Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 841–848

ISBN 978-83-60810-22-4

Decomposition of SBQL Queries for Optimal Result Caching

Piotr Cybula
Institute of Mathematics
and Computer Science
University of Lodz, Poland
Email: cybula@math.uni.lodz.pl

Kazimierz Subieta
Institute of Computer Science
Polish Academy of Sciences, Poland
Polish-Japanese Institute of Information
Technology, Warsaw, Poland
Email: subieta@ipipan.waw.pl

***Abstract*—We present a new approach to optimization of query languages using cached results of previously evaluated queries. It is based on the stack-based approach (SBA) which assumes description of semantics in the form of abstract implementation of query/programming language constructs. Pragmatic universality of object-oriented query language SBQL and its precise, formal operational semantics make it possible to investigate various crucial issues related to this kind of optimization. There are two main issues concerning this topic - the first is strategy for fast retrieval and high reuse of cached queries, the second issue is development of fast methods to recognize and maintain consistency of query results after database updates. This paper is focused on the first issue. We introduce data structures and algorithms for optimal, fast and transparent utilization of the result cache, involving methods of query normalization with preservation of original query semantics and decomposition of complex queries into smaller ones. We present experimental results of the optimization that demonstrate the effectiveness of our technique.**

I. Introduction

CACHING results of previously evaluated queries seems to be an obvious method of query optimization. It assumes that there is a relatively high probability that the same query will be issued again by the same or another application, thus instead of evaluating the query the cached result can be reused. There are many cases when such an optimization strategy makes a sense. This concerns the environments where data are not updated or are updated not frequently (say, one update for 100 retrieval operations). Examples are data warehouses (OLAP applications), various kinds of archives, operational databases, knowledge bases, decision support systems, etc.

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. A query result can be stored as a collection of OIDs, but for special purposes can also be stored e.g. as an XML file enabling further quick reuse in Web applications. A cached query is created as a side effect of normal evaluation of user query. A transparency is the most essential property of a cached query. It implies that programmers need not to involve explicit operations on cached results into an application program. In contrast to other query optimization methods, which strongly depend on the semantics of a particular query, the query caching method is independent of a query type, its complexity and a current database state.

Our research is done within the stack-based approach (SBA) to object-oriented query/programming languages. SBA is a formal theory and a universal conceptual frame addressing this kind of languages, thus it allows precise reasoning concerning various aspects of cached queries, in particular, query semantics, query decomposition, query indexing in the cache, and so on. We have implemented the caching methods as a part of the optimizer developed for the query language SBQL in our last project ODRA (Object Database for Rapid Application development) devoted to Web and grid applications [1]. In [2] we have described how query caching can be used to enhance performance of applications operating on grids.

There are two key aspects concerning the development of database query optimization using cached queries. The first concerns the organization of the cache enabling fast retrieval of cached queries (for optimal queries selection and rewriting new queries with use of cached results) and optimal, fast and transparent utilization of the cache, involving methods of query normalization with preservation of original query semantics (enabling higher reuse of cached queries for semantically equivalent but syntactically different queries), decomposition of complex queries into smaller ones and maintenance of assigned resources by removing rarely used results. The second problem is development of fast methods to recognize consistency of queries and automatic incremental altering of cached query results after database updates (sometimes removing or re-calculating).

In this paper we deal mainly with the first issue of the optimization method. The second aspect is widely researched in [3], [4]. The paper is organized as follows. Section II discusses known solutions that are related to the contributions of the paper. In section III we briefly present the Stack-Based Approach. Section IV shortly describes the architecture of the caching query optimizer. Sections V and VI contain the description of optimization strategies - query normalization, decomposition and rewriting rules. Section VII presents experimental results and Section VIII concludes.

II. Related Work

Cached queries remind materialized views, which are also snapshots on database states and are used for enhancing information retrieval [5], [6], [7]. The papers assume some restrictions on a query language expressions and cached structures. Such materialized views are currently implemented in popular relational database systems as DB2 and Oracle [8], [9], [10], [11]. Materialization of query results in object-oriented algebras in the form of materialized views is considered in [12] and [13]. Some solutions for view result caching at client-side in object and relational databases and for optimal combination of materialized results in cache to answer a given query are presented in [14] and [15]. In [16] and [17] a solution for XML query processing using materialized XQuery views is proposed.

