Available at: www.researchpublications.org

Query Optimization in OODBMS using Query Decomposition and Query Caching

Atul Thakare
M.E (C.S.E) persuing
Sipna's COET,
Amravati (M.H) India
aothakare@rediff.com

Prof. Ms. S.S. Dhande
Asso. Professor
CSE Department
Sipna College of Engineering & Technology,
Amravati (M.H) India
sheetaldhandedandge@gmail.com

Dr. G. R. Bamnote
Professor & H.O.D
CSE Department
P.R.Meghe Institute of Tech. & Research,
Badnera (M.H) India

Abstract: Query optimization is of great importance for the performance of databases, especially for the execution of complex query statements. This paper is based on relatively newer approach for query optimization in object databases, which uses query decomposition and cached query results to improve execution times for a query. Multiple experiments will be executed to prove the effectiveness of this approach. The two main areas which will be covered are caching of query results and maintenance of the consistency of result sets after database updates.

Keywords: Query Caching, Query Decomposition, Query Optimization, Stack-Based Approach, Transaction Processing Stack-Based Query Language, Object Databases, Query Performance, Query Evaluation.

.

1. INTRODUCTION

In various types of Database Systems (Relational as well as Object-Oriented), many techniques of query optimization are available. Few of them are "Pipelining", "Parallel Execution", "Partitioning", "Indexes", "Materialized Views", "Hints" etc. The one technique which has not been convincingly implemented is "Query Caching". Huge performance benefits can be reaped out of "Query Caching" methodology which will be storing the cached query and its results. It is quite obvious that the cached results will provide very high performance benefits over results that are not cached. When there is a high probability of queries being repetitive in nature, "Query Caching" will provide optimum performance. Instead of spending time re-evaluating the query, the database can directly fetch the results from already stored cache. The most obvious benefit of "Query Caching" can be seen in systems where Data Retrieval rate is very high when compared to Data Manipulation. Data Manipulation can invalidate the cache results because the inserted/modified/deleted data can have direct impact on the cached results. The cached results will be out of sync which will necessitate regeneration of the cached results. Data Warehousing Systems, Decision Support Systems, Archival Systems are very good examples of Database Systems where "Query Caching" can be optimally used because Data Manipulation will be very low. Conceptually, the cache can be understood as a two-column table, where one column contains cached queries in some internal format (e.g. normalized syntactic query trees), and the second column contains query results.

2. OBJECT ORIENTED DATABASES AND STACK BASED APPROACH

We are going to experiment & study the results with query caching and decomposition of queries in object-oriented databases and by using the stack based approach.

2.1 Stack-Based Approach

Stack-Based Query Language (SBQL) is useful for the design and implementation of wide range of database models [1]. SBQL is developed according to the Stack-Based Architecture (SBA), a conceptual framework for developing object-oriented query and programming languages [2]. In SBQL the data is stored in the form of persistent objects and the collection of data objects is called as Object Store. Hence adding, deleting or updating information in Object-oriented Databases is nothing but adding, deleting or updating the objects. Objects may contain other objects (aggregation) or references to other objects. Hence the Object-Oriented Modelling concepts of complex objects, associations between objects, classes, types, methods, inheritance, dynamic roles, encapsulation, polymorphism, semi-structured data and other features are employed in the creation of Object Store Models, a representation of the database in Object Databases.

2.2 Operators in SBQL

SBQL permits the use of all well-known query operators such as selection, projection, navigation, path expressions, join, quantifiers, etc. SBQL has special as and group as alias operators, apart from binary operators [either of algebraic or non-algebraic type] [3].

In the evaluation of SBQL queries 2 stack are in use namely ENVS (Environmental Stack) and QRES (Result Stack). In the processing of algebraic operators ENVS is not required to be used [4]. The examples of algebraic operators are Operators for arithmetic and string comparisons, set-oriented operators, aggregate functions, auxiliary naming operators, Boolean operators, etc. In contrast, processing non-algebraic operators involves operations on ENVS. The examples of non-algebraic operators are selection (where), projection/navigation and path expressions (dot), dependent join (join) etc.

2.3 Distinction of SBQL Queries

In contrast to SQL and OQL, SBQL queries can be easily decomposed into subqueries, down to atomic ones, connected by unary or binary operators. This property simplifies implementation. Also decomposed atomic queries along with query caching plays an important role in query optimization.

