IMPLEMENTATION OF ALGEBRA FOR QUERYING WEB DATA SOURCES

Iztok Savnik
Faculty of Mathematics, Science and Information Systems, University of Primorska, Koper, Slovenia
iztok.savnik@famnit.upr.si

Keywords: Database algebras, query languages, distributed databases, query optimization, Web query languages.

Abstract: The paper presents the implementation of query execution system Qios. It serves as a lightweight system for the manipulation of XML data. Qios employs the relational technology for query processing. The main aim in the implementation is to provide a querying system that is easy to use and does not require any additional knowledge about the internal representation of data. The system provides robust and simple solutions for many design problems. We aimed to simplify the internal structures of query processors rooted in the design of relational and object-relational query processors. We propose efficient internal data structures for the representation of queries during all phases of query execution. The query optimization is based on dynamic programming and uses beam search to reduce the time complexity. The data structure for storing queries provides efficient representation of queries during the optimization process and the simple means to explore plan caching. Finally, main memory indices can be created on-the-fly to support the evaluation of queries.

1 Introduction

The Internet contains large amount of different data sources accessible through ftp files, XML or HTML documents, and the wrappers around relational and object-relational database systems. The data available via such data sources may vary from simple lists of records, catalogs containing large amounts of flat tables, to complex data repositories including large conceptual schemata and tables composed of complex objects (S. Abiteboul, 1993; M.A. Roth, 1988). Querying Web data sources poses several new problems such as uniform access to the data sources, integration of information provided by the data sources, and efficient manipulation of large data sets.

Our work focuses on the design of robust and flexible query execution system for querying and integration of data from Web data sources. The design of query execution system `Qios`(Savnik, 2007) is based on the existing work on relational and object-relational query execution systems (Graefe, 1993; M. Jarke, 1984; D. Daniels, 1982). The kernel of algebra comprises relational operations extended for the manipulation of complex objects. Further, the algebra includes a set of operations for querying conceptual schemata. These operations allow for browsing the conceptual representation of data sources and using the elements of conceptual schemata for querying extensional data (I. Savnik, 1999).

We investigated the efficient and robust design of the internal structures of the query execution engine. In order to simplify the internal structure of system a single data structure is used for the representation of queries during all phases of query execution. The algorithm used for the optimization is based on graph representation of query trees and query transformation rules. Query transformation is seen as matching rule input tree against a query tree and than duplicating the rule output tree. Similarly, query evaluation module uses the same structure for the implementation of scans. Finally, to provide efficient implementation of query optimization and evaluation queries are stored in a data structure called Mesh (G. Graefe, 1987). The data structure is refined to provide efficient access to the stored queries.

The query optimization algorithm is based on a version of dynamic programming called *memoisa-*

tion. The most promising results were obtained with optimization algorithm which uses beam search for the exploration of the hypothesis space of equivalent queries. Further, we explore plan caching (Graefe, 2005; Marathe, 2006) which speeds up significantly the subsequent execution of queries issued on the same domain.

The query evaluation is based on dynamic selection of the query evaluation plans. We use a simple strategy that exploits the large quantity of main memory which is lately provided by almost any personal computer. The main memory indices can be constructed on-the-fly to support query evaluation. The construction of indices is based on standard index selection rules available from any database textbook. For instance, hash-based index is used whenever a larger tables has to be joined. In this way we achieve fast performance of query processor for relatively large quantities of data. Furthermore, the user does not need to concern about the details of query evaluation process.

The paper is organized as follows. The following section presents the data model used for the representation of data from Web data sources. Section 3 introduces the basic operations of algebra and presents the examples of queries. The main portion of paper presents the implementation of algebra. Section 4 describes storage manager, parser, query representation, query optimization and evaluation. Section 5 overviews the empirical results of query execution in `Qios`. Finally, concluding remarks are stated in Section 6.

2 Algebra

The data model used for the representation of data stored at different types of data sources must meet the following requirements. First, the data sources provide various types of data including semi-structured data, XML data, (flat) relational tables, and objects represented by object-relational database models. Third, we expect that besides the extensional data, the data sources will include large amounts of intensional data describing the structure and the contents of the extensional databases.

The F-Logic data model was used as the formal basis of the system (M. Kifer, 1995): it was shown (I. Savnik, 1999) that it can serve as the basis for the representation of the semi-structured data, it provides a convenient representation of the relational and the object-relational database models, it can be used for the representation of complex objects, and, it can be used very naturally to represent intensional data.

