Chapter 4

QUERIES WITH MULTIPLE AGGREGATES FOR SINGLE LEVEL OF NESTING

4.1 An Overview

The purpose and framework of the ONAM have been outlined in the earlier chapter. In this chapter, an attempt is made to illustrate an overview of the approach that has been adopted in the new aggregation model. This overview serves to reiterate the motivation behind progressive evaluation, namely, providing fast response of approximate answers to the users. However, the two types of nested queries with aggregates relevant to this work may be reviewed first.
4.2 Types of Nested Queries with Aggregates

A classification of nested query types reveals two of the types involving aggregates in the inner query block [45]. This can be illustrated by using the following example of a large database.

The database may have the following schema:

ITD (Pval, Empid, Tax, Savings, Deductions, Earnings, Name, Address)

STUDENT (Pid, Income, Marks, Name, Address)

ITD10 (Pval, Earnings, Deductions, Savings, Tax, Policies, Name, Address)

In the relation ITD, it may be assumed that an individual is uniquely identified by Pval and Empid, makes Earnings, pays Tax, and makes Savings to get Deductions. The number of tuples present in the table is 40,00,000. In STUDENT table Pid is taken as the key, which indicates the personal identification number. Income earned, Marks obtained, Name and Address of the student are represented by the respective column names. The relation ITD10 is somewhat similar to the ITD except Empid column of ITD is replaced by the Policies column.

4.2.1 Type-A Nesting

A nested query is said to be of Type-A nesting if the inner query block Q has no correlation to the outer query block, that is, if 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 employees with Pval greater than the average Pval and
pays $Tax$ greater than average $Tax$. The SQL expression of this query is given by the expression of Query 4.1. Traditionally, this query is evaluated in two steps. In the first step, the inner query block is computed to determine the average of $Pval$ and average of $Tax$. In the second step, the outer query is evaluated with the inner query block being replaced by its answer. But step 1 is time consuming, as the system needs to examine all result tuples 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 the users, so a new mode of computing for such queries is necessary.

**Type-A Enumerated Query**

```
SELECT Empid, Name, Address
FROM ITD
WHERE Pval >
  (SELECT avg(Pval)
    FROM ITD)
AND Tax >
  (SELECT avg(Tax)
    FROM ITD);
```

*Query 4.1: SQL expression of sample Type-A Enumerative Query*

Again, under the traditional query evaluation model, the inner query is to be completely evaluated before the outer query can proceed. This may be unacceptable to the users, as the evaluation of the inner query takes a long time.
The next example, which is more complex, is essentially a Type-A aggregative nested query where the outer query block has an aggregate function. The Query 4.2

**Type-A Aggregative Query**

```
SELECT avg(Marks)
FROM STUDENT
WHERE Income >
(SELECT avg(Income)
FROM STUDENT);
```

Query 4.2: SQL expression of sample Type-A Aggregative Query

asks for the average *Marks* of students whose *Income* is greater than the average *Income*. The desired interface with sample results is shown in Figure 6.2. 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 are also the % done and status bar display to indicate the amount of processing remaining before the computation of the outer aggregate completes.

### 4.2.2 Type-CA Nesting

In Type-CA nesting, the inner query block and the outer query block are correlated. That is, the WHERE clause of inner query block contains the outer query column.

The following self explanatory query is an example for a Type-CA nested query.
SELECT S1.Pid, S1.Name, S1.Income
FROM STUDENT S1
WHERE S1.Marks =
   (SELECT max(S2.Marks)
    FROM STUDENT S2
    WHERE S2.Address = S1.Address)

4.2.3 Overview of Online Evaluation

Instead of blocking the execution of the outer query block until the inner query block is completed, 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 of 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 above discussion raised a number of moot questions, which have to be resolved to realize the proposed progressive evaluation of nested queries are as follows:

(i) **Threaded query execution**: How can the execution of the inner and outer query blocks be interleaved or how can the processing across the two query blocks be time sliced? What happens when the outer query block completes
the processing based on the estimated inner running aggregates before the inner query block does? In other words, how can the outer query block be refined?