3. OBJECT DATABASE MODELS

3.1 Database Store

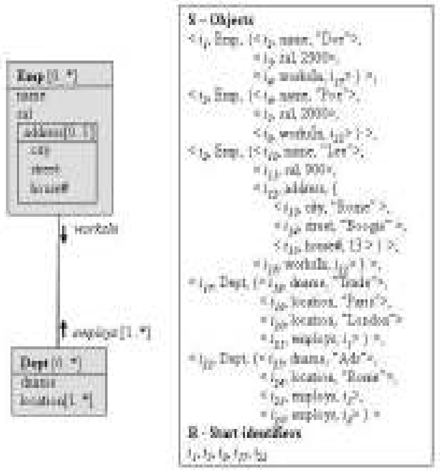


Fig.1 An example of the AS0 model

In the AS0 object database model all types of store objects (atomic objects, pointer objects and complex objects) can be represented by triplets <i, n, v>, as follows:

Atomic object: <i, n, v> where i is an internal identifier of the object [Unique Identification number (ID) assigned by the system], n is an external name assigned to the object [usually decided by the programmer], and v is a value of the object (e.g. an integer, a string, etc.)

Pointer (reference) object: <i1, n, i2> where i1 is an ID of the object, n is its external name, and i2 is an ID of the object to which I1 or a current object n is referring. Hence value of object I1 is a reference/address of object I2. As object I1 stores the reference we say it is a pointer/ reference object [5].

· Aggregation (complex) object: <i, n, T> where i is an ID or an identifier of the object, n is its external name assigned to the object at the time of its declaration, and T is a set of objects comprising the aggregate. Hence unlike atomic or pointer object, aggregate objects does not contain a single value, but it contains in itself multiple objects, where each contained object can be either of atomic or reference or of complex type.

3.2. Complex Objects

An example of a complex objects having three sub-objects is presented below:

<I5, Emp, {<I6, name, “Poe”>, <I7, sal, 2000>, <I8, worksIn, I22>}>

Here Emp is an external name of the object, I5 is its internal identifier, and it is comprised of 3 sub-objects namely name, sal and worksin. The ID’s of name, sal and worksin objects are I6, I7 & I8 respectively. Name and sal are atomic objects having values “Poe” and 2000 respectively whereas worksin is a pointer object containing an address of object I22. Each object’s ID is unique & is internal and non-printable[6].

As shown in the next diagram (Fig. 2) there are three Emp objects and two Dept objects in the database. Each Emp object represents one employee and contains his/her name & salary details in name & sal atomic fields (sub-objects) whereas worksin is the pointer field. Worksin field of each Emp object is having a reference of a Dept object which contains the information related to the department in which the employee is working. This reference in Employee object to the Department object describes the relationship between Employee and Department entities (like foreign key relationship in Relational DBMS). Dept object contains the department name, it’s one or more locations (branches) held in the atomic fields, and the n number of employs objects each of them pointing to one employee working in that department.

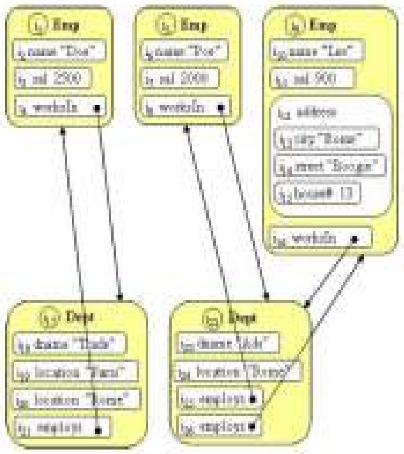


Fig. 2 graphical representation of a small database.