There are, however, two essential differences between cached queries and materialized views. The first one concerns the scale. One can expect that there will be at most dozens of materialized views, but the number of cached queries could be thousands or millions. Such scale difference implies the conceptual difference. The second difference concerns transparency: while materialized views are explicit for software developers, cached queries are an internal feature that is fully transparent for them. Our research is just about how this transparent mechanism can be used to query optimization, assuming no changes to syntax, semantics and pragmatics of the query language itself.

New Oracle 11g database system [11] offers also caching of SQL and PL/SQL results. The cached results of SQL queries and PL/SQL functions are automatically reused while subsequent invocation and updated after database modifications. On the other hand, in opposition to our proposal, materialization of the results is not fully transparent. Query results are cached only when query code contains a comment with a special parameter `result_cache`, so the evaluation of old codes without the parameter is not optimized.

Query cache is also implemented in MySQL database [18], where only full `SELECT` query texts together with the corresponding results are stored in the cache. In the solution caching does not work for subselects and stored procedure calls (even if it simply performs a `SELECT` query). Queries must be absolutely the same - they have to match byte by byte for cache utilization, because of matching of not normalized query texts (e.g. the use of different letter case causes insertion of different queries into the query cache).

There is in Microsoft .NET query language LINQ [19] some kind of query result caching as an optimization technique for often requested queries, but it is also not transparent for programmers. They have to explicitly place the results of queries into a list or an array (calling one of the methods `ToList` or `ToArray`) and in a consequence each subsequent request of such query will cause getting its results from the cache instead of the query reevaluation.

But there is not any result caching solutions implemented in current leading commercial and non-commercial object-oriented database systems. Most of them bases their query languages on OQL (Object Query Language) proposed as a model query language by ODMG (Open Database Management Group) [20]. Only a cache of objects is introduced in some implementations for fast access of data in a distributed database environment.

III. Overview of the Stack-Based Approach (SBA)

The *Stack-Based Approach* (SBA) along with its query language SBQL are thoroughly described in [21], [22], [23]. SBA assumes that query languages are a special case of programming languages. The approach is abstract and universal, which makes it relevant to a general object model. The SBQL language has several implementations - for the XML DOM model, for OODBMS Objectivity/DB, and recently for the object-oriented ODRA system [1]. SBQL is based on an abstract syntax and the principle of *compositionality*: it avoids syntactic sugar and syntactically separates as far as possible query operators. In contrast to SQL and OQL, SBQL queries have the useful property: they can be easily decomposed into subqueries, down to atomic ones, connected by unary or binary operators. The property simplifies implementation and greatly supports query optimization. The SBQL operational semantics introduces two stacks, ENVS responsible for scope control and for binding names and QRES known as query result stack for storing temporary and final query results. The two stacks architecture is the core of SBA. The syntax of SBQL is as follows:

- A single name or a single literal is an (atomic) query. For instance, `Student`, `name`, `year`, `x`, `y`, `"Smith"`, `2`, `2500`, etc., are queries.
- If q is a query, and σ is a unary operator (e.g. `sum`, `count`, `distinct`, `sin`, `sqrt`), then $\sigma(q)$ is a query.
- If q_1 and q_2 are queries, and θ is a binary operator (e.g. `where`, `.`(dot), `join`, $+$, $=$, `and`), then $q_1\,\theta\,q_2$ is a query.
- There are not other queries in SBQL.

SBQL, unlike SQL and other query languages, avoids big syntactic and semantic patterns. Atomic queries are single names and literals. Nested queries can be arbitrarily composed from atomic and nested queries by unary and binary operators, providing they have a sense for the programmer and do not violate typing constraints. Classical query operators, such as selection, projection/navigation, join, quantifiers, etc. are also binary operators, but their semantics involves ENVS. For this reason they are called "non-algebraic" - their semantics cannot be expressed by any algebra designed in the style of the relational algebra. Below we present the exemplary operational semantics for one of the often used "non-algebraic" operator of projection (dot operator):

1) Initialize an empty bag (*eres*).
2) Execute the left subquery.
3) Take a result collection from QRES (*colres*).
4) For each element *el* of the *colres* result do:
    a) Open new section on ENVS.