(ii) **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?

(iii) **Outer query block:** What are the answer spaces of a query, that is, the 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 can the user interpret these answers, whether they are likely to be correct or approximately correct? Are there any mechanisms to provide users with answers that are likely to be correct before those that are approximately correct? How can the outer query be refined when some answers are missing when an estimated running aggregate is used? [27]

Issues (i), (ii) and (ii) are considered in this section. Some solutions to the second issue can also be obtained by accessing the tuples from the inner query, for the running aggregates and their confidence intervals [7].

### 4.3 Multi-Threaded Nested Query Evaluation Model

Traditionally, query processing is performed under a **sequential** or single-threaded model; requiring 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 completely evaluated before the outer query can be evaluated. Even if one can interleave multiple
tasks, like sample data from inner subquery, evaluate outer query, sample more data from inner subquery, evaluate outer query, and so on, the sequential model would make it cumbersome to facilitate the features discussed in Section 4.1.2.

In this section, it is proposed to evaluate the nested query using a multi-threaded model. Under the multi-threaded model, two threads are used to evaluate the nested query in a concurrent fashion. One thread is for the inner query block and the other thread is for the outer query block. The purpose of these threads are given below:

(i) Inner query thread evaluates the inner query block in phases. In the first phase, the estimates and their corresponding confidence intervals are obtained. In subsequent phases, they are refined.

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

The two threads operate on a producer/consumer relationship, where inner query thread produces some estimates of the inner query block with increasing accuracy, which are then consumed by outer query thread in its evaluation of the outer query block. It is noted that the two threads have to be synchronized only once: when inner query thread produces some estimates before outer query thread can begin evaluating the outer query. Subsequently, both threads operate concurrently and there is no need to synchronize the two, as outer query thread 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. The
following approach has been adopted:

(i) For enumerative queries, the inner query thread is always in a ready state, that is, it is always being processed to refine the inner block aggregates except when it is being preempted by the outer query thread, terminated by the user or blocked because of I/O operations. On the other hand, the outer query thread 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, the outer query thread resumes processing with a higher priority than the inner query thread. This allows it to preempt the inner query thread so that it can rapidly return answer tuples to the user.

(ii) For aggregative queries, both threads share equal time-slice by default.

In the first case, the user controls the evaluation of the outer query. 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 times. 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 estimate as more data are examined in the inner query. On the contrary, for a user who browses through the answers quickly, most of the answers are based on estimates that are cruder.

4.3.1 Evaluation Strategies for the Inner Query

The inner query of a nested query is evaluated progressively. After each tuple is retrieved, running aggregates and corresponding confidence intervals for each of the
columns are calculated separately. For calculation of the aggregates, the simple basic formula used is
\[ \mu = \sum_{i=1}^{n} v_i \]
where \( \mu \) is the mean(average), \( v_i \) is the value of the column of \( i^{th} \) tuple in relation to \( n \) tuples retrieved until now. The calculated aggregates and confidence intervals are reflected to the user interface.

There are two issues concerning the inner query evaluation:

(i) Random accession of the tuples

(ii) Estimation of Aggregates

For the inner query to provide meaningful running aggregates, mechanisms are needed to generate or access random answer tuples from which the aggregates are computed. Randomness is crucial since any bias may lead to poor estimate of the aggregates that are much different from the actual aggregate values. The following may be some of the solutions for the above two issues:

(i) To access data randomly from a relation, heap scan can be employed for heap files\[7\] index scan when there is no correlation between the attribute being aggregated and the indexed attribute \[7\] and the pseudo random sampling schemes for \( B^+ \)-trees \[3\].

