A METHOD AND APPARATUS FOR OPERATING A DATABASE.

BACKGROUND

1. Field

The present application relates to databases, particularly relational databases and RDBMS (Relational Database Management System) implementations. The present application also relates to databases or systems used to provide directory services, such as X.500 service directories and LDAP service directories.

2. Description of the Related Art

It is considered that there is an ongoing need to provide systems which have improved performance over prior art systems, and in particular improved performance in the features of speed, performance and / or accuracy. Certainly, users seem to constantly seek an ever increasing improvement in speed, performance and accuracy.

It is not uncommon to have hundreds of thousands, or even millions, of entries in a database. Users of applications that use database systems that store a large number of entries, such as service directories, complain that the performance (execution) of such applications is often slow. A common complaint is that if a search is performed on such a large database, the search may take seconds, minutes or hours to provide a result. From a user's perspective, this sort of delay is considered irritating. Users prefer to have little, if any, delay in operating a database system.

This perception of lack of performance or lack of speed is especially noticeable when executing a search instruction/service. In essence, the problem is that in executing a search instruction/ service, the application has to take a complex directory query, such as a complex X.500 directory service query, and map that query into SQL instructions. The SQL performed is based on a table design, and these tables may use metadata. Even though the step of mapping the query is possible, conventionally the mapping is complex or produces relatively complex SQL expressions to be executed by the database application. In general, the more complex the SQL instructions, the slower the performance.

Nonetheless, users want the instruction/service executed in such a way that the system exhibits higher performance.

Part of what aggravates the performance problem is that, in a metadata design, the tables tend to be very large tables. The larger a table grows, the slower the application's perceived performance. Also, if the tables are large and a complex query joins a number of the large tables the perceived performance problem may get even worse.

To illustrate this problem, an example will be described of the mapping of a relatively complex X.500 query into SQL. The problem with the mapping is that X.500 and LDAP basically gives a query language that involves AND's, OR's and NOT's. SQL also gives a query language which involves AND's, OR's and NOT's. A complex SQL expression may use one or more joins and / or subselects. When there are a number of joins and / or subselects, the query may take a relatively long time to evaluate. There may be a number of ways to write an SQL query and if it only contains joins and not subselects then it is often termed a "flattened" query.

The illustration will be made in respect of a relatively simple example. Looking at Figure 1, a search for an X.500 query using LDAP notation is expressed in expression 101. The search example uses a filter to look for a common name = Rick, AND surname = Harvey. The figure shows a search table 102 and also an attribute table 103 for reference. Lets suppose that attribute identifier (AID) number 3 is for the common name (cn) and AID number 4 is for the surname (sn). In the search table there are a number of AIDs. The Rick Harvey entry is EID (Entry Identifier) 101 with AID 3, corresponding to a common name 'RICK' and also EID 101 has 'HARVEY' for the normalised value. There is also another AID 3 in the table for John Smith, John who has EID 200 and Smith who has EID 200. In doing a search for Rick AND Harvey, the search will try to find all objects (EIDs) who have AID = 3, norm = RICK, and AID = 4, norm = HARVEY.

So, in essence, the search wants to select entries 'Rick' and 'Harvey'.

One way of doing this search is by using a subselect (or nested query). The SQL required for this is:

```
Select EID from search
Where AID=3 AND NORM='RICK'
AND EID IN
    (select EID from SEARCH
        where AID=4 AND NORM = 'HARVEY')
```

In this nested query, the clause in brackets is a subselect. The subselect is evaluated corresponding to where AID = 4 and Norm = HARVEY and the resulting list of EIDs is saved. Then the outer clause is evaluated corresponding to where AID = 4 and Norm = HARVEY such that the list of EIDs returned is also in the list of EIDs previously saved.

The problem with the sub-select is that if there are many, many 'HARVEY's a huge set will be built and there may not be many 'RICK's and thus this query will be lopsided and may take a long time to evaluate.

Another way of doing this search is by using a join (or flattened query). The SQL required for this join is:

```
Select S1.EID from search S1, search S2
    Where S1.AID=3 AND S1.NORM = 'RICK'
    AND S2.AID=4 AND S2.NORM = 'HARVEY'
    AND S1.EID = S2.EID
```

The result is that if table S1 has a million entries, and table S2 has a million entries, the search may be conducted over a huge combination of entries when the tables are joined. As can be seen, even for this quite simple search/ instruction, performance of an application can be severely diminished in a relatively large database. However, usually a join version of a query will be faster than a subselect.

