Query Optimization in OODBMS using Query Decomposition & Query Caching

Atul Thakare  
M.E (C.S.E) pursuing  
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  
sheetaldhandedange@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. A query optimizer determines the best strategy for performing each query. These decisions have a tremendous effect on query performance, and query optimization is a key technology for every application, from operational systems to data warehouse and analysis systems to content-management systems. For example, query optimizers transform query statements, so that these complex statements can be transformed into semantically equivalent, but better performing, query statements. The query optimizer chooses, for example, whether or not to use indexes for a given query, and which join techniques to use when joining multiple tables. Query optimizers are typically ‘cost-based’. In a cost-based optimization strategy, multiple execution plans are generated for a given query, and then an estimated cost is computed for each plan. The query optimizer chooses the plan with the lowest estimated cost.

This report 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 were performed to prove the productivity of this newer way of optimizing a query. The limitation of this technique is that it’s useful especially in scenarios where data manipulation rate is very low as compared to data retrieval rate.

Index Terms: 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 for query optimization are available. Few of them are “Pipelining”, “Parallel Execution”, “Pipelining”, “Partitioning”, “Indexes”, “Materialized Views”, and “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. Hence database i.e. data store get modified after the long periodic intervals. During these intervals if a particular query is fired 1000 times then the result of the query is calculated only once i.e. for the first time and 999 times the stored result is reused. Data Manipulation can invalidate the cache results because the inserted/modified/deleted data can bring the difference between the cached results and the actual results. Hence regeneration of the cached results will be required to restore the results back again to the useful state. If data manipulation rate is not low, database system will have to spend a considerable amount of time in bringing the invalidated results once again into a valid state, thus forfeiting the advantage of using this technique. 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.

The experimentation for optimization of query will be done based on queries written in Stack based query language [SBQL] syntax which is designed and implemented using a stack-based architecture (SBA) framework. The performance gains will be measured by comparing the results against the performance of the identical queries written and executed in DB4o object database.

2. OBJECT ORIENTED DATABASES AND STACK BASED APPROACH

2.1 Object-Oriented Databases

An object database management system (ODBMS, also referred to as object-Object-oriented database management system or OODBMS), is a database management system (DBMS) that supports the modeling and creation of data as objects. This includes some kind of support for classes of objects and the inheritance of class properties and methods by subclasses and their objects [5].

2.2 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 [4].

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.

2.4 Object Store Models:

SBA assumes a family of object store models which are enumerated AS0, AS1, AS2 and AS31. The simplest is AS0, which covers relational, nested-relational and XML-oriented databases. AS0 assumes hierarchical objects with no limitations concerning the nesting of objects and collections. AS0 also covers pointer links (relationships) between objects [3].

2.5. Evaluation of SBQL Query

![Diagram of Emp where (name = ‘poe’ and sal > 1000)](image)

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 (I11, 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 will be evaluated first (by using ENVs if it involves non-algebraic operator/operands) and its result will be stored in the result stack (QRES) along with the query. This result will be used in the evaluation of immediate enclosing query & then this query with its result will be stored in QRES. This process will be repeated till the complete query is evaluated and then the final result along with the complete complex query will be stored. 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 queries may be resubmitted to database system as it is. Moreover, as we have the cache memory 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.
3.1. 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 query engine on the client side database interface. Query engine will pass the query to the parser. The parser receives it and transforms into a syntactic tree[6].
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[9]. 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 deployed on the client side database system. Whereas Metabase, Query Cache Manager, Query Cache Registry etc are on the server side DB system [7].
5) The query optimizer rewrites the received normalized query using particular strategies like query decomposition. Each decomposed part of the complex query is send to the query cache manager. Query cache manager checks whether the received sub query is already cached or not. If sub query is present in query cache registry, the Unique Identification number of the query result is dispatched to the query optimizer. Optimizer replaces (rewrites) the subquery tree of the total query tree by a node containing this unique identification number. This UIN will be used by query interpreter to directly fetch the result from the server side DB by consulting object management module. Hence all the parts of the query whose results are already stored in cache will get replaced by their respective UIN. Due to this all subqueries which gets replaced by corresponding UIN, their results will be brought from the cache & hence their re-evaluation will be avoided. 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].

Algorithm for query execution:
Step 1: receive the query.
Step 2: divide the query into set of tokens
Step 3: construct the parse tree from the set of tokens
Step 4: normalize the parse tree by applying normalization rules.
Step 5: traverse the normalized parse tree to get normalized text query.
Step 6: send the normalized text query to cache optimizer.
Step 7: decompose the normalized text query into set of queries.
Step 8: While the complete result of the query is not known

Do
Send the smallest independent subquery to the query cache manager.
If the result reference is received
Replace the subquery with the reference
Else
Calculate the result & Cache the query.
[endif]
Done

3.2. Query Caching

Once the query result is calculated for the first time (may be if available in the cache making reuse of results of some of its parts), the query is cached into the query cache registry 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 semantically equivalent query (written in the same or different way) is submitted by the same or other user, after parsing, type checking and normalization of the query, optimizer sends the query to the 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 query optimizer, thus avoiding the recalculation of previously stored result. Using this UIN, query interpreter (on the client side database system) can fetch the stored result of the query directly from the server side DB system.
If the query is not found in query cache registry, query cache manager will send a message to an optimizer (on the client side database 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.