(ii) For join queries, the ripple join algorithms can be applied \[14\]. The basic idea behind the ripple join algorithms on tables \( R \) and \( S \) is as follows. At all time, two sets of data are maintained, the \( r \)-tuples from table \( R \) and the \( s \)-tuples from table \( S \). Initially, both sets are empty. The algorithm randomly accesses an \( R \)
tuple, r, adds it to r-tuples and evaluates the join of r and s-tuples. Next, an S
tuple, s, is randomly accessed, added to s-tuples and the join of s and r-tuples
is evaluated. This process of randomly accessing an R tuple and an S tuple
alternatively is repeated until the entirety of relations R and S are examined.
The novelty of the proposed algorithms includes the flexibility to adopt the rate
at which R tuples or S tuples are retrieved.

All the formulae mentioned are simple to implement and are directly used from
the statistical information. The Central Limiting Theorem \( Z_{n/2} \frac{\sigma}{\sqrt{n}} \) has been taken
from the statistical information for use in the proposed ONAM.

By using the statistical formulae one can estimate the bounds for the aggregate
of the population. This is done when the aggregate of a sample is known and the
sample is made up of randomly chosen tuples from the relation. For this purpose the
following statistical formulae are used.

**Calculation of the Running Aggregate**

The basic formula used for calculating the aggregate is given by \( \frac{\text{Sum}}{\text{Count}} \) where \( \text{Sum} \) is
the total of values and \( \text{Count} \) is the number of values of a relation. This formula can
be used to calculate the aggregate of a column in a relation as

\[
\mu = \sum_{i=1}^{n} x_i
\]

where \( x_i \) is the value of the column of \( i^{th} \) tuple in relation to \( n \) tuples.

This formula can be used to calculate the aggregate of the column progressively.
The value of \( n \) is taken as 1 initially and incremented by 1 each time. When each
tuplet is retrieved, then the value of the column is added to the previous sum and divided by \( n \), which gives the current running average of the column.

**Calculation of the Running Confidence Interval**

The formula for calculating the large sample variance is given by

\[
s^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{\mu})^2
\]

These above-mentioned formulae cannot be used for calculation of the variance progressively using a computer program without losing performance. Hence the following formula has been derived from the above.

\[
\Rightarrow s^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i^2 + \bar{\mu}^2 - 2x_i\bar{\mu})
\]

\[
\Rightarrow s^2 = \frac{1}{n} \sum (x_1^2 + x_2^2 + x_3^2 + \ldots + x_n^2) + \bar{\mu}^2 - 2\bar{\mu}\frac{1}{n} (x_1 + x_2 + x_3 + \ldots + x_n)
\]

\[
\Rightarrow s^2 = \frac{1}{n} (x_1^2 + x_2^2 + x_3^2 + \ldots + x_n^2) + \mu^2 - 2\mu^2
\]

\[
\Rightarrow s^2 = \frac{1}{n} (x_1^2 + x_2^2 + x_3^2 + \ldots + x_n^2) - \mu^2
\]

This is the formula used for calculating the variance of the sample progressively in the program. The concept of sample here is that the sample starts with 1 tuple and ultimately grows to the size of the population.
The standard deviation is given by the formula

\[ \sigma = \sqrt{s^2} \]

The standard deviation is used to calculate standard error or the confidence interval level.

The interval is given by

\[ \varepsilon = Z_\frac{\alpha}{2} \frac{\sigma}{\sqrt{n}} \frac{N - n}{N - 1} \]

Where

\[ Z_\frac{\alpha}{2} = 1.96 \] for 95% confidence

\( N \) is the population size

\( n \) is the sample size

\( \frac{N - n}{N - 1} \) is the correction factor.

The interval is used to estimate the bounds of the aggregate of the population. If \( \bar{\mu} \) is the current running aggregate and \( \varepsilon \) is the interval with probability \( p = 95\% \). The aggregate of the population will lie between \((\mu - \varepsilon - \delta)\) and \((\mu + \varepsilon + \delta)\) with 95\% confidence. The interval tends to a zero as the size of the sample grows to the size of the population. The intervals can be calculated for any confidence level, say 90\% or 99\%. One of the crucial factors in estimating the boundaries of the aggregation of the population is the size of the random sample. The size of the sample must be at least 5\% of the size of the population to get the correct estimation of the aggregation of the population. Some correction factor \( \delta \) may be used to get some more accuracy in calculating the boundaries of the aggregation of the population.
The table consisting of values for $Z_{\frac{3}{2}}$ various confidence levels from 90 to 99 are enclosed in APPENDIX.