The operations for inquiring about the basic properties of objects which relate to the representation of objects are called *model operations*. Besides the standard comparison operations $=, >, >=, <, <=$, the set operations $\in$ and $\subseteq$, and the component selector operator ".", which are defined in relational algebra, the algebra includes the following model operations. The operations `ext` and `exts` map class objects to the sets of theirs members, or the set of their instances, respectively. Next, the operation `class_of` allows for the mapping of the ground objects to their parent classes[1]. Further, the poset comparison operations $\prec_o, \preceq_o, \succ_o$ and $\succeq_o$ are used to relate objects with respect to the partial ordering relationship defined among objects. The operations `subcl` and `supcl` map class objects to a set of their subclasses or super-classes, respectively. Finally, the operation `=~` is defined for searching the text using regular expressions. The operation `=~` is defined as in the `Perl` programming language. The *declarative operations* of the algebra are used for the manipulation of the sets of objects. The detailed presentation of the model and declarative operations can be found in (I. Savnik, 1999). The following groups of declarative operations are defined in the algebra.

Relational operations. These operations include standard relational operations `select`, `project`, `union`, `differ` and `join` which are extended for the manipulation of objects.

Nested-relational operations. The operation `group(s,a,b)` is defined similarly to the SQL *group-by* construct. It groups objects from `s` by the values of attributes from the set `a`. The values of the attributes which are not in `a` are stored as the relation which is the value of the attribute `b`. The operation `unnest(s,a)` is used to unnest a set valued attribute `a` of objects from the argument set `s`.

Object-restructuring operations. The operation `collapse(s,a)` collapses the tuple structured attribute `a` nested in the objects from the set `s`. The operation `flatten(s)` is used for collapsing the set of sets `s`.

Operation `apply` (P. Buneman, 1979). The functional operation `apply`(s, f) is used for the application of a query expression `f` to a set of objects `s`. This operation is useful for the application of a query to the sets of objects that can be located at different sites.

3 Query Execution System

`Qios` (v0.9) is a system for the manipulation of

[1]A ground object can have a single parent class.

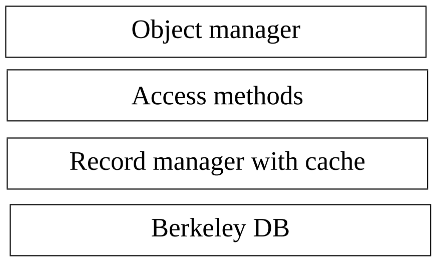

Figure 1: Storage manager

data from internet data sources. The system is intended to serve as the lightweight kernel of a data manipulation server. The main aims in the design of `Qios` are to provide: capabilities to manipulate collections of data in a fast manner, various data manipulation functions from classical querying to data restructuring, and, the capabilities to organize, store and browse the data collections obtained from the Internet data sources on the local host. The system currently provides the interface for XML. The treatment of other data formats requires the addition of the interface routines for the conversion of data into internal database format.

The query execution system `Qios` is composed of the following components: the storage manager, the parser, the subsystem for query optimization, the subsystem for Web access, and the query evaluation subsystem. In this section we overview the implementation of the `Qios` components. Not all operations of the presented object algebra (I. Savnik, 1999) are implemented in the current version of system. The operations that are included are: model-based operations, the operations `select`, `project`, and `join`.

Record manager. The record storage manager is based on the Berkeley DB storage system providing access to different storage structures such as for instance hash-based index or B+ tree. Record manager implements a data store for records representing individual and class objects. Records are treated as arrays of bytes, the structure of which is known at the object level. Each record has a record identifier (abbr. rid) implemented as system generated identifier which is used as the key for the hash-based index in Berkeley DB. Therefore, records are stored as oid/value pairs where the values are packed in the sequences of bytes. Record manager includes the methods for the work with object identifiers, routines for reading and writing records, methods for the realization of the hierarchical database model, and access to main memory indices that are associated to the class object.

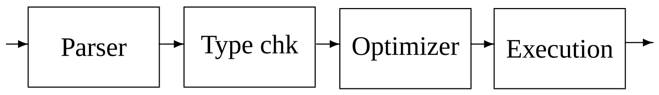

Figure 2: Query processing

Object manager. The object manager serves as a cache of objects loaded from Web as well as for storing intermediate results during query processing. `Qios` persistent objects are objects that are tied to the database via object manager. Each object has an identifier which is implemented by means of record identifier from the subordinate level. External identifiers which are unique within the datafile can be assigned to objects when they are created. Object cache is realized using LRU (least recently used) strategy for the selection of objects to be removed from cache. The size of object cache can be set as the system parameter via the configuration record of a datafile as well as at run-time.