3.3. Evaluation of Query

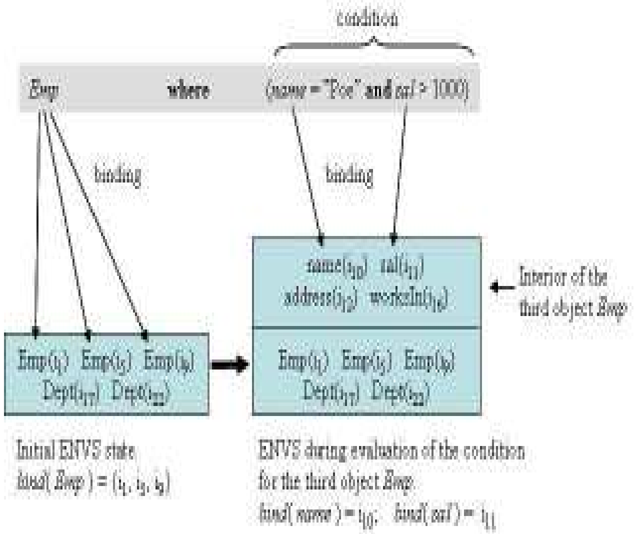

condition
Emp where (name = "Poe" and sal > 1000)
binding
binding
name(i_{10}) sal(i_{11}) address(i_{12}) worksIn(i_{16})
Interior of the third object Emp
Emp(i_1) Emp(i_5) Emp(i_9) Dept(i_{17}) Dept(i_{22})
Emp(i_1) Emp(i_5) Emp(i_9) Dept(i_{17}) Dept(i_{22})
Initial ENVS state
bind(Emp) = (i_1, i_5, i_9)
ENVS during evaluation of the condition for the third object Emp
bind(name) = i_{10}; bind(sal) = i_{11}

Fig.3 pushing a new section on ENVS to evaluate a condition.

Let us consider the evaluation of a query “Emp where (name = ‘poe’ and sal >1000)” by using the environmental stack ENVS [7]. Initially the base section of the stack contains the binders to global library procedures, environmental variables, views, user session objects, stored procedures, functions etc. As our query contains the table named Emp, the binders for all the objects of Emp type (I1, I5, I9) will be pushed on the top of the newly opened section on the stack. The binders of all the objects referenced by I1, I5, and I9 are also pushed on the stack (I17, I22). Each object of Emp type represents one row of Emp table which contains the information of one employee. Similarly each row of Dept table is described by each object of Dept type & it contains information related to one department. One by one all the three Emp objects (I1, I5, I9) are examined and whichever of them satisfies the given condition (name = 'poe' and sal > 1000), those will be added to the resultset object. Before the evaluation of condition for Emp object I1, a new section will be opened on the stack top, and the interiors of Emp object I1 (binders to name, sal, address and worksin of Emp Object I1) are pushed onto the stack. After evaluating the query for object I1, its interiors will be popped from the top & the next Emp object will be taken for the examination. Here we are just storing the references to data objects onto the stack and not the data itself. Secondly we are bringing only those object references on the stack, which points to the data required for calculating the result of the query. The Object binders or references on the stack will provide the means to access the Objects data (required for evaluating the query).

While evaluating the complex queries large number of data objects of various types may be required. As discussed earlier, the complex (nested) query will be decomposed into number of atomic sub queries [8]. An independent sub query (present at the last level) will be evaluated first (by using ENVS if it involves non-algebraic operator/operators) and its result will be stored in the result stack (QRES) along with the query. This result will be used in the evaluation of previous level query as required. The previous level query then along with its calculated result will be stored into the QRES and the interpreter will move to its previous level & so on. This simplifies the process of executing the complex query and at the same time improves the performance of database system, as the cached queries may form the part of other complex queries or the cached query may be resubmitted to database system as it is. Moreover in client-server architecture, if we keep the cache area on the server side, the query which is cached on the server side for one user, its results may be reused for servicing the requests from the same as well as the other users. Also the query which is being cached for a particular user in one of his database sessions can be reused for servicing the same user in his subsequent sessions. [Note: Volatile cache memory on the client side will lose its contents i.e. will lose cached queries & its results once the system is put off or user session ends]. On the other hand as the servers are up and running almost always, the server side cache arrangement will behave like persistent cache.

4 PROJECT WORK

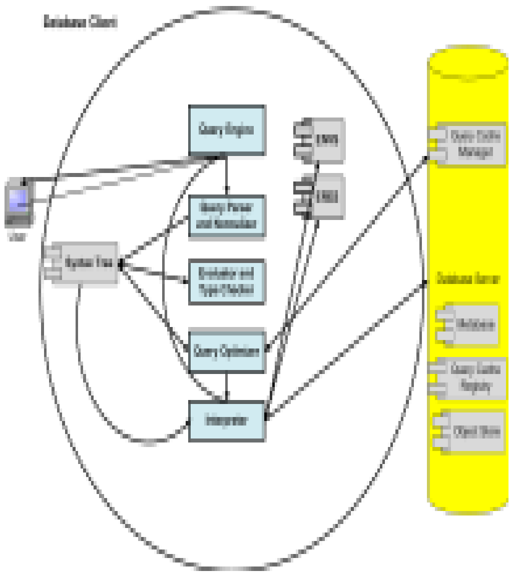


Fig. 4. Query optimization steps

4.1. Query Optimization Steps

The scenario of the optimization using cached queries in query evaluation environment for SBA is as follows (Fig. 4).