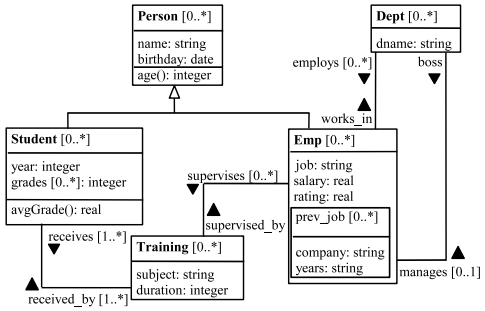

Fig. 1. Class diagram of the example database

b) Execute function *nested*(*el*).
c) Execute the right subquery.
d) Take its result from QRES (*elres*).
e) Insert *elres* result into *eres*.

5) Push *eres* on QRES.

Step 4b) employs a special function *nested* which formalizes all cases that require pushing new sections on the ENVS, particularly the concept of pushing the interior of an object. This function takes any query result as a parameter and returns a set of binders.

For the operator of selection (`where`) all steps are the same except for 4e) and a new 4f):

e) Verify whether *elres* is a single result (if not exception is raised).
f) If *elres* is equal to `true` add *el* to *eres*.

For the navigational join operator (`join`) the steps are:

e) Perform Cartesian Product operation on *el* and *elres*.
f) Insert obtained structure into *eres*.

For SBQL optimization examples presented in next sections we assume the class diagram in Fig. 1. The schema defines five classes (i.e. five collections of objects): `Training`, `Student`, `Emp`, `Person` and `Dept`. The classes `Training`, `Student`, `Emp` and `Dept` model students receiving trainings, which are supervised by employees of departments organizing these trainings. `Person` is the superclass of the classes `Student` and `Emp`. `Emp` objects can contain multiple complex `prev_job` subobjects (previous jobs). Names of classes (as well as names of attributes and links) are followed by cardinality numbers, unless the cardinality is 1.

IV. Query Optimizer Architecture

In most commercial client/server database systems (c.f. SQL processors) all the query processing is performed on the server. In SBA majority of query processing is shifted to the client side, to avoid server overloading and primarily to meet the orthogonal persistence principle, which implies, in particular, that a state involves persistent (server-side) and volatile (client-side) data on equal rights. Fig. 2 presents query processing architecture in SBA. Firstly, similarly to indices, the *query cache registry* is stored at the server. Hence the client-side query optimizer looks up in this registry before starts optimization and processing a given query. Secondly,

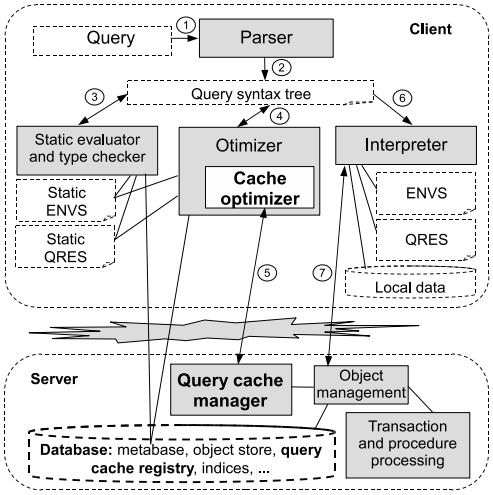

Fig. 2. Query optimization steps

in opposite to the traditional approaches, because only the client knows the form of the query and its result, the client is responsible to send the pair <*query*, *result*> to the server in order to include it within the query cache registry. The registry indexes cached queries with search keys being query texts, normalized using some sophisticated techniques mentioned in the next section. Non-key values of the index are references to nodes storing meta-information (MB_ID) and data (DB_ID), mainly compiled query and results, of cached queries.