The sub-select is equivalent to the join, and in fact many prior art database applications may convert a sub-select into a join. However, this may be too difficult if there is more than one level of nesting or the where clause is too complex.

A further example will now be discussed where a search involves a 'NOT' instruction. In other words, in this example a filter asks for 'not equal'. We are looking for 'RICK' NOT 'HARVEY' that is common name is Rick and surname not equals Harvey.

The nested query, in SQL would be:

```
Select EID from search
Where AID =3 AND NORM = 'RICK'
AND EID NOT IN
        (select EID from search
        where AID =4 AND NORM = 'HARVEY')
```

The flatten version of the above query may be accomplished with an outer join. An outer join is considered to be quite complex and relatively slow in execution. An outer join for this search would be something like:

```
SELECT S1.EID FROM
        (SEARCH S1 LEFT JOIN SEARCH S2
                ON S1.EID = S2.EID
                AND S2.AID=4)
Where S1.AID=3
                AND S1.NORM = 'RICK'
AND (S2.NORM< > 'HARVEY')
```

Again, the above example relates to a relatively simple search for 'Rick' and not 'Harvey'; in other words, a search of sets involving set 'A' and NOT set 'B' i.e A.!B.

If we look at a search involving more complex searches, such as

A.(B+C.!D), ...expression 201

the SQL for this query would be (in abstract):

```
SELECT 'A'
        AND EID IN
                (SELECT 'B' OR
                        (SELECT 'C' AND EID NOT IN
                                (SELECT 'D')))
```

Note that the above query is very difficult to flatten into an expression that involves only joins. Also, note that "!D" means "NOT D"

SUMMARY

The present application provides, in one aspect, a method of processing a directory service query. In one embodiment, the method includes receiving a service query, applying principals of logic (e.g., DeMorgan's theorem) to the service query to obtain a sum of terms, evaluating each term as one or more separate SQL instructions, and executing each separate SQL instruction. The sum of terms may additionally be expanded to remove NOT operators, using for example Boolean logic.

In a further aspect, the present application provides, in executing a separate SQL instruction, a NOT can be expanded to subtraction. Directory service arrangements and databases for implementing the present application are also contemplated. In principle, the present application simplifies complex queries by reducing the query to a set of smaller or simpler SQL queries. By expanding a search filter into a sum of terms by the application of known principles of logic (DeMorgan's theorem). If a term includes a NOT, the term including a NOT can be expanded into a sum of negative terms. The performance of queries is improved by processing a series of separate SQL statements from such expansions Final results are achieved by combining the results of each of the separate SQL statements in a way that disregards duplicate SQL results. In one form, a term may be a set of AND operations. Furthermore, size and time limits of the instructions are improved by stopping the process of evaluating terms when a size or time limit occurs.

BRIEF DESCRIPTION OF THE DRAWINGS

A preferred embodiment of the present application will now be described, with reference to the accompanying drawings, in which:

Figure 1 illustrates a background example of a database search;

Figure 2 illustrates schematically a general method in accordance with the present application;

Figure 2a illustrates schematically in more detail a method of evaluating terms;

Figure 2b illustrates schematically in more detail a method of evaluating NOT terms; and

Figure 3 illustrates graphically an equation used in explaining the method of the present application.

DETAILED DESCRIPTION

The present application in its various aspects includes a number of steps. In the following description we will provide an illustrative embodiment of those steps.

With reference to Figure 2, an outline of the invention of the present application can be seen. The present application relates to a method of and apparatus for processing a directory service query. In essence, the query is received as represented by step 104. The query is represented as a logic expression as represented by step 105. The logic expression is expanded to an expression involving a sum of terms at step 106. Each expanded term is evaluated separately at step 107 and the results are collected at step 108, such that duplicates are ignored, and step 109 shows the end of the process of the present application.

The step 107 of evaluating each term is further illustrated in Figure 2a. From step 106 (Figure 2), step 110 determines whether the term involves a NOT operator. If the term involves a NOT operator, the term is evaluated at step 111, which will be more fully detailed with reference to Figure 2b. If the term does not involve a NOT operator, at step 112 the term is converted to SQL, and the SQL executed at step 113. Step 114 tests the results of the executed SQL for duplicates. If the results are a duplicate, the result is ignored at step 115. If the result is not a duplicate, the result proceeds to step 108 (Figure 2).