Query Cache Registry before executing a query “(emp where name = Poe and sal > 20000).contactno,email”:

It has total 2 cached queries with the results.

When we Execute a query “(emp where name = Poe and sal > 20000).contactno, email” for the first time, It takes 108221 microsecs. The query result is calculated and for that a hard disk is accessed for fetching the data from the data files. The query along with it’s result is stored in Query Cache Registry.
Query Cache Registry after executing a query “(emp where name = Poe and sal > 20000).contactno, email”:

As expected as the executed query was not previously cached, its result was calculated and gets stored in cache along with the query.

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

In this case the disk is not accessed. And as 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 1/25 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 approx. 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 the increase in the number of records in the database tables. Even if query is more complex i.e. contains lots of aggregate functions and nested statements involving multiple number of tables (hence involving costly natural join/ theta join operations), then 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.

We execute 5 SBQL queries and got the results which are represented in the following bar chart.

**Performance Gain due to Caching**

Result 1 – Time taken by the query when executed for the first time (result calculated). [query1 – query5 \(\rightarrow\) queries with increasing complexity]

Results 2 - Time taken by the query when executed while its result being already available in cache. (Result reused).

Conclusion \(\rightarrow\) with increasing complexity of the query the benefits of caching keeps increasing. This is because complexity has a role to play when the result of the query is calculated and not when the calculated result of the query [previously saved] is brought from the cache. Time required to calculate the result of the query keeps increasing with the increasing complexity of the query while time taken to make reuse of previously calculated result remains almost constant.

**DB4o [with indexing] Vs simulator performance [with caching]**

I have also tried to execute some native queries in DB4o object database system and executed the similar queries written in SBQL syntax in our simulator. We have an identical database at the back end of DB4o as well as Simulator. I have used indexing as the optimization technique in DB4o native queries and used caching as the optimization technique in simulator.

**3.3. 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

\((\text{emp} \text{ where } 20000 < \text{sal} \text{ and } 'Poe' = \text{name}).\text{contactno, email}\)
It’s Parse tree and normalised parse tree will be

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 by the query cache manager (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 returns the Cache ID of the query 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 also 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.

The normalized query tree which will be stored in the cache.

I have created 3 data sources having 800, 8000, 800000 records respectively & in all 3 different scenarios tried to execute the same query. I found that with the growing size of the database the time taken to calculate the same query increases but the time taken to bring the stored result from the cache is almost constant. Hence the benefits of caching keep improving with the increase in the size of the database. Similar has been the results when we try to execute a set of queries with increasing complexity. Complexity of the query has the role to play only while calculating the result of the query and not while bringing the result of the query from the cache. With the increasing complexity the benefits of the query caching increases.

3.4 Query Decomposition and Rewriting

Step-1: Independent subquery “((employee where name = 'krishna_kant').salary)” gets executed & cached first & its cache
reference is used to reform the outer enclosing top level query as,

```
employee where salary < CACHEDRESULT id 4
```

Step-2: Query “employee where salary < CACHEDRESULT id 4” is executed next and gets stored in the cache with its result. The result is also send to the user screen.

**Future scope**

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 reevaluate 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 for a query 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 overhead for searching into the cache memory will also increase) are critical issues and is in the future scope of this project. 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 after a specific time interval. For effective and fast searching in the cache the techniques like indexing also can be used.

4. **CONCLUSION**

Query caching can provide good performance gains when the amount of I/O and CPU work required to repeat the same results is significant. Query Caching can be used in all types of database systems. Since the database for SBQL is object-oriented, this database can be very compatible with Object Oriented Languages. Also SBQL can be easily decomposed leading to better optimization and caching of queries. Although stack based approach is good in performance and has an advantage in object-oriented systems, still its early days and lot of work is required to be done, before it can seriously contribute in majority of databases.

Based on the experimental results we can state that Decomposition and Caching techniques in Object Oriented Queries have resulted in approx. 5 to 25 times 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 results 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 Data Warehousing which work mostly in Data Retrieval mode.

Through experimental results we can conclude that the time taken to calculate the result of the query increases with the growing size of the database but time required to fetch their result remains (with relatively small variation) constant. Hence the benefit of caching improves with the increase in the size of the database as well as with the increasing complexity of the query. This is because database size as well as complexity of the query is concerned factors only when calculating the result of the query and not the concerned factors when system is fetching the readily available (previously calculated) result of the query.

**REFERENCES**