The scenario of the optimization using cached queries in query evaluation environment for SBA is as follows (step numbers as in Fig. 2):

1) A user sends a query to a client-side database interface.
2) The *parser* receives it and transforms into a *syntactic tree*.
3) The tree is statically evaluated for *type checking* with the use of the static stacks (ENVS and QRES) and a database schema stored in the metabase at the server-side. After successful static evaluation the nodes of the query tree are augmented with type signatures for easier optimization reasoning.
4) The tree is sent to the *cache optimizer* being one in a sequence of optimizers employed at the client-side database system.
5) The cache optimizer rewrites it using strategies presented in next two sections including the precise algorithms for the most important methods of query normalization and decomposition. The optimizer employs the server-side *cache manager* which proposes optimal matching of results cached in the query cache registry, performs proper steps for a new query caching if suggested by the optimizer and maintains cache usage statistics for optimal cache utilization and cleaning. For each new cached query the manager generates additional structures, which describe a subset of involved objects

for maintenance purposes. The system updates cached results after changes in the database [3], [4].

6) The optimized *query evaluation plan* is produced and sent to *query interpreter*.
7) The plan is evaluated by the query interpreter. Some parts of the plan rewritten by the cache optimizer suggest taking the cached results from the server-side object store instead of reevaluation of them. For new queries being candidates for caching the interpreter generates their results and sends it to the cache manager for storing at the database server.

V. Query Normalization

To prevent from placing in the cache queries with different textual forms but the same semantic meaning we introduce several query text *normalization methods*. These methods are applied in a way of reconstructing a query text from early generated query syntactic tree or directly by change some nodes or their order within the tree.

Alphabetical ordering of operands: The method is suitable for operators, which for a succession of operands is not substantial, such as comparing operators ($=$, $\neq$, $\leq$, $<$, $>$, $\geq$), arithmetic operators ($+$, $-$, $*$, $/$), logical operators (`or`, `and`), operators of sum and intersection of sets, structure constructor (`struct`), i.e. a query:

```
Emp where salary >= 1100 or salary = 1000
```

is normalized to:

```
Emp where 1000 = salary or 1100 <= salary
```

The general algorithm of the method is presented on Algorithm 1.

Ordering of operators: Sum and multiply operations are put before subtractions or divisions [3], i.e. an arithmetic expression is transformed as follows:

```
a / b / c * d / e
```

is normalized to:

```
a * d / b / c / e
```

Unification of auxiliary names: Auxiliary names used by the programmer for `as` or `group as` operator are unified, but only if such an operator doesn't finalize the evaluation of the query (it is not the root of the syntactic tree, which case is easy to recognize based on query result signature evaluated earlier by the static evaluator), i.e. a query:

```
(((Emp where salary > 900) as e) join
(e.works_in.Dept as d)).(e.name, d.dname)
```

is normalized to:

```
(((Emp where salary > 900) as $cache_aux1)
join ($cache_aux1.works_in.Dept
as $cache_aux2)).($cache_aux1.name,
$cache_aux2.dname)
```

Algorithm for normalization of auxiliary names is presented on Algorithm 2.

Algorithm 1 alphaNormalize(Q)

for all non-leaf depth-first node N in query syntax tree with the root node Q **do**
  if $N.op \in$ {"$<$", "$\leq$", "$>$", "$\geq$"} **then**
    alphaNormalize($N.left$);
    alphaNormalize($N.right$);
    if text($N.right$) < text($N.left$) **then**
      $N.op \leftarrow$ ">", "$\geq$", "$<$", "$\leq$"; {respectively}
      swap($N.left$, $N.right$);
    end if
  else if $N.op \in$ {"$=$", "$\neq$"} **then**
    alphaNormalize($N.left$);
    alphaNormalize($N.right$);
    if text($N.right$) < text($N.left$) **then**
      swap($N.left$, $N.right$);
    end if
  else if $N.op =$ "struct" **then** {possibility of more than two child nodes}
    $queue \leftarrow$ empty alphabetically-sorted queue
    for all node $child \in N.childs$ **do**
      alphaNormalize($child$);
      $queue$.push(text($child$));
    end for
    while $queue \neq \oslash$ **do**
      $child \leftarrow queue$.pop();
      $N.childs$.push($child$);
    end while
  else if $N.op \in$ {"+", "$-$", "$*$", "/", "and", "or", "union", "intersect"} **then**
    $queue \leftarrow$ empty alphabetically-sorted queue
    alphaNormalize($N.right$);
    $queue$.push(text($N.right$));
    $L \leftarrow N.left$;
    while L is non-leaf **and** $L.op = N.op$ **do**
      alphaNormalize($L.right$);
      $queue$.push(text($L.right$));
      $L \leftarrow L.left$;
    end while
    alphaNormalize(L);
    if $L.parent.op \notin$ {"$-$", "/"} **then**
      $queue$.push(text(L));
      $L \leftarrow L.parent$;
      $L.left \leftarrow queue$.pop();
    end if
    while $queue \neq \oslash$ **do**
      $L.right \leftarrow queue$.pop();
      $L \leftarrow L.parent$;
    end while
  end if
end for