All the above formulae can be used in estimating the aggregation of the relation where

$x_i$ represents the value of the column in current tuple

$\mu$ represents the running aggregate

$s^2$ represents the running variance

$\sigma$ represents the running standard deviation

$\varepsilon$ represents the running confidence interval

The mechanisms and algorithms employed in evaluating the inner query to support the proposed online feedback approach are presented in this section.

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 and passed to the outer query. The aggregates are computed cumulatively. In other words, if there are $k$ phases, and phases $i$ ($1 \leq i \leq k$) produce $k_i$ answer tuples, then the aggregates at phase $i$ are computed from

$$\sum_{j=1}^{i} k_i$$

answer tuples. Moreover, the first phase is the most critical, in the sense that the 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 not of much concern.

47
4.3.2 Evaluation Strategies for the Outer Query

The mechanisms and algorithms that the proposed approach adopts in evaluating the outer query are presented in this section. First the evaluation strategies are presented for enumerative queries where the outer query blocks do not have aggregates, followed by aggregation queries where the outer query blocks involve aggregates. Then the strategies to refine the answer spaces are presented.

Enumerative Outer Query Block

Enumerative queries whose outer query blocks do not involve an aggregate operation have the following form:

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

where $R$ is the relation and $A$ is an attribute of $R$, and $op$ is one of the relational operators $\gtrsim, \geq, <, \leq$. The sample queries in Section 1.2 and Section 4.1.1 are examples of queries in this category.

Traditionally, there is only one answer space or 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. It is in the order of things to discuss the answer spaces and their interpretations before presenting the evaluation strategies.
Type-A Nested Query

The case when the nested query is of Type-A nesting may be considered first. The set of running aggregates produced for the inner query block in the course of evaluating the nested query may be \( \hat{\mu}_1 \pm \delta_1, \hat{\mu}_2 \pm \delta_2, \ldots, \hat{\mu}_n \pm \delta_n \).

where \( \hat{\mu}_i \pm \delta_i \) is the running aggregate for phase \( i \), and \( \mu_n \) is the final or actual aggregate with \( \delta_n = 0 \). The corresponding answer spaces could be \( A_1, A_2, \ldots, A_n \).

Answer Space and its Interpretation

Since \( \hat{\mu} \) may take on a different value in different phases, the answer space \( A_i \) is not likely to be the same as \( A_j \), for \( i \neq j \). However, if the access to the tuples of the relations is random, the overlap in the answer spaces could be significant. The concept of answer space is only a logical premise and not all answers will be retrieved for the user at a single phase since only one set of tuples is displayed each time. More specifically, if the current running aggregate is \( \mu - \delta < \mu < \mu + \delta \), then the corresponding answer space is given as follows:

(i) If the operation is '>' , that is, R.A \( \succ \) aggregate, the answer space includes tuples that satisfy the condition R.A \( \succ \mu - \delta - \varepsilon \) for some predetermined \( \varepsilon > 0 \). In other words, the answer space is expected to contain a superset of the final answers. This way, no answer will be missed even if the actual aggregate is out of the range bound by the running aggregate.

(ii) Similarly, if the operation is '<', the answer space includes tuples that satisfy the condition R.A \( \prec \mu + \delta + \varepsilon \) for some predetermined value of \( \varepsilon \).

(iii) Finally if the operation is '=' , the answer space will include all tuples that
satisfy the condition \( R.A \geq \mu - \delta - \varepsilon \) and \( R.A \geq \mu + \delta + \varepsilon \)

As the estimate is refined, the current answer space is also refined accordingly and tuples that no longer satisfy the query will be removed. On the other hand, if very few tuples are in the current set, then an ADMIT 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 their quality, whether they are likely to be correct in the final answer set, approximately correct or in or likely to be incorrect or out. This approach includes the attribute R.A in the target list if it is not already user specified. This way, the user knows exactly the value of R.A, and hence can take note of those tuples that are fuzzy. For example, if the operation is "\( \geq \)", and the estimate has the running aggregate of 2001729.553 ± 4334.404792, and an answer tuple has \( R.A = 3880123 \), then the user can be sure that this tuple will be in the final answer. If the data are accessed randomly, it is unlikely for the estimator to be out of the actual aggregate value.