Parser. This module includes the implementation of a parser for the algebra expressions. The algebra expressions are checked for syntactical errors and then translated into query trees.

The basic skeleton of the query tree is constructed during the parsing process. The variables and the names of the data sources and spans are stored in symbol table `symtab`. The query nodes represent the operations, spans linking rules, and access methods. The data for the different phases of query processing is stored in the same tree nodes. Each particular module (e.g. query optimization) manipulates its own view of query nodes.

Type-checking is implemented by procedure computing the types of query nodes bottom-up, that is, from access methods toward the root of query tree. The class objects related to the access methods are retrieved during the parsing process. The type of a query node is stored by constructing a new class object.

Query optimizer. The query optimization subsystem is composed of the following main modules. The query tree manipulation module includes routines for manipulation of query trees, application of rules on the nodes of query trees, and property functions by which the physical and logical properties of the query tree nodes are determined.

The cost function is realized by a separate module. The extensions of the cost functions used for the relational query optimizer are defined. The optimization

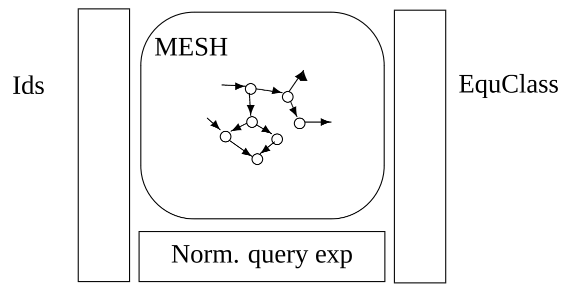

Figure 3: Mash with access paths

module comprises the algorithms for the optimization of query expressions. The algorithms which are studied are the exhaustive search and an algorithm based on dynamic programming.

The core of the module is the data structure `mesh` for the representation of sets of queries. Common sub-expressions of queries are shared ie. each query is represented in `mesh` only once. Queries are organized into equivalence classes. Let us first present the data structure Mesh.

`mesh` stores query expressions as query trees organized as a directed acyclic graph (dag). The query trees share common parts hence there is only one representation of a query expression in `mesh`. Further, the query trees are organized into equivalence classes including logically equivalent queries. `mesh` has three entry points.

The algorithm for query optimization is based on dynamic programming. We use the variant of dynamic programming algorithm called *memoisation* which stores the optimal results of the sub-queries and uses them in the computation of the composed queries. A simple brute force enumeration of all reasonable compositions of operations from a given query to provide the basis for classical dynamic programming would be computationally too inefficient. The enumeration of equivalent queries of a given query have to be guided by some form - in our case, the compositional structure of query.

Memoisation is realized in a very simple manner. Every time a query is to be optimized we first check Mesh if the optimal query already exists for a given equivalence class of input query. At this point we can see the use of Mesh for *plan caching*. If we do not clean Mesh after the execution of a given query that the optimization of the subsequent queries can make use of the existing optimized queries in Mesh. The optimization time is reduced significantly.

The *query transformation rules* are used for the transformation of query trees into logically equivalent query trees that have different structure and potentially a faster evaluation method. The logical optimization rules are in Qios specified in a language that follows strictly the syntax and the semantics of the Qios query expressions.

Finally, the cost function module is used to estimate the number of bytes processed by the physical query tree including the cost of materialization of the intermediate query evaluation results. The computation of the cost estimation for the operations select, project and join is realized using standard relational formulas based on the selectivity of attributes and presupposing the normal distribution of attributes values.

The computation of statistics for classes is generated dynamically, triggered by query compilation procedure when classes are loaded into the main memory. The following data is gathered for a given class: the number of class objects, the size of objects and the cardinality of all attributes.

Query evaluation system. The query evaluation module is based on the iterator-tree representation of the query evaluation plans. The physical query execution plan is computed from the optimized query trees by adding to the existing query nodes information about physical operation that will implement given logical operation (query node). The query nodes already contain information about the statistics, index selection, and cost estimation.

The main strategy which was used in the implementation of query execution is to select reasonably fast access methods on-the-fly without considering alternatives. Simple rules are used for index selection. Firstly, if selection or join is based on equality of attributes than hash-based index is generated. Secondly, if selection or join involves range predicate we use B-tree index. The selection of query execution plan is implemented by the procedure which computes physical operations for all logical operations (query nodes) forming the physical query tree in a bottom-up manner.