Algorithm 2 auxNormalize(Q)

$resultList \leftarrow$ empty sorted list;
$nameMapList \leftarrow$ empty sorted mapping list;
for all static binder $n(x) \in$ result signature of the query with the root node Q **do**
  $resultList$.push(n);
end for{the list remains empty for the above example query (without any binder in its result)}
$counter \leftarrow 1$;
for all non-leaf depth-first node N in query syntax tree with the root node Q **do**
  if $N.op \in$ {"as", "group as"} **then**
    if $N.name \notin resultList$ **then**
      $nameMapList$.push($N.name$, "$cache_aux" + text($counter$));
      $N.name \leftarrow$ "$cache_aux" + text($counter$);
      $counter \leftarrow counter + 1$;
    end if
  else if $N.op$ is name expression **then**
    if $N.name \in nameMapList$ **then**
      $N.name \leftarrow$
      $nameMapList$.getMappedValue($N.name$);
    end if
  end if
end for

VI. Query Decomposition and Rewriting

After normalization phase query is virtually decomposed, if possible, into one or many simpler candidate subqueries. *Query decomposition* is a useful mechanism to speed up evaluating a greater number of new queries. If we materialize a small independent subquery instead of a whole complex query, then the probability of reusing of its results is risen. In addition, a simple semantic of the decomposed query reduces the costs of its updating. Each isolated subquery and finally a whole query is independently analyzed in context of the set of cached queries defined in the query cache registry and if it hasn't yet cached, it becomes a new candidate for caching.

Too simple queries (without object names or non-algebraic operators) are omitted. While analyzing, query is converted to the text form and the optimizer performs search process using query index stored in the query cache registry. If found, the tree of the query is replaced with a call of a special *cache function* parameterized with unique references to nodes of matched cached query in the metabase and the object store (these MB_ID and DB_ID parameters are non-key elements of cached query index mentioned earlier). Each not yet cached candidate query is also replaced with a call of the cache function - new cached query is placed into the query index. In this case a query node in the object store doesn't contain query results - it is marked as "not fully cached" and will be populated with its results while the first need of use (when the interpreter will evaluate it). The analyzing algorithm is presented on Algorithm 3.

Algorithm 3 analyze($Q, resultType$)

if $resultType =$ FULL **then**
  $(MB\_ID, DB\_ID) \leftarrow$ searchCache(text(Q));
  if $MB\_ID \neq 0$ **then** {cached query found}
    $Q \leftarrow$ new tree with function call "$cache_fun($MB_ID, DB_ID$)";
  end if
else {PARTIAL}
  if $Q.op =$ "." **and** $Q.right.op \in$ {"sum", "avg", "min", "max", "count"} **then**
    if $Q.left.op$ is name expression **and** $Q.left.name$ is class object **then**
      $Q.op \leftarrow$ "join";
      $(MB\_ID, DB\_ID) \leftarrow$ searchCache(text(Q));
      if $MB\_ID \neq 0$ **then**
        $name \leftarrow Q.left.name$;
        $Q \leftarrow$ new tree with function call "$cache_fun($MB_ID, DB_ID, name$)";
      end if
    end if
  end if
end if
if Q is not a call of function "$cache_fun" **then** {cached query not found}
  $(MB\_ID, DB\_ID) \leftarrow$ insertCache(Q); {new cached query}
  $Q \leftarrow$ new tree with function call "$cache_fun($MB_ID, DB_ID$)";
end if
return Q

Factoring out independent subqueries: The concept of query idependence is thoroughly investigated in [3], [21], [22]. Instead of caching such a complex query as:

```
Emp where salary <
((Emp where name = "Smith").salary)
```

we isolate an internal independent query:

```
(Emp where name = "Smith").salary
```

and transform the whole query to the following form:

```
(((Emp where name = "Smith").salary)
group as v).(Emp where salary < v)
```

The independent query is matched and proposed as cached query uniquely identified by its node references MB_ID and DB_ID, and finally the original query is rewritten to:

```
Emp where salary <
$cache_fun(MB_ID, DB_ID)
```

Algorithm of the method is presented on Algorithm 4 (assuming the existence of a special function *isIndependent* checking the independence and if need changing the query in an appropriate manner).

