QUERY OPTIMIZATION BY STORED QUERIES

K. Subieta and W. Rzeczkowski

Institute of Computer Science, Polish Academy of Sciences
P.O.Box 22, 00-901 Warsaw PKiN, Poland

ABSTRACT

A stored query is a pair <query, response>, where "response" is the query meaning for the current database state. When a collection of stored queries is available responses to some queries may be obtained easily. Stored queries give a possibility of improvement of database system response time regardless of the complexity of user request and the data model assumed. The method is a generalization of methods based on indices. Its main properties and problems are outlined, particularly the problem of updating stored queries. The presented solutions are based on detecting whether the response associated with a query is influenced by a database update, and on correcting the response after an update. The methods concern NETUL, a user-friendly query language, with the power of programming languages, for network/semantic data models.

1. INTRODUCTION

Practical implementation of attractive query language features (such as non-procedurality, lack of concepts related to physical data, high universality, and integration with updating capabilities) meets considerable performance difficulties. This problem, commonly known as query optimization, arose mainly in connection with the relational data model (RDM) which was criticised for poor efficiency of proposed user interfaces. The research on query optimization has improved the position of RDM in this respect. (The current state of art of query optimization may be found in [19,26] together with comprehensive bibliography). Nevertheless the results are not satisfactory [16,39]. Many queries remain beyond the scope of developed methods or cannot be noticeably optimized, eg. queries with aggregates, with inequality conditions, queries involving arithmetics in conditional clauses, queries implying many joins, and so on.

The research on query optimization is typically based on the relational algebra, thus implicitly assumes that semantics of query-manipulation languages can be consistently expressed via the relational algebra and that it covers typical user's needs. Both assumptions are not true in practice. Many vital aspects of query languages cannot be consistently explained via the relational algebra or relational calculi [36], despite many attempts. Similarly, queries expressible in the relational algebra are thought to be typical since current relational query languages essentially cannot offer more. Usually their power is restricted to relational completeness augmented (with limitations) by arithmetics and aggregate functions. Relational completeness, however, is an ill-motivated concept of query language universality. If a query language had higher power, such queries might become far less typical. This is our observation concerning NETUL [34-37], a user-friendly query language with the power of algorithmic programming languages.

Our further objections to RDM concern its intellectual limitations. Majority of database applications are still non-relational, not only for the reason of performance problems. A major drawback of RDM data structures is that normalized relations cannot reflect the conceptual structure of the modelled environment. Also, NETUL has shown that RDM has no advantages with respect to user interfaces, which are an argument in favour of semantic data models such as E-R Model [6,35]. In RDM relationships among entities are not explicitly...
There are few papers concerning query optimization addressing network data structures [10, 11, 16]. The motivation for it is essentially different from that for RDM. The source of performance difficulties in relational queries is the join operator (or equivalently, the Cartesian product) which is used mainly for navigation through relations according to primary and foreign keys. In network databases this function is fulfilled by explicit pointer links, thus the presence of joins in query languages for network data structures may be questioned. NETUL, being more powerful than current relational query languages, has no operator resembling the join at all. Since in network databases joins may be replaced by navigation via links, the performance is much improved; nevertheless there are queries causing difficulties. Current optimization methods, as a rule, are unapplicable to such queries.

In this paper we deal with non-relational (network) data structures and consider query languages having the power of algorithmic programming languages. We propose an optimization method based on stored queries. A stored query is a pair <query, response>, where "query" is the query meaning evaluated for the current database state. When a collection of stored queries is available in the database system, responses to some queries can be obtained easily.

Two basic problems should be solved before such an idea can be put to practical use. These are:

- The size of a stored queries file. Usually query languages contain too many (typically, an infinite number) of different queries; thus not all can be stored.
- The updating problem. After a database update the responses to some stored queries may no longer be valid. Additional effort is required to put them in working order.