1) A user submits a query to a client-side user interface.

2) The user interface system passes it to the parser. The parser receives it and transforms into a syntactic tree

3) The query syntax tree is then received by static evaluator and type checker. It checks whether the query is syntactically correct or not. If not, it will report the errors. It also validates the tablenames, columnnames, operators, procedure names, function parameters involved in the query. Hence it will check the query's semantics. For this purpose, it will use the Metabase present on the server side. Metabase is a part of the database system which contains the Meta information related with the data in the various objects. This is static evaluation of the various nodes in the query syntax tree [9].

4) This type checked and statically evaluated query tree is then sent to the query normaliser which reconstructs the query according to the rules of normalization. This normalised query is then send to the query optimizer. All these components query parser, query type checker, query normaliser, query optimizer and query interpreter are employed on the client side system.

5) The cache optimizer rewrites the received normalized query using particular strategies like query decomposition. Each decomposed part of the complex query is send to the server. Server checks whether the received sub query is already cached or not. If sub query is present in cache, the Unique Identification number of the entry in cache which corresponds to the result of the given subquery is dispatched to the optimizer on the client side. Optimizer replaces (rewrites) the subquery tree of the query tree by a node containing that unique identification number. This is for maximum reuse of the cached queries. This rewriting will generate the best evaluation plan which promises to give the best performance & having a least cost in terms of time and storage.

6) The optimized query evaluation plan is then sent to query interpreter.

7) The plan is executed by the query interpreter [10].

4.2. Query Caching

Once the evaluation plan is executed successfully, the query is cached on the server side in pair <query, result>.

Following that the calculated result of the query is send to the client user who has submitted the query.

When the same query / a subquery (already stored with its result in server side cache) is submitted by the same or other user, after parsing, type checking and normalization of the query, optimizer sends the query to the server side query cache manager. Query cache manager searches for the query in the query cache registry and if found there will return the unique identification number (UIN) of the corresponding result to the client, thus avoiding the recalculation of previously stored result. Using this UIN, query interpreter (on the client system) can fetch the stored result of the query directly from the server.

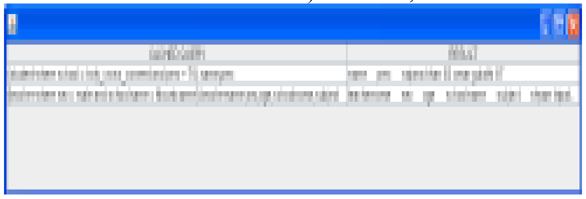
If the query is not found in query cache registry, query cache manager will send a message to an optimizer (on the client system) indicating that a query is not cached and hence its result needs to be calculated. Optimizer then does not rewrite the query i.e. does not reconstruct a parse tree. That part of the query will be then calculated by the query interpreter at runtime using runtime ENVS (Environmental Stack) and runtime QRES (Result Stack) [11].

Description of few components on server side:

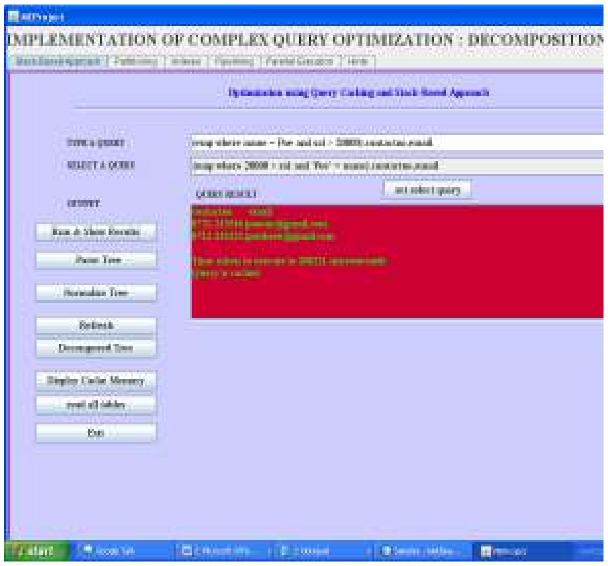
Query Cache Manager – This is a program running in the server and whose job is to check the Query Cache Registry and figure out if the query is cached. The Query Optimizer with pass a normalized query (or normalised inner sub-query) to the Query Cache Manager.