XML loader. XML loader provides the interface to the Web. It can read XML data from local files as well as the internet data sources. The `XML` files are parsed and translated into the internal representation using a parser SAX. Module includes the procedures for loading XML data, conversion of the DTD schemata into internal database schemata, and discovery of database schemata from XML data.

4 Empirical results

This section presents the initial experimental results. A simple artificially generated database is used for the experiments. We constructed eight tables that include three attributes: the first two attributes are of type integer, and the last one is a string. Each table includes 10000 randomly generated records: the first two attributes are set to random integer number in the range of 0 to 5000. String includes 300 characters.

The queries used in the experiments form a chain of joins of length N. Each query is restricted by a selection to force small number of query output records. The following example presents test query that forms a chain of 5 classes.

Example 4.1 *The query in this example joins classes using the attributes $p1$ and $p2$. Note that the expressions of the form $r :: p$ represents the name of attribute p defined for a relation r. The condition $u.r1 :: p1 < 5$ restricts the output to small number of records.*

```
select( join(r1.ext,
     join(r2.ext,
          join(r3.ext,
               join(r4.ext,r5.ext,
                    i,j: i.r4::p2 == j.r5::p1 ),
               x,y: x.r3::p2 == y.r4::p1 ),
          s,d: s.r2::p2 == d.r3::p1 ),
     z,w: z.r1::p2 == w.r2::p1 ),
u: u.r1::p1 < 5 )
```

The empirical results are presented in Figure 4. We observe two separate executions of the same query. The first is executed after the system is started; in this case the statistics and the indices are not created initially. The second execution is performed after the first one is completed so most indices are already created. We observe the following parameters: number of joins (⋈), the execution of query after system startup (R=1) or later (R=2), time needed for the compilation of query including computation of statistics (Comp), number of classes for which the statistics was computed (S), optimization (Opt), evaluation of query including the creation of indices and console output (Eval), number of created indices (Inx), and, finally, the amount of memory used by the system (Mem). Time is specified in milliseconds. The experiments were done on Pentium 4 computer with 500MB RAM running FreeBSD.

Let us now give some comments on empirical results presented in Figure 4. Firstly, it is obvious that the creation of indices for larger tables can significantly slow down the overall performance. This pays off through the efficient evaluation of queries. Indices are usually constructed gradually: the overall computation is performed for sequence of queries where the previous queries may trigger the construction of indices used by the subsequent queries. Least recently used indices are dropped if there is no more room in the main memory for a new index. This did not happen during the experiments presented in Figure 4. The amount of memory used during query execution never exceeds 87M which is acceptable for practically all recent personal computers.

⋈	R	Comp	S	Opt	Eval	Inx	Mem
1	1	12768	2	31	7440	1	25M
1	2	11	0	35	6358	0	25M
2	1	19439	3	129	8627	2	34M
2	2	17	0	149	6397	0	34M
3	1	26650	4	441	1682	3	42M
3	2	23	0	491	6991	0	42M
4	1	37857	5	739	13121	4	52M
4	2	30	0	731	7315	0	52M
5	1	42780	6	957	13077	5	62M
5	2	33	0	925	7295	0	62M
6	1	51541	7	1277	14296	6	72M
6	2	40	0	1194	7631	0	72M
7	1	64389	8	1830	16539	7	87M
7	2	46	0	1982	7473	0	87M

Figure 4: Query execution

The optimization phase is relatively fast comparing to computation of statistics and index creation. This is achieved primarily by using beam search algorithm that restricts the number of choices during the search to N queries for each equivalence class. In the case of exhaustive search the exponential time curve turns very steep after 5 joins. In the presented experiments we have used the window (beam) of 7 queries in each equivalence class.

5 Related work

All recent database algebras have evolved from the relational database algebra proposed by Codd in (Codd, 1970). Although some algebras do not include the operations of relational algebra directly, each of them is relationally complete i.e. it includes the ability to compute selection, standard set operations, projection and Cartesian product.

Our work on object algebra has been influenced by the early functional query language FQL proposed by Buneman and Frankel (P. Buneman, 1979) and by some of its descendants, for instance, the functional database programming language FAD (S. Danforth, 1992). The algebra is by its nature a functional language, where the operations can be combined by the use of functional composition and higher-order functions to form the language expressions. The presented object algebra can be treated as a generalization of FQL for the manipulation of objects. It subsumes the operations of FQL, i.e. operations extension, restriction, selection and composition.