At first sight these problems may cause the idea to seem unrealistic. We show that this impression is wrong. Methods which are particular cases of this idea are well known; for example, methods based on indices. Indeed, each index item may be considered a stored query, i.e. a pair <query, response>, where "query" is an indexing term, and "response" contains references (pointers) associated with the term. (Unlike RDM, typical NETUL queries may return references.) Indices imply exactly the same problems as above [7, 29]; nevertheless, they are widely used. Indices do not work for many queries (eg. queries involving arithmetics, aggregates, inequality, and so on). A method based on stored queries has (theoretically) no such limitations.

Although the idea of stored queries seems to be new (at least in the mathematical formulation based on the denotational semantics, used in this paper), there are many related papers. A similar idea is presented in [12] where "stored subqueries" are considered to be a tool for global optimization of a collection of relational queries. Several papers are devoted to the problem of updating materialized (concrete) views or snapshots, eg. [3, 4, 22, 24, 31], which corresponds to the problem of updating stored queries. These papers are limited to particular relational operators. Thus the results presented can hardly be generalized for richer languages and network data structures. Some papers concerning optimization methods for RDM are also partly relevant, for example, papers dealing with equivalence among query language expressions, eg. [1], and papers on decomposition of queries into more elementary components, eg. [38]. These problems are significant in the context of the stored queries file size. Another relevant topic, considered in [28], concerns derivability of the meaning of a query from other (stored) queries.

In Section 2 we give a short introduction to NETUL and to related concepts. (A comprehensive description of NETUL and its theory based on denotational semantics may be found in [34-37].) Basic observations concerning the stored queries idea are presented in Section 3. Further sections are devoted to the most vital updating problem. At least four methods are possible:

a) Evaluating a new response for a query; in view of poor efficiency the method may be used for very frequent queries only.

b) Removing a stored query from the file: applicable to difficult to update queries, or not promising to be profitable in the future.

c) Detecting whether the response associated with a given query can be influenced by the database update, Section 4.

d) Correcting the meaning of a query instead of
its full evaluation. The method determines the part of the response which becomes invalid after a database update, and then determines the part of a database (as small as possible) for which the query must be re-evaluated in order to replace the invalid part of the response, Section 5.

2. INTRODUCTION TO NETUL

Currently several formal approaches to network (semantic, functional) databases is known, cf. [8,17,25,401]. These ideas cover basic features, such as entities, attributes and relationships; however many other aspects such as duplicates, ordering, repeating attributes, updating, aggregations, generalization, roles, overlapping entity sets, and so on, have no formal counterparts. They require additional or modified formal concepts, which will make the formalization complex and non-homogeneous. The approach presented below (see also [33-37]) is more general and provides a consistent mapping of all these concepts.

The basic concept is called "address". Addresses are abstract objects; we are not interested in their physical nature. They are denotations of some locations in the data storing medium. Let A be the set of addresses, N the set of data names, and V the set of atomic values; A \( \cap \) V = \( \emptyset \). A database content is a relation con \( \subseteq A \times N \times A \times A \times N \times V \). Triples \( <a_1,n,a_2> \) and \( <a_1,n,v> \) we interpret as "at address a1 there is a datum with name n and with a pointer to data at address a2" or "at address a1 there is a datum with name n and with a value v". Thus, a database content is a directed graph with addresses as nodes and values as leaves; edges of this graph are labelled by data names. Some addresses of the graph are understood as its starting points, thus we introduce the set fov \( \subseteq A \) called the field of vision. A database instance is viewed as a pair \( <fov, con> \). (In [35,36] a database instance includes a mathematical object reflecting data ordering. For simplicity, here ordering is not considered.) An address can "store" more than one pointer (Fig.2), and more than one name may be associated with it. We assume that an address can store at most a single atomic value, that is, \( <a_1,n_1,v_1> \in con \) and \( <a_1,n_2,v_2> \in con \) implies \( v_1 = v_2 \). Other obvious constraints concerning allowed database instances are presented in [36].

This general definition of a database instance allows us, through further constraints and rules, to map database instances of most of the current data models. In [36] we show how to map relational and CODASYL databases, and in [35] we show how to map advanced modelling primitives of the entity-relationship model. We have also no difficulties mapping database instances induced by functional models [5,30,401], binary relational data models [13], and many others. The points which distinguish our approach are the following.