Query Cache Registry – This contains all the cached queries along with the results.

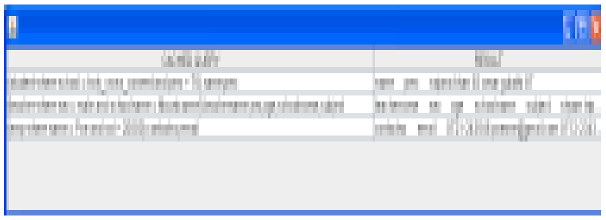
CACHE MEMORY before executing a query "(emp where name = Poe and sal > 20000).contactno,email":



When we Execute a query "(emp where name = Poe and sal > 20000).contactno, email" for the first time, It takes 108221 microsec as the query result is calculated. In this case, disk is accessed for fetching the data from the data files. The query along with it's result is stored in cache memory.

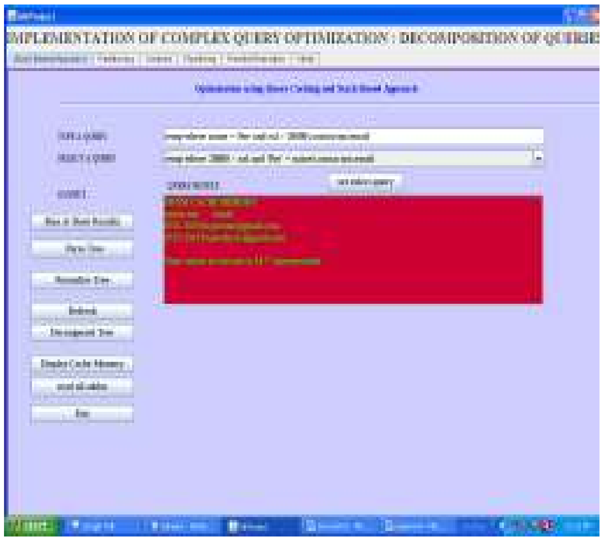


CACHE MEMORY after executing a query "(emp where name = Poe and sal > 20000).contactno,email":



When we reexecute the query "(emp where name = Poe and sal > 20000).contactno, email" result comes from cache memory and the time taken is 1477 microsec

In this case the disk is not accessed. And as we shown by the result the time taken by the query has been reduced considerably. We have experimented with number of such queries written in SBQL syntax and found that the time taken by the query get's reduced to approx. 1/5 to 1/100 times in most of the queries. We also observed that with the increase in the size of the database this difference keeps on growing. As the time required to fetch the result from the cache memory is apprx. Constant and the time required to calculate the resultset for the query by moving through the records in the database files will keep increasing (with sequential access) with the increase in the number of records in the database tables / increase in the size of data in datafiles. Even if query is more complex i.e. contains lots of aggregate functions and nested statements involving number of tables (hence involving costly join operations), then also the difference between time taken to return the previously calculated result and time taken to calculate the result of the query is large and keeps increasing with the complexity of the query.



4.3. Query Normalization

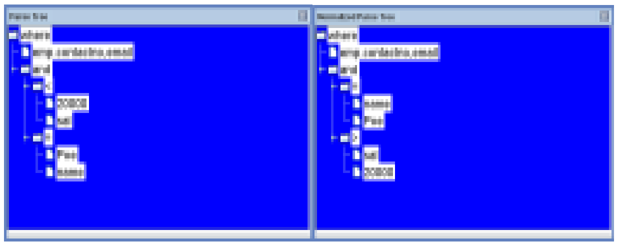
To prevent from placing in the cache, queries with different textual forms but the same semantic meaning (& hence also will generate the same result), several query text normalization methods will be used. Hence if a query is already stored in the cached with its result, all semantically equivalent queries will make reuse of the stored result, as all those queries will be mapped with the already stored query (due to normalization) [12].

