the case without a join operation.

For the proposed approach, the half-width of the interval is obtained with 99% confidence, and the evaluation of the outer query is based on a conventional hash-based join of the table (with the incremental strategy). We note that the proposed approach remains effective. The first set of tuples starts to appear at about 20 sec when the half-width is about 10% of the estimated aggregate. If we had picked a half-width of 1% before evaluating the outer query, the first set of tuples would appear at about 200 sec. While this is twice as much compared to the first experiment, it is still much shorter than that of the conventional approach. The initial response time is higher (compared to the first experiment) because the inner query is generating a set of estimates, and each has to have sufficient samples before the estimate can be meaningful. As in the earlier experiment, we note that all answers in the first partition are correct answers.

We also study the effect of user browsing time by using three different user browsing time, $t = 5$, 10, and 20 sec. Figure 5(b) shows the result of the experiment. As shown, if the user browsing time is low (i.e., 5 sec), it will take about 60 answer sets before the half-width is reduced to 1%. On the other hand, when the brows-



(a) Initial response time.



(b) Effect of browsing time.

Figure 6: Type-JA nested query where the outer query does not involve an aggregate.

ing time is 20 sec, it takes fewer than 20 answer sets before the half-width dropped to 1%. In any case, the number of answer sets is small in all cases. Again, all the 20 answer sets contain the correct answers.

From the experiments, it is clear that the multithreaded model proposed can provide quick answers to users without sacrificing on the quality of the initial answers.

6.3 Experiment 2: Outer Queries With Aggregates

In this experiment, we study the performance of the proposed approach for outer queries with aggregates. We shall evaluate the `Type-A` nested query that finds the average GMAT score of applicants whose salary is greater than 50000 and has GMAT score greater than the average GMAT-score of applicants from the US region. For this experiment, we also set ϵ to be 1% of the inner query's running aggregate. Figure 7 shows the result of this experiment. In this study, we use the default of equal time being allocated to each query block. In the figure, the number of iterations represents the number of timeslice that has been expended on each thread.

Our first observation (see Figure 7(a)) is that the proposed approach can provide very fast feedback to the user compared to the traditional blocking model. The traditional model requires evaluating both the inner and outer query blocks completely before producing the final answer tuple. Because of the large table sizes, this takes up a total of almost 1400 sec. On the contrary, the proposed approach starts to produce the first estimate on the 8th iteration which has an initial time of 11 sec only!

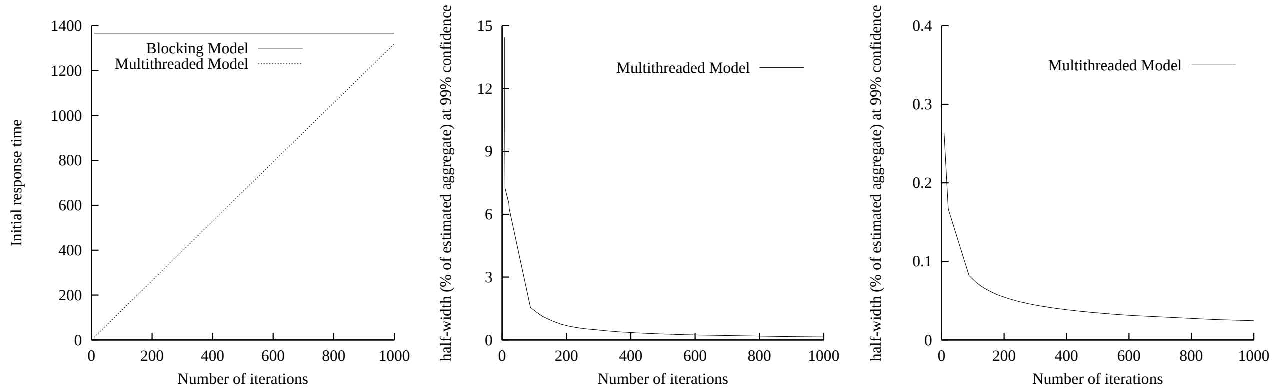
Figure 7(b) and Figure 7(c) show the half-widths of the inner and outer query running aggregates at 99% confidence respectively. From the figures, we note that the half-widths drop rapidly. Within 200 iterations (which is about 200 sec from Figure 7(a)), the half-width for the inner query has reduced to less than 1% while that of the outer query has dropped to less than 0.1% (this is only for one value of the outer query; the result is similar for other values). This results clearly show that the proposed approach is a promising alternative to the conventional blocking model: it can produce reasonably good answers to users quickly.