The step 111 of evaluating a NOT term is further illustrated in Figure 2b. From step 110, the term containing a NOT is expanded, which may result in positive and / or negative terms. The process illustrated in Figure 2b then diverges at step 117, one for positive terms, one for negative terms. At step 118, positive terms are executed. Step 120 tests the results of step 118 for duplicates. If the results are duplicates they are ignored (step 122). At step 119, negative terms are executed. Step 121 tests the results of step 119 for duplicates. If the results are duplicates they are ignored (step 122). Both positive and negative terms, which each are non-duplicate, are then subtracted at step 123. The result of the subtraction is passed to step 114 (Figure 2a).

A further description of the method of the present application is now provided with reference to the examples below.

EXAMPLE 1

Represent Query as an Expression

A service query can be represented mathematically as an equation. It is not necessary to represent a service query as an equation, but for the purposes of illustrating the method of the present application referring to an equation, does help to explain the present invention.

Expression 201 represents a general service query:

A.(B+C.!D).....expression 201

Expand Expression Using Known Principles of Logic

For the purpose of illustration, to expand an expression representing a service query known principles of logic is applied to the service query.

Using known principles, expression 201 can be expanded to produce what we call a sum of terms. In this example, the sum of terms for expression is:

Σ_{terms} = A.B + A.C.!D.....expression 202

The sum of terms represents the service query, i.e., the expression representing the service query, as an expression of OR of AND operators. This is what we can call a simple sum of terms.

The sum of terms provides a number of opportunities for further optimization:

1. opportunity to flatten terms
2. opportunity to stop processing if time or size limit hit
3. opportunity to evaluate terms in any order
4. opportunity to evaluate terms in parallel
5. opportunity to evaluate NOTs using a subtraction

Using known translation techniques equation 201 can be translated into an SQL instruction which, by inspection, could be degenerated into a nested set of SQL instructions using AND, OR and NOT operators. However, this is a general technique that results in slow performance. On a multimillion row table, it could take hours to execute an instruction, such as the search instruction represented by expression 201.

The present application provides improved performance over known translation techniques by avoiding nested SQL instructions. Nested instructions are removed by simplifying the expression representing the service query, e.g., expression 201, to a set of terms. Each term is then converted into a flatten query involving zero or more join(s) operations. In this way, a relatively long expression, such as expression 201, can be converted into a number of smaller instructions (or expressions), each of which can be flattened. The flattened and smaller instructions (or expressions) run much faster than the complex queries, thus improving performance. In addition, to further improve performance, the database could be configured to could choose the best technique of evaluating or processing the flattened terms.

Evaluating Each Sum Term Separately from the Expanded Expression

The method according to the present application can also improve performance of database queries by removing OR operators from the simplified expression. An OR operator can be removed in a sum of terms by evaluating each term as a separate SQL instruction. The applications builds the full result from all the partial results by summing all the partial results whilst ignoring

duplicates. In effect, the summation that the OR is performing is done by the application. Opportunities for further optimizations are noted above.

By applying known principles of logic (e.g., DeMorgan's theorem) to simplify the complex expression representing the service query, the OR operators have been pushed to the top and the AND operators have been pushed below the OR operators. In other words, in a relatively complex expression representing a service query, an inner nested OR operator gets multiplied out or expanded so that it 'pops' to the top. The method according to the present application can then remove the OR operators by processing (or evaluating) each term separately. In this manner, a relatively complex expression, such as expression 201, can be expanded out and executed separately so as to avoid nested SQL OR instruction performance liabilities.

Evaluating a Term Containing a NOT Operator From an Expanded Expression

If a term with one or more NOT operators is to be flattened, the method according to the present application preferably rewrites the term as an expression without NOT operators.

For example, the term A.!B can be expressed as A.(1-B), which can be written as A – AB. This expression no longer includes the NOT operator.

As another example, if a service query is represented as expression 203 below

A.C.!D.....expression 203

Boolean logic can be used to further expand the expression or a sum of terms, so that expression 203 can be expressed as:

A.C – A.C.D.....expression 204

This expanded form of expression 203 no longer includes the NOT operator.

It should be noted that a database that supports SQL may not supply a subtraction operator. In such instances a problem in processing the sum of terms as described above may arise. In order to process (or evaluate) a subtraction, the method according to the present application: collects all positive

terms in a list; collects all negative terms into another list; and then subtracts the positive term list and the negative term list whilst ignoring duplicates.