- The database instance is viewed as a graph with addresses as nodes. Important database aspects, such as ordering and updating, cannot be consistently explained without this (or equivalent) concept [36,37]. These aspects can hardly be considered "non-conceptual"; hence the address concept in our approach.

- The most popular data models define fixed frames for the main logical data structures, eg. tuples in RDM, or records (entities) in network models. Such record-orientation was criticised in [18]. Our formalism allows to construct a variety of heterogeneous database structures. For example, it allows to model PASCAL-like records with variants, arbitrarily nested repeating groups, aggregates, optional data, etc.

- In many formal approaches there is unclear attitude to data names. Some of them denote sets, relations, or functions; thus from the mathematical viewpoint they are elements of the metalanguage. Instantaneous data bases do not contain them. Thus, in the denotational definition of a query language additional formal concepts, such as aggregations, "is-a" relations, roles, overlapping entity sets, and so on, which usually require a specific use of data names. In our approach we treat all names in a similar way and shift them to the level of instantaneous databases.

Example 1

Fig. 1 presents a diagram for a sample CODASYL database. It consists of DEPT and EMPS set type from DEPT to EMPS. The DEPT record type has a multivalued attribute LOC. Suppose that a database consists of two DEPT records (Sales,(Rome,Paris)), (Service,(Rome,Tokyo,London)), five EMPS records (Brown,1000), (Casey,1200), (Jones,1000), (Lewis,1500), (Smith,1400), and two occurrences of EMPS: one owned by the record for Sales with the member records for Brown, Casey and Smith, and the other owned by the record for Service with member records for Jones and Lewis.
This database instance is expressed as a pair \( \langle \text{fov}, \text{con} \rangle \), according to the principles of mapping CODASYL databases presented in [36]. A DBTG-set is represented as a collection of links, where a link is a pointer-valued attribute. For a set of type \( S \) the owner contains \( n \) such links (where \( n \) is the number of members), each named \( S \), linking the owner with the members, and each member contains a single link, named \( !S \), linking the member with the owner. The field of vision contains addresses of all records. For example, the record for Sales department may be represented as follows:

\[
\langle a1, \text{DEPT}, a11 \rangle, \langle a11, \text{DNAME}, \text{"Sales"} \rangle, \\
\langle a1, \text{DEPT}, a12 \rangle, \langle a12, \text{LOC}, \text{"Rome"} \rangle, \\
\langle a1, \text{DEPT}, a13 \rangle, \langle a13, \text{LOC}, \text{"Paris"} \rangle, \\
\langle a1, \text{DEPT}, a14 \rangle, \langle a14, \text{EMPS}, a3 \rangle, \\
\langle a1, \text{DEPT}, a15 \rangle, \langle a15, \text{EMPS}, a4 \rangle, \\
\langle a1, \text{DEPT}, a16 \rangle, \langle a16, \text{EMPS}, a5 \rangle
\]

The corresponding graph representation of the whole database instance \( \langle \text{fov}, \text{con} \rangle \), including the above record for Sales, is shown in Fig.2; \( a1, a2, a3, a4, a5, a6, a7 \) belong to fov.

NETUL queries are subdivided into joins and predicates. A join returns an \( n \)-column table of arbitrary length, while a predicate returns a boolean value. NETUL tables differ from RDM relations in a few essential points:

- Tables are sequences, thus the order of tuples may be significant.
- Duplicate tuples are allowed.
- Columns of a table are unnamed.
- Elements of tables are addresses or values.
- An element may be a pair \( (m, x) \), where \( m \) is an auxiliary name and \( x \) is a value or address.