**Multi-criterion Partition Approach**

The proposed partitioning approach for two aggregates is explained here. The running aggregates of R.A1 may be \( \mu_1 + \delta_1 \) and running aggregates of R.A2 may be \( \mu_2 + \delta_2 \). The maximum values of R.A1 and R.A2 are available in DBMS statistics. The maximum value of R.A1 may be \( V_1 \) and that of R.A2, \( V_2 \). This approach splits the answer space into 'K' partitions. The partition number in which a tuple falls has to be determined. For this purpose the partition number is identified. This process is repeated for each column that needs to be aggregated. To determine a partition number for a particular column value, it is required to find the range of each partition. Range of the partition is given by the formula \( P = [(V - (\mu - \delta - \varepsilon))/K] \) where 'P' is the partition number, 'V'
is the maximum value of that column or attribute, $\mu$ is the current running aggregate, 
$\varepsilon$ is the confidence interval, and ‘$K$’ is the total number of partitions. In other words, 
records in the range $[V-P, V]$ fall in partition 1, records in the range $[V-2P, V-P]$ fall 
in partition 2 and those in the range $[V-KP, V-(K-1)P]$ are in the partition $K$. This 
approach is followed when there is only one aggregate in the nesting.

When more than one aggregate function is used in the query, the partition of 
each column is identified individually and the maximum of the column partitions is 
taken as partition number of the tuple. In this approach, if a tuple has both $R.A1$ 
and $R.A2$ in the first partition then the partition in which the tuple falls will be the 
maximum of $(1, 1)$, that is, partition 1. Similarly, if a tuple has $R.A1$ in the first 
partition and $R.A2$ in the fifth partition, then the partition in which the tuple falls 
will be the maximum of $(1, 5)$, that is, $5^{th}$ partition. This way one can be sure that 
the most likely tuples of the final result set will be displayed first instead of the least 
likely. Even if $R.A1$ is in the first partition, it may not be in the final result set when 
$R.A2$ is not in any partition.

The calculation of running average of $R.A2$ does not require any extra thread. 
The same thread that is used for the calculation of running average of $R.A1$ can be 
used for $R.A2$ as well. This reduces the process overhead.

The proposed approach of the ONAM can also be used for the queries with more 
than two aggregates. The approach is simple to implement and is very effective. The 
$\delta$ and $\varepsilon$ values must be computed separately for each running aggregate and displayed 
to the user separately.

For enumerative queries, answer tuples are returned upon explicit request from 
the users. Thus thread EnumerativeOuterThread becomes active only when the users 
request for more answer tuples. If the current answer space is $A_{current}$, and the answer
space based on the refined aggregate value is $A_{refined}$, then there are three possible cases:

(i) The current answer space is not affected by the refined aggregate value, when $A_{current} = A_{refined}$. In this case, nothing needs to be done. A message indicating 'no more new tuples' may be displayed.

(ii) The current answer space contains the answer space of the refined aggregate value, that is, $A_{refined} \subseteq A_{current}$. In this case also nothing needs to be done. Although $A_{current}$ contains tuples that are not in $A_{refined}$, the users can make use of the additional attributes involved in the aggregation to determine their validity.

(iii) The current answer space is a subset of the answer space of the refined aggregate value, that is, $A_{current} \subset A_{refined}$. In this case, a refinement has to be done in order to obtain the missing tuples. The approach followed is detailed in the next section.

### 4.4 Refinement of Answer Space

Though the proposed approach, ONAM takes all the sets of tuples, which are most likely to occur in the final result set or answer space, still they require a refinement. This refinement is essential in both the following cases:

(i) Some tuples may not enter the final answer space even though they are present in the calculation of the final aggregate.