7 Related Work

In [7], Hellerstein et. al. proposed modifications to database engine to support online aggregation. These include techniques to randomly access data, to evaluate operations (such as join and sort) without blocking, to incorporate statistical analysis [4], etc. For complex aggregate queries involving joins, the authors also proposed a new family of join algorithms, called *ripple joins* [5]. Experimental studies showed that online aggregation is promising and can reduce the initial response time. Moreover, the confidence intervals converge in a reasonable time. However, these work focused on non-nested queries.

To facilitate online aggregation, it is important that records be accessed in random order and that the running aggregate be computed meaningfully for it to be useful. Sampling from base relations provide a means of randomly accessing records [8, 9]. In [13], Olken studied the methods to access records randomly from B^+-trees and hash files, and how random samples can be obtained from relational operations and from select-project-join queries.

When records are accessed in a random order, the running aggregate can be viewed as a statistical estimator of the final result. As such, the precision of the running aggregate to the final result can be expressed in terms of a running confidence intervals. In [4, 7], formulas for running confidence intervals in the case of single table and multi-table `AVG`, `COUNT`, `SUM`, `VARIANCE` and `STDEV` queries with join and selection predicates are presented together with methods to compute these formulas efficiently. Duplicate elimination with `GROUP-BY` operations are also considered.



(a) Initial response time. (b) Half-width of inner query. (c) Half-width of outer query.

Figure 7: Type-A nested query where the outer query involves an aggregate.

Several earlier work [6, 8, 9, 11] have also addressed the issue of obtaining confidence intervals.

There has also been some work on *fast-first* query processing, that returns the first few answers to users quickly. These work largely focused on developing pipelined join methods or cost models that can predict the cost to obtain the first few rows of a query result set [1, 15], and aim to minimize initial response time. More recently, Tan et. al. [14] studied how a query can be rewritten into subqueries so that users can obtain the answers to the first subquery quickly.

8 Conclusion

In this paper, we have proposed a mechanism to provide rapid feedback to users issuing nested queries with aggregates. The proposed approach evaluates a nested query progressively using a multi-threaded evaluation model: as soon as the inner query produces estimates to its aggregates, the outer query is evaluated to produce approximate answers to users; as the inner query's estimates are refined, the approximate answers are refined too. We have implemented a prototype system using JAVA, and evaluated our system. Our results showed that the proposed mechanism can provide rapid online feedback without sacrificing much on the quality of the answers.

References

[1] R. Bayardo and D. Miranker. Processing queries for the first few answers. In *CIKM'96*, Rockville, MD, 1996.

[2] U. Dayal. Of nests and trees: A unified approach of processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB'87*, pages 197–208, Brighton, England, September 1987.

[3] R. Ganski and H.K.T. Wong. Optimization of nested sql queries revisited. In *SIGMOD'87*, pages 23–33, June 1987.

[4] P. J. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *SSDBM'96*, pages 51–63, 1997.

[5] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD'99*, Philadelphia, June 1999.

[6] P.J. Haas, J.F. Naughton, S. Seshadri, and A.N. Swami. Selectivity and cost estimation for joins based on random sampling. *Journal of Computer and System Sciences*, 52(3), June 1996.

[7] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD'97*, pages 171–182, June 1997.

[8] W.C. Hou, G. Ozsoyoglua, and B.K. Taneja. Statistical estimators for relational algebra expressions. In *PODS'88*, pages 276–287, Austin, March 1988.

[9] W.C. Hou, G. Ozsoyoglua, and B.K. Taneja. Processing aggregate relational queries with hard time constraints. In *PODS'89*, pages 68–77, Portland, May 1989.

[10] W. Kim. On optimizing an sql-like nested query. *ACM Transactions on Database Systmes*, 7(3), September 1982.

[11] R.J. Lipton, J.F. Naughton, D.A. Schneider, and S. Seshadri. Efficient sampling strategies for relational database operations. *Theoretical Computer Science*, 116, 1993.

[12] G.M. Lohman, D. Daniels, L.M. Haas, R. Kistler, and P.G. Selinger. Optimization of nested queries in a distributed relational database. In *VLDB'84*, pages 403–415, Singapore, June 1984.

[13] F. Olken. *Random Sampling from Databases*. PhD thesis, University of California, Berkeley, 1993.

[14] K. L. Tan, C. H. Goh, and B. C. Ooi. On getting some answers quickly, and then more later. In *ICDE'99*, Sydney, Australia, March 1999.

[15] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in parallel main-memory environment. In *PDIS'91*, pages 68–77, Miami Beach, December 1991.