The meaning of a NETUL query is understood as a function from the set of states into the set of tables (for joins) or into the set \{true,false\} (for predicates). Each state has two components: the database instance, as defined previously, and the stack ("field of vision stack") which is used for determining the meaning of data names nested in queries. The stack plays a vital part in the definition of basic NETUL constructs, such as selection (operator "where"), projection (operator "."), operators "with", operator "order by", quantifiers (operators "for any ... holds" and "for some ... holds") and transitive closure (operator "closed by"). Besides, there are typical operators, such as comparisons, arithmetic operators and functions, aggregate functions, and so on. NETUL allows arbitrarily nested queries with the only limitation concerning semantic types of nested subqueries.

Examples

Q1. Give employees earning less than 1200.

\[
\text{EMP where \( \text{SAL} < 1200 \)}
\]

The result is a single-column table of record addresses for employees earning less than 1200.
Q2. Give names and departments for employees earning less than 1200.
\((\text{EMP where } \text{SAL} < 1200) \cdot (\text{NAME}, (!\text{EMP}.\text{DEPT}))\)
The query returns a two-column table, where the first column contains addresses of names of proper employees, and the second contains addresses of records for their departments.

Q3. For departments having more than 2 employees and located in more than one city give the maximum salary and the average salary.
\((\text{DEPT where count}(\text{EMP}) > 2 \text{ and count}(\text{LOC}) > 1)\)
with \((\max(\text{EMP}.\text{EMP}.\text{SAL}), \avg(\text{EMP}.\text{EMP}.\text{SAL}))\)
The result is a three-column table where the first column contains addresses of proper DEPT records, and the next contain proper numbers.

3. BASIC OBSERVATIONS CONCERNING STORED QUERIES

Example 2

(See Example 1 for the database instance.) A stored queries file based on NETUL is shown in Fig. 3. The additional queries involved mean:
Q4: Names of employees from Sales department.
Q5: An average salary in Sales department.
Q6: Departments located in Paris and employing less than 50 persons.
Q7: Departments together with their employees earning more than 90% of the maximum.

3.1. Stored queries and indices

Consider Example 1. An index for employee names contains pairs \langle Brown, a3\rangle, \langle Casey, a4\rangle, \langle Smith, a5\rangle, \langle Jones, a6\rangle, \langle Lewis, a7\rangle. Similarly, the set of stored queries of the form \(\text{EMP where NAME = ...}\), consists of pairs \langle EMP where NAME = Brown, a3\rangle, \langle EMP where NAME = Casey, a4\rangle, \langle EMP where NAME = Smith, a5\rangle, \langle EMP where NAME = Jones, a6\rangle, \langle EMP where NAME = Lewis, a7\rangle. The only difference is that additional text "EMP where NAME =" precedes an indexing term. Assuming compression (eg. by macro facilities) this text may be reduced, thus we obtain the typical index.

3.2. Flexibility and adaptability

Stored queries do not exclude other (classical) methods of performance improving. By maintaining only most frequent stored queries we can reduce the necessary storage without affecting the average performance efficiency. It is also possible to consider self-adaptable systems, where the system automatically augments (reduces) the set of stored queries. For example, all new queries entering the system may be stored, and queries not used for a long time may be deleted. Many different strategies of adaptation can be used. The problem of adaptability\cite{14} concerns optimal index selection in adaptable or self-adaptable DBMS. Stored queries have at least two advantages in this respect:
- they may be partly created as a side effect of the normal query processing,
- the elementary decision of inserting or deleting concerns single stored query, while in the mentioned case it concerns whole indices which may be very large. Much smaller granularity of decision objects stimulates better adaptability and self-adaptability.

3.3. Equivalent transformation of queries

Syntactic transformation of queries should be performed before storing, eg. reduction of spaces, representing in a canonical form, etc. There are also many semantic rules, eg. "j where \(N_1 = N_2 \text{ and } N_2 = c\)" is equivalent to "j where \(N_1 = c \text{ and } N_2 = c\)\', cf. the tableau method \cite{11}.

3.4. Decomposition of queries

Some queries may be decomposed into more elementary parts, eg. "j where \(p_1 \text{ or } p_2\)" is an equivalent of "(j where \(p_1\)) union (j where \(p_2\))" (\(p_1,p_2\) are predicates); thus the query may be substituted by the union of two elementary queries and such a complex query need not be stored. Similarly for boolean "and". Another kind of decomposition is connected with nested queries, such as "Give best-paid employees".