Algorithm 4 independDecompose(Q)

if isIndependent(Q) **then** {query has been transformed and starts with dot operator}
  $left \leftarrow$ analyze($Q.left.left$, FULL); {analyzing idepen-dent query without auxiliary name}
  if $left.op$ is function call **and** $left.name =$ "$cache_fun" **then** {query cached}
    for all non-leaf depth-first node N in query syntax tree with the root node $Q.right$ **do**
      if $N.op$ is name expression **and** $N.name = Q.left.name$ **then** {auxiliary name}
        $N \leftarrow left$;
      end if
    end for
    $Q \leftarrow Q.right$;
  end if
end if
$Q \leftarrow$ analyze(Q, FULL);

Factoring out aggregations: Aggregating functions (`avg`, `min`, `max`, `sum`, `count`) are in many cases time consuming queries. Such functions can be interpreted as virtual materialized attributes of database objects of some classes, i.e. query:

```
Dept join avg(employs.Emp.salary)
```

is cached as a group of cached queries for each `Dept` object instance which becomes an additional third parameter of cache function `$cache_fun`. Thus another query:

```
Emp where salary >
works_in.Dept.avg(employs.Emp.salary)
```

is decomposed by isolating cached query:

```
Dept.avg(employs.Emp.salary)
```

and rewriting the whole query as follows (Fig. 3):

```
Emp where salary >
works_in.$cache_fun(MB_ID, DB_ID, Dept)
```

Algorithm of the method is presented on Algorithm 5.

Removing path expressions: Reference paths finalizing query evaluation are isolated, but only if they are quickly evaluable (thanks to referential nature of object-oriented database). If a query is finalized with a sequence of navigational operators (dot) or the constructor of a structure (`struct`) containing such sequences, and all the objects within such expressions are unique subobjects or reference objects (with cardinality 1 or 0..1), the longest expressions fulfilling this condition are cut forming simpler independent query for caching, i.e. query:

```
(Training where count(received_by) > 12).
(subject, duration,
supervised_by.Emp.salary)
```

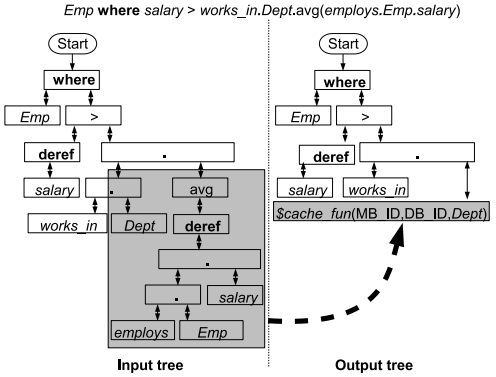

Fig. 3. Sample query optimization

Algorithm 5 aggDecompose(Q)

for all non-leaf depth-first node N in query syntax tree with the root node Q **do**
  if $N.op =$ "." **and** $N.right.op \in$ {"sum", "avg", "min", "max", "count"} **then**
    if $N.left.op$ is name expression **and** $N.left.name$ is class object **then**
      if $N = Q$ **then**
        $N \leftarrow$ analyze(N, FULL);
      else
        $N \leftarrow$ analyze(N, PARTIAL);
      end if
    else if $N.left.op =$ "." **then**
      if $N.left.right.op$ is name expression **and** $N.left.right.name$ is class object **then**
        $left \leftarrow N.left.left$;
        $N.left \leftarrow N.left.right$;
        $N.right \leftarrow$ analyze(N, PARTIAL);
        $N.left \leftarrow left$;
      end if
    end if
  end if
end for
$Q \leftarrow$ analyze(Q, FULL);

is ended with an implicit structure constructor with a path expression. Each expression has the cardinality 1, so all expressions (and in consequence structure constructor, too) are ignored while isolating the query:

```
(Training where count(received_by) > 12)
```

and finally rewriting the input query to:

```
$cache_fun(MB_ID, DB_ID).
(subject, duration,
supervised_by.Emp.salary)
```