Let us now present the work related to the implementation of the presented query execution system. Firstly, the implementation is closely related to the implementation of Query Algebra originally proposed by Shaw and Zdonik in (G.M. Shaw, 1990) and implemented by Mitchell (Mitchell, 1993). In particular, we have used a similar representation of query expressions by means of query trees. Furthermore, the representation of query expressions in Qios is optimized by using single operation nodes and query trees during all phases of query processing.

The design of the query execution system was based on the design of Exodus optimizer generator (G. Graefe, 1987) and its descendant Volcano (G. Graefe, 1993). The data structure MESH used in Exodus query optimizator generator is improved by adding additional access paths. The data structure can be accessed through: unique identifier, normalized query expression, and equivalence class. The algorithm for query optimization is rooted in Graefe's work on Volcano optimizer algorithm (G. Graefe, 1993). This algorithm uses top-down search guided by the possible "moves" that are associated to a query node. The algorithm uses memoisation to avoid repeated optimization of the same query. The search is restricted by the cost limit which is a parameter in optimization.

We intended to implement a lightweight system able to manipulate middle size XML databases including up to some 100.000 records. This size is reasonable for most of collections appearing on the Web as well as in our local data environments. One of the aims in the design of Qios was to exploit this advantage. The design of Qios has the following salient features. A single data structure is used during the complete optimization and evaluation process. The central data structure of query optimization Mesh is robust and simple to use. Rules are treated as query trees and rule matching and application procedures are based on graph (tree) algorithms.

The computation of query execution plan that uses indices created during query evaluation has similar objectives: it is expected that XML database will be of the above stated size and that the user is not informed about the existence of indices and the selected access paths during the execution. All decisions about the creation of main memory indices are done by the system.

6 Conclusions

The paper presents the implementation of an object algebra. The data model based on F-Logic provides a convenient environment for the representation of semi-structured as well as structured data. Algebra includes standard operations on sets which evolved from the relational and nested-relational algebras and the operations for querying the conceptual schemata. Object algebra is implemented in the query execution system `Qios` which is rooted in the architecture of the relational and object-relational query processors.

REFERENCES

Codd, E. (1970). A relational model of data for large shared data banks. In *Comm. of the ACM, Vol. 13 , Issue 6*. ACM.

D. Daniels, E. (1982). An introduction to distributed query compilation in r*. In *IBM Research Report RJ3497 (41354)*. IBM.

G. Graefe, D. D. (1987). The exodus optimizer generator. In *Proceedings of ACM SIGMOD international conference on Management of data*. ACM.

G. Graefe, W. M. (1993). The volcano optimizer generator: extensibility and efficient search. In *Proc. of IEEE Conf. on Data Engineering*. IEEE.

G.M. Shaw, S. Z. (1990). A query algebra for object oriented databases. In *Proc. of IEEE Conf. on Data Engineering*. IEEE.

Graefe, G. (1993). Query evaluation techniques for large databases. In *Comp. Surveys, Vol.25, No.2*. ACM.

Graefe, G. (2005). Query processing. In *Slides, ICDE Influential Paper Award*. ICDE.

I. Savnik, Z. Tari, T. M. (1999). Qal: A query algebra of complex objects. In *Data & Knowledge Eng. Journal, Vol.30, No.1*. North-Holland.

M. Jarke, J. K. (1984). Query optimization in database systems. In *Comp. Surveys, Vol.16, No.2*. ACM.

M. Kifer, G. Lausen, J. W. (1995). Logical foundations of object-oriented and frame-based languages. In *Journal of ACM, Vol.42, No.4*. ACM.

M.A. Roth, H.F. Korth, A. S. (1988). Extended algebra and calculus for non 1nf relational databases. In *Trans. Database Systems, Vol.13, No.4*. ACM.

Marathe, A. (2006). Batch compilation, recompilation, and plan caching issues in sql server 2005. In *Microsoft, White paper*. Microsoft.

Mitchell, G. (1993). Extensible query processing in an object-oriented database. In *Ph.D. thesis*. Brown University.

P. Buneman, R. F. (1979). Fql- a functional query language. In *Proc. of the ACM Conf. on Management of Data*. ACM SIGMOD.

S. Abiteboul, C. B. (1993). On the power of the languages for the manipulation of complex objects. In *Verso Report No.4*. INRIA.

S. Danforth, P. V. (1992). A fad for data intensive applications. In *Trans. on Know. and Data Eng., Vol.4, No.1*. IEEE.

Savnik, I. (2007). Qios: Querying and integration of internet data. In *http://www.famnit.upr.si/˜savnik/qios/*. FAMNIT.