(ii) Some tuples may enter the answer space even though they may not be present in the calculation of the final aggregate.
4.4.1 ADMIT Query

The ADMIT Query is applied for the inclusion of the wanted tuples that have not been included. As shown in the preceding sections, there is a chance that the estimated aggregate of the inner query may fall below the calculated bound. This does not allow some of the tuples to enter the calculation of the outer query. This needs a special attention. Even though the occurrence of such state is remote, one must be prepared to handle this situation in case it arises. When the current outer aggregation fails to include some tuples in the actual answer space, an ADMIT query may have to be generated. In the following discussion, it is assumed that the operator used is ' \geq ', that is, R.A \geq aggregate.

The ADMIT query is necessary only when all the tuples are examined in the inner query and the inner query aggregate is less than the estimated outer query bound, that is, Bound \geq \mu - \delta - \varepsilon. In this case all the tuples that have been previously generated have to be again generated and those tuples that have R.A \geq aggregate and R.A < bound have to be picked and used in the calculation of outer aggregation. The Bound facilitates not only the case of removing the unwanted tuples but also the inclusion of the wanted tuples. If R.A \geq \mu - \delta - \varepsilon (13^{th} line of the algorithm) had been chosen to be used, it would have become very difficult to remove or include tuples since some of the tuples might have been included and others might have been skipped as a result of changing running aggregates.

As a part of the performance consideration, only one thread has been used to increase the efficiency by reducing the number of accesses to the relation. Each accessed tuple is used for calculation of inner aggregate as well as the outer aggregate at the same time. This makes the thread CPU intensive rather than I/O intensive. Since the query execution is done in a single thread, there is no overhead of thread
scheduling.

If the initial estimation of the bound for the inner aggregate is good enough, then there is no requirement for the ADMIT query to execute and the code (lines 40-52) can be eliminated from the algorithm. This gives scope for further refinement of the algorithm. Although a simple statistical formula is used for calculating the confidence interval values, better formulae can be explored from the work done on Online Aggregation [7]. When the early estimation of the bound is accurate then the remaining 5% tuples can be checked against the actual aggregate rather than against the bound. This reduces the number of tuples that must be removed at refinement stage.

**ADMIT Query Execution**

When an ADMIT query is generated, that is, when any one of the calculated bounds becomes greater than its current running aggregates then the partitioning approach is applied again on the newly generated tuples.

All the tuples that are above the current newly calculated bound, below previous bound and above the bounds of the other aggregates must be generated. But partitioning of the tuple is based on only those attributes because of which the new tuples are generated.

When the tuples are generated, if only one of the calculated bounds goes above its current running aggregate then the row partition will be done based on that column only. Previous bound of the column becomes the maximum value for the bound and the range is again calculated for the partition.
4.5 Algorithms

This section gives the proposed algorithms to carry out the ONAM for the queries with multiple aggregates with a single level of nesting.

1: int DetermineTuplePartition(t)
2: {
3: for each column of aggregate function do
4:   {
5:     p[i]=DetermineColumnPartition(c)
6:   }
7: end for
8: return max(p)
9: }

1: int DetermineColumnPartition(c)
2: {
3: pno=(NP-(c.val-c.bound)/c.range)
4: return pno
5: }

1: EnumerativeOuterThread()
2: {
3: if first then
4:   {
5:     for each column in multiple aggregate do
6:       {
7:         c.bound = c.\bar{P} - c.currentinterval

55
8:    c.range = ( c.maxval - c.bound ) / NP
9:    }
10:  end for
11:  first = false
12:  }
13:  end if
14:  for (i = 0; i < 10 ; i++) do
15:  {  
16:    t = next()
17:    if (t == 0) then
18:      exit()
19:    else
20:      display(t)
21:    end if
22:  }
23:  end for
24: }