In case of another query:

```
(Dept where dname = "Database").
employs.Emp.prev_job
```

Algorithm 6 pathDecompose(Q)

```
done ← false;
if Q.op = "." then
  if Q.right.op is name expression then
    if cardinality(Q.right.name) ≤ 1 then
      pathDecompose(Q.left);
      done ← true;
    else if Q.left.op is name expression
    and cardinality(Q.left.name) ≤ 1 then
      pathDecompose(Q.left);
      done ← true;
    end if
  else if Q.right.op = "struct" then
    proper ← true;
    for all node child ∈ Q.right.childs do
      if child.op is name expression then
        if cardinality(child.name) > 1 then
          proper ← false; break
        end if
      else if child.op = "." then
        pathDecompose(child);
        if child.op is name expression then
          if child.childs ≠ ∅
          or cardinality(child.name) > 1 then
            proper ← false; break
          end if
        else
          proper ← false; break
        end if
      else
        proper ← false; break
      end if
    end for
    if proper then
      pathDecompose(Q.left);
      done ← true;
    end if
  end if
end if
if not done then
  Q ← analyze(Q, FULL);
end if
```

both `prev_job` (subobject) and `employs` (reference object) attributes have cardinality $0..*$, so the optimal solution is to cache the whole query. Algorithm of the method is presented on Algorithm 6.

Transforming queries involving logical and set-based expressions: Thanks to the distributivity property of the selection operator in SBQL (`where`), it is possible to decompose queries with complex predicates containing some logical operators (`or`, `and`, `not`) into two or more simpler queries joined by set operators (`union`, `intersect`, `minus`) on bags of results. For instance, the complex query:

Algorithm 7 setDecompose(Q)

```
for all non-leaf depth-first node N in query syntax tree with
the root node Q do
  if N.op = "where" and N.right.op ∈ {"or", "and",
  "not"} then
    left ← empty tree;
    right ← empty tree;
    right.op ← "where";
    right.left ← N.left;
    if N.right.op ∈ {"or", "and"} then
      left.op ← "where";
      left.left ← N.left;
      left.right ← N.right.left;
      right.right ← N.right.right;
    else {"not"}
      left ← N.left;
      right.right ← N.right.left; {left is the only node}
    end if
    left ← analyze(left, FULL);
    right ← analyze(right, FULL);
    N.op ← "union", "intersect", "minus"; {respectively}
    N.left ← left;
    N.right ← right;
  end if
end for
Q ← analyze(Q, FULL);
```

```
Emp where (job = "clerk") or
(job = "consultant")
```

is transformed into query:

```
(Emp where job = "clerk") union
(Emp where job = "consultant")
```

and finally into:

```
$cache_fun(MB_ID1, DB_ID1) union
$cache_fun(MB_ID2, DB_ID2)
```

Algorithm of the method is presented on Algorithm 7.

VII. Experimental Results

We have tested the performance of the optimizer by calculating response times for 100 subsequent requests using a set of queries retrieving data from database containing over 100000 objects being instances of `Dept` or `Emp` class according to the schema presented in Fig. 1. Input queries with the same semantics were syntactically different but after the normalization or decomposition they became unified. We have compared four optimization strategies: without optimization (NoCache), caching in volatile memory (TMP), caching in persistent memory (DB) and mixed caching (TMP+DB). The results presented in Fig. 4 show that in case of the TMP strategy average response time is more than 10 times shorter than response without using of the cache. In many cases, especially for more complex queries (using multi-parameterized predicates or aggregations), responses were 100 times faster.