---

**Fig.3. A stored queries set.**

<table>
<thead>
<tr>
<th>Query</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP where SAL &lt; 1200</td>
<td>\langle a3, a6 \rangle</td>
</tr>
<tr>
<td>EMP where SAL &lt; 1200 \cdot (NAME, (!\text{EMP}.\text{DEPT}))</td>
<td>\langle a3, a4, a6, a7 \rangle</td>
</tr>
<tr>
<td>DEPT where count(EMP) &gt; 2 and count(LOC) &gt; 1 \cdot with (max(EMP.EMP.SAL), avg(EMP.EMP.SAL))</td>
<td>\langle a1, 1400, 1200 \rangle</td>
</tr>
<tr>
<td>DFPT where DNAME = Sales.EMP.EMP.NAME</td>
<td>\langle a1, a4, a5 \rangle</td>
</tr>
<tr>
<td>avg(DEPT where DNAME = Sales.EMP.EMP.SAL)</td>
<td>\langle 1200 \rangle</td>
</tr>
<tr>
<td>DEPT where Paris in LOC and count(EMP) &lt; 50</td>
<td>\langle a1 \rangle</td>
</tr>
<tr>
<td>DEPT with (EMP.EMP where SAL &gt; 0.9 \cdot max(EMP.SAL))</td>
<td>\langle a1, a6 \rangle, \langle a2, a7 \rangle</td>
</tr>
</tbody>
</table>

Proceedings of the 13th VLDB Conference, Brighton 1987 373
EMP where SAL = max(EMP.SAL). It may be decomposed into queries: "max(EMP.SAL)" and "EMP where SAL = c", where c is the result of the first query.

3.5. Derivation of query meaning from other stored queries

An elementary example is "EMP where NAME = Smith. SAL", which may be easily derived from "EMP where NAME = Smith". Network databases, having explicit pointer links, possess more such possibilities, and quite complex queries can be efficiently derived. In RM they would require joining many relations, eg. "Give DNAMEs and NAMES of employees from each Paris department employing some Smith": "EMP where NAME = Smith. !EMPS.DEPT where (Paris in LOC).(DNAME, (EMP.EMP.NAME))" which in RM would require a Cartesian product (a join) of four relations.

3.6. Dispersing stored queries among the database structure

Consider two pairs of queries:
a1) For each department, give the average salary: "DEPT with avg(EMP.EMP.SAL)"
a2) For each department, give DNAME:
   "DEPT with DNAME"
b1) Give the average salary in Smith's department: "EMP where NAME = Smith. !EMPS.DEPT. avg(EMP.EMP.SAL)"
h2) Give DNAME of Smith's department:
   "EMP where NAME = Smith. !EMPS.DEPT. DNAME"

Queries a1 and b1 contain the subquery "avg(EMP.EMP.SAL)", and we can observe from a2 and b2 that it is used in queries similarly to the use of attribute DNAME. Hence "avg(EMP.EMP.SAL)" may be treated as a virtual attribute of DEPT, and its value may be stored within DEPT records. That is, each such record (stored at addr1) may be augmented by an address addr2, and then new triples <addr1, DEPT. addr2>, <addr2, avg(EMP.EMP.SAL)>, are added to the database content, (where v is the actual average salary in the department). Note that the subquery "avg(EMP.EMP.SAL)" is treated as a data name. Due to the orthogonality of NETUL, it may be used in other arbitrary contexts, eg. "Give departments with the maximal average salary": "DEPT where avg(EMP.EMP.SAL)=max(DEPT.avg(EMP.EMP.SAL))".

4. THE ELIMINATION METHOD

The elimination method is based on the concept of subschema. The idea is that with a query and with an update we associate "sufficient" subschemata. If these subschemata are "disjoint", in a proper sense, then the update cannot influence the query meaning. We start from a few basic definitions then prove a theorem concerning the method. Then, as examples of general properties, we construct simple "subschema languages" and show an