An alternative to the subtraction process noted above, is to collect all negative terms in a list, and in the process of collecting all positive terms in another list, only keep the terms that are not in the negative list. As a result, this positive list will have the subtracted results.

Evaluating a Term Having More Than One NOT Operator From Higher Order Subtractions

Expression 205 represents a service query with high order subtractions:

A.!B.!C.....expression 205,

where "!B" is "NOT B" and "!C" is "NOT C".

Expression 205 can be further expanded as follows:

A (1 – B). (1 – C).....expression 206,

A.(1 - C - B + B.C).....expression 207

A – A.C – A.B + A.B.C.....expression 208

Expression 208 can be further processed (or evaluated) to remove or ignore duplicate or overlapping results, where: the A operation provides a positive list; the AC operation provides a negative list; and the AB operation also provides a negative list. In order to evaluate the subtraction, the subtraction evaluation described above can be used in respect of the lists resulting from expression 208. It is to be noted that the lists can be evaluated in any order, or in parallel, or in accordance with the optimizations noted above.

By ignoring duplicate results there is no need to evaluate the term A.B.C. With reference to mathematical principals, the subtraction of A.B and A.C effectively subtracts A.B.C. twice, which is a duplicate result.

Looking at Figure 3, in a mathematical sense, if we are evaluating A.!B.!C, we can evaluate all of A, then subtract AC, then subtract AB. But this means graphically we have subtracted ABC twice (once in the AB subtraction and once in the AC subtraction). This is why mathematically ABC is again added in. In the present application, however, in executing service queries, results are listed, and it is realised that AC and AB have an overlapping portion called ABC. This

overlap is considered a duplicate because the results found by ABC have already been listed in AC and AB and thus they do not need to be again listed. The results of ABC are therefore not listed. Nor does the term ABC need to be evaluated as it will not be listed (added back in as was the case in the mathematical explanation). Each of these operations, A, AC, AB, some of which include AND operators can be evaluated separately and can be flattened out, without the need for subselects.

Note that A.!B.!C is an order 3 term, but this can be evaluated by A-AC-AB, which contains no greater than order 2 terms. Thus the implementation using subtraction lists is considered very efficient and results in improved database query performance.

It is clear that the method according to the present application can be generally applied to the evaluation / execution of database service queries. The present application should not be limited to the examples disclosed herein, and the expressions referenced herein are used for illustrative purposes. Clearly, the invention described in the present application can be used in respect of many different service queries, whether or not represented by equations of different simplicity or complexity.

I CLAIM

1. A method of processing a database service query, comprising:

receiving a service query,

applying principles of logic to the service query to obtain a sum of terms,

evaluating each term as a separate SQL instruction, and

executing each separate SQL instruction.

2. The method as claimed in claim 1, further comprising expanding each term to remove NOT operators.

3. The method as claimed in claim 2, wherein the a sum of terms are expanded using Boolean logic.

4. The method as claimed in claim 1, in which the service query is an X.500 or LDAP service query.

5. The method as claimed in claim 1, in which the service query is a search service query.

6. A method of processing a database service query, comprising:

determining a SQL instruction representative of a function;

listing the results of a subtracted SQL instruction in a first list, listing the results of a non-subtracted SQL instruction in an second list; and

not listing results which are duplicates of previously listed subtracted or non-subtracted results.

7. The method as claimed in claim 6, in which the service query is an X.500 or LDAP query.

8. The method as claimed in claim 6, in which the service query is a search service query.

9. A directory service arrangement including:

a database using a plurality of tables, each table having a plurality of rows and columns, and storing arbitrary data; and

means for processing a service query by applying principles of logic to the service query to obtain a sum of terms, evaluating each term as a separate SQL instruction, and executing each separate SQL instruction.

10. The directory service arrangement as claimed in claim 9, further including means to perform X.500 or LDAP services.

11. A directory service arrangement including:

a database using a plurality of tables, each table having a plurality of rows and columns, and storing arbitrary data, and

means for processing a service query by determining a SQL instruction representative of a function, listing the results of a subtracted SQL instruction in a first list, listing the results of a non-subtracted SQL instruction in a second list, and not listing results which are duplicates of previously listed subtracted or non-subtracted results.

12. The directory service arrangement as claimed in claim 11, further including means to perform X.500 or LDAP services.

13. A method for processing a database service query, comprising:

translating a service query to an expression;

simplifying the expression to a number of smaller expressions, each smaller expression being capable of being flattened;

flattening each smaller expression; and

executing each flattened expression.