Examples of few techniques useful in the process of normalization are:

a) Alphabetical ordering of operands
b) Unification of Auxiliary Names
c) Ordering based on column names (in the order in which they appear in the table description).
d) Column names should be maintained to the left side of each operator.

Consider a SBQL query
(emp where 20000 < sal and 'Poe' = name).contactno, email

It's Parse tree and normalised parse tree will be



And the normalized query we get from the normalized parse tree is:
(emp where name = Poe and sal > 20000).contactno, email
Hence any query which is semantically equal to the following query (may be written differently)
(emp where 20000 < sal and 'Poe' = name).contactno, email
will get transformed or normalised to
(emp where name = Poe and sal > 20000).contactno,email
It is the normalised query which will be stored in the cache.

```
Procedure parsetree()
{
1. Find the first "WHERE". This is the root node
2. The tablename before "WHERE" should be the left child node
Also mark the tablename node (MARKED).
3. Find the AND or OR if present in the query if not present goto step - 4
  3a the condition after the AND or OR should be the right subtree rooted at operator node.
  3b For each operator node set the value/columnname to its left as left node & right as right node.
  3c goto step-5
4. The condition after where should be the right child node.
  4.a Look for binary operators and create a node for them. (>, = etc)
  4 b.The values/columns before and after the binary operator become the left and right node.
  4 c. If there is a query after the binary operator then do the steps from 1 again. Till you reach the end.
4.        Get the list of columnnames following the ).
5.        Attach the list of columnnames to the MARKED node.
}
```

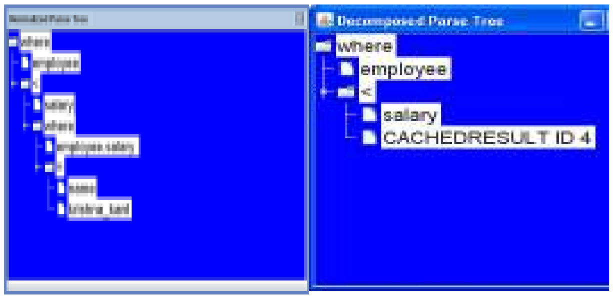
```
Procedure Normalise ()
{
operatorslist [] = {=,!=,<=,>=,>,<};
read a query
repeat for all the operators in the query
{
// ensures constants lies to the right of operator.
if(columnvalue is to the left of operator)
        swap the left & right side of the operator.
if(operator has on both sides table attributes)
{
        serialize the condition on tablenames in the list of tables in the database.
}
}
Repeat for each 'and' || 'or' in the query
{
//occuranceof returns the occurance number of the operator in the operators list.
if occuranceof (left-side condition operator) > occuranceof (right-side condition operator)
swap the left and right side conditions
else if occuranceof (left-side condition operator) == occuranceof (right-side condition operator)
{
//occuranceof returns the occurance number of the column in the table description.
if (occuranceof (left-side condition columnname) > occuranceof(right-side condition columnname)
        swap the left and right side conditions
}
}
For list of attributes after each ').'
        Rearrange the list of attributes by referring/according to table description
For I =0 to numberofauxiliarynames do
        Auxiliary-name[i] = "AUX" + I;
}
```

4.4 Query Decomposition and Rewriting

After normalization phase query is decomposed, if possible, into one or many simpler candidate sub queries. Query decomposition is a useful mechanism to speed up evaluating a greater number of new queries. If we materialize a small independent sub query instead of a whole complex query, then the probability of reusing of its results is risen [13]. Because the same query may occur as a sub query in many other queries, hence reuse of its stored result will speed up the performance of all those queries and as a whole of database system.

We have performed execution of the following complex query.
employee where salary < ((employee where name = 'krishna_kant').salary)
The above query is first decomposed into 2 queries
(employee where name = 'krishna_kant').salary) as v -------(1)
employee where salary < v ---------- (2)
The independent inner subquery i.e. (employee where name = 'krishna_kant').salary) as v is normalised first and verified on the server side (whether cached or not). As the result for this query was not readily available in cache the query is executed and the result of the query is stored in cache. The cache manager received the Cache ID of the query from query cache registry and returns this ID (Unique Identification Number) to the query optimizer. Query optimizer reforms the enclosing outer query as employee where salary < "CACHEDRESULT ID " + UIN
And then the outer query is executed during which inner query result is reused. After execution the result was returned to user

screen and the query “employee where salary < CACHEDRESULT ID 4” has been cached.
Hence during all this process two queries has been cached i.e. query (1) and (2). And the query (2) has a cache reference to the result of query (1). Any subsequent queries which has any parts of it mapping with any of the above two queries will make reuse of the stored results.
Hence in our optimization process, decomposition is an approach using which caching of the query starts with the smallest independent part of the total query and ends with the caching of the largest outermost query. Outermost query will have the reference to result of it’s inner query, inner query will have a reference to its inner query result and so on.