1: tuple next()
2: {  
3:   while (more) do
4:  {  
5:    tp = DetermineTuplePartition(t)
6:    if ( tp == 1 ) then
7:      return(t)
8:    else
9:      list.add(t,tp)
10:    end if
11: }
12: end while
13: cur_par = 2
14: while (cur_par ≤ NP) do
15: {
16:    t = GetTupleFromPartition( cur_par )
17:    if ( t == 0 ) then
18:        cur_par++
19:    else
20:        return(t)
21:    end if
22: }
23: end while
24: if (cur_par > NP) then
25: {
26:    if ( for any column c.currentavg < c.bound ) then
27:        {
28:            c.maxval = c.bound
29:            first = true
30:        }
31:    else
32:        return 0
33:    end if
34: }
35: end if
Algorithm 4.1: Algorithm for Type-A Enumerative Query with Multiple Aggregates

The algorithm is split into various functions for easy understanding.

*DetermineTuplePartition* function calculates partition number of the tuple in which it falls. This function determines the partition number of each column by calling the function *DetermineColumnPartition* function. After determining each column partition number, it returns the maximum value as the partition number of the tuple.

*DetermineColumnPartition* determines the partition number based on column value, number of partitions (NP), value of the column in the tuple (c.val), bound and the range calculated.

*EnumerativeOuterThread* is the start of the algorithm. This thread is activated when the user clicks on the ‘Next’ button on the interface. Most of the time it is in a suspended mode. When this thread is activated for the first time, it calculates the bounds for each attribute present in the aggregate function. This is done based on the current running aggregates and the interval generated by the inner thread (lines 5-11). This bound is used as a condition for fetching the record from the relation. This thread displays ten (10) most likely records at a time when a user clicks the ‘Next’. The thread stops when there are no tuples to display (lines 12-19).

The next function is the key or heart of the algorithm. It retrieves the tuples that satisfy the bound conditions from the relation and determines the partition of the tuple by calling the *DetermineTuplePartition* function. If the tuple falls in the first partition it is returned immediately, otherwise tuple and its partition number
are stored in a list (lines 3-10). If all the tuples in the first partition are displayed then the tuples in the next partition are returned. If there are no tuples in the current partition then the next partition becomes the current partition (lines 11-19). If the user requests for more tuples after all the partitions have been displayed and the running aggregate goes below the bound, then some more tuples have to be generated that have been missed. In this case all the tuples that are between old bound and newly calculated bound are generated, that is, the ADMIT Query is executed (lines 20-29).

4.6 Conclusions

Various aspects of the ONAM with respect to the performance and trade-offs among various parameters for the queries with multiple aggregates for single level of nesting are given in this section.

(i) Number of Partitions

If the number of partitions is more, then the range of each partition will be small. If the range of partition is small, then the probability of number of records falling in that partition is less. Therefore the user has to wait for a long time for the initial set of answers to get displayed. If the number of partitions is small, then the range of partition is large and the probability is also more. Thus it takes less time to display the initial result set of answers.

(ii) No record in first partition to display

Sometimes there may be a situation that there is only one record in the case of a single aggregate and no records in the case of multiple aggregates in the first
partition. This causes the enumerative outer thread to scan complete relation to give the initial set of results. This takes a long time and the user may not be prepared for it.

(iii) Single Relation versus Multiple Relations in Subqueries

If each of the subqueries is from the same relation, only one thread can be used to calculate all the aggregates of subqueries. This increases the efficiency by reducing the number of accesses to a relation. When each subquery is from a different relation, then a separate thread is required to evaluate each subquery. The approach can be used against a query with aggregates of any number of columns within a relation. Although this approach has been explained with only the two aggregates within a nested query, it can be extended to a query with any number of aggregates within a nested aggregate query.

(iv) Rate at which the user presses the Next button

The rate at which the user presses the Next button also affects the evaluation of the inner query. If the rate of pressing the Next button is low, then the aggregate evaluation threads can take most of the CPU time to process the aggregates. If the user operates the Next button at a high rate, then the enumerated outer thread consumes most of the CPU time to display the tuples based on the current bound. Thus it takes more time to get the exact results, and vice versa.