VIII. Conclusions and Future Work

We have presented an approach to optimization of query execution using caching of the results of previously answered queries. Our solution addresses the stack-based approach to object-oriented query languages. The cached queries method as a tool for optimization ensures short and scalable response time to any user request types. Proper structures and strategies for fast retrieval and high utilization of cached queries results have been proposed. We have presented the architecture of the query cache optimizer for optimal query selection and rewriting new queries with the use of cached results. Methods of query normalization were developed, with preservation of the original query semantics (enabling higher reuse of cached queries for semantically equivalent but syntactically different queries). Query decomposition of complex queries into smaller ones was presented. Some experimental results of the optimization were introduced that demonstrate the effectiveness of our method.

The work on cached queries is continued. There are many open research areas concerning this optimization method. The main areas concern some additional features of SBA and SBQL not mentioned in this paper, such as inheritance and dynamic object roles. Another open issue is recognizing some parts of the cached results helpful for answering other queries and combining many cached queries while producing a result of one wider query. In general, the problem is practical rather than theoretical, hence much effort should be devoted to experiments with different strategies of caching queries and keeping in sync their stored results.

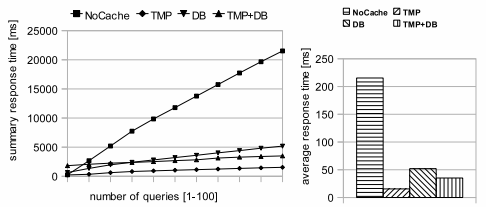

Fig. 4. Efficiency of optimization using cached queries

References

[1] "ODRA (Object Database for Rapid Application development), Description and programmer manual." http://sbql.pl/various/ODRA/ODRA_manual.html.

[2] P. Cybula, H. Kozankiewicz, K. Stencel, and K. Subieta, "Optimization of distributed queries in grid via caching," in *Proceedings of the On the Move to Meaningful Internet Systems 2005, OTM GADA Workshop*, vol. 3762 of *LNCS*, pp. 387–396, Springer, 2005.

[3] 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.

[4] 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.

[5] J. A. Blakeley, P. Larson, and W. Tompa, "Efficiently updating materialized views," in *Proc. of ACM SIGMOD*, pp. 61–71, 1986.

[6] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing queries with materialized views," in *Proc. of Intl. Conf. on Data Engineering*, pp. 190–200, 1995.

[7] C. M. Chen and N. Roussopoulos, "The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching," in *Proc. of Intl. Conf. On Extending Database Technology*, 1994.

[8] "IBM DB2 Universal Database SQL Reference." http://www.ibm.com/software/data/db2/udb, Vol. 2, Version 8, 2002.

[9] "Faster federated queries with MQTs." http://www.db2mag.com/db_area/archives/2003/q3, DB2 Magazine, Vol. 8, No. 3, 2003.

[10] "Oracle 9i materialized views, An Oracle White Paper." http://www.oracle.com/database, May 2001.

[11] "On Oracle Database 11g," Oracle Magazine, Vol. XXI, No. 5, 2007.

[12] M. A. Ali, A. A. A. Fernandes, and N. Paton, "MOVIE: An incremental maintenance system for materialized object views," in *Proc. of Data and Knowledge Engineering*, vol. 47, pp. 131–166, 2003.

[13] A. Kemper and G. Moerkotte, "Access support in object bases," in *Proc. of ACM SIGMOD*, pp. 364–376, 1990.

[14] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan, "Semantic data caching and replacement," in *Proc. of VLDB*, 1996.

[15] 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.

[16] L. Chen and E. A. Rundensteiner, "ACE-XQ: A CachE-ware XQuery Answering System," in *Proc. of WebDB*, pp. 31–36, 2002.

[17] M. El-Sayed, L. Wang, L. Ding, and E. A. Rundensteiner, "An algebraic approach for incremental maintenance of materialized XQuery views," in *Proc. of WIDM*, 2002.

[18] "MySQL 5.4 reference manual, chapter 7.5.5: The MySQL query cache." http://www.mysql.com, 2009.

[19] "LINQ: .NET Language-Integrated Query," http://msdn.microsoft.com/pl-pl/library/bb308959(en-us).aspx, Microsoft Corporation, 2007.

[20] R. G. G. Cattell, and D. K. Barry (eds.), "The Object Data Standard: ODMG 3.0," Morgan Kaufmann, 2000.

[21] K. Subieta, "Theory and practice of object query languages," Polish-Japanese Institute of Information Technology, 2004. In Polish.

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

[23] 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.