2.5 Update of cached results

Whenever any of the tables involved in the already stored query gets updated either by insert/delete/update operation, the stored result will be out of use or outdated. In this situation to restore the stored result back again to a useful state, we have to revaluate the stored query once again and overwrite its outdated result in the cache memory with a new one. Due to this, Query caching is not very useful in database systems where database is frequently updated. Also updation of single table in the database may affect ‘n’ number of cached results as updated table may be present in ‘n’ number of cached queries. Hence using effective algorithms and data structures for efficient updation of the cached results after database updates, effective searching in the cache memory to check whether the result is readily available or not, and better management of the cache memory (as with the growing number of stored results the overheads for searching into the cache memory will also increase) are critical issues and is in the future scope of this paper. For better management of the cache memory we may adopt the policies like cached queries which are least frequently used may be deleted from the cache memory after a specific time interval.

5 CONCLUSION

Based on the experimental results we can state that Decomposition and Caching techniques in Object Oriented Queries will result in upto 500% increase in performance and query output. High performance of these techniques will make these queries ideal for scenarios when Data Retrieval ratio is very high when compared to Data Manipulation. This is due to the fact that the cached results will not have to updated with the latest data at a frequent interval which in turn will boost the performance of the database. With more development in these techniques, the database can be a boon in the areas of DataWarehousing which work mostly in Data Retrieval mode.

REFERENCES

[1] J.Płodzień, A.Kraken. Object Query Optimization through Detecting Independent Subqueries Information Systems 25(8), Pergamon Press, September 2000, pp. 467-490

[2] J.Płodzień, A.Kraken. Object Query Optimization through Detecting Independent Subqueries Information Systems 25(8), Pergamon Press, September 2000, pp. 467-490

[3] K. Subieta, Catriel Berri Florean Matthes “A Stack Based Approach to Query Languages” Institute of Computer Science Polish Acedemy of Sciences, Report 738 Warszawa Dec 1993.

[4] OMG Object Database Technology Working Group: Next-Generation Object Database Standardization, OMG White paper, http://www.omg.org/docs/mars/07-09-13.pdf, September 27,

[5] Piotr Cybula, Kazimierz Subieta Decomposition of SBQL Queries for Optimal Result Caching Proceedings of the Federated Conference on Computer Science and Information Systems pp. 841–848

[6] Piotr Cybula, Kazimierz Subieta Decomposition of SBQL Queries for Optimal Result Caching Proceedings of the Federated Conference on Computer Science and Information Systems pp. 841–848

[7] P. Cybula, Cached Queries as an Optimization Method in the Object- Oriented Query Language SBQL. PhD thesis, Institute of Computer Science, Polish Academy of Sciences, Warsaw, 2010. In Polish.

[8] P. Cybula and K. Subieta, “Query optimization through cached queries for object-oriented query language SBQL,” in Proceedings of SOFSEM 2010, vol. 5901 of LNCS, pp. 308–320, Springer, 2010.

[9] K. Subieta, “Stack-Based Approach (SBA) and Stack-Based Query Language (SBQL),” http://www.sbql.pl/overview/, 2008.

[10] K. Subieta, C. Beeri, F. Matthes, and J. W. Schmidt, “A Stack Based Approach to query languages,” in Proc. of 2nd Springer Workshops in Computing, 1995.

[11] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham, “Materialized view selection and maintenance using multi-query optimization,” in Proc. of ACM SIGMOD, pp. 307–318, 2001.

[12] Piotr Cybula, Kazimierz Subieta Decomposition of SBQL Queries for Optimal Result Caching Proceedings of the Federated Conference on Computer Science and Information Systems pp. 841–848

[13] Piotr Cybula, Kazimierz Subieta Decomposition of SBQL Queries for Optimal Result Caching Proceedings of the Federated Conference on Computer Science and Information Systems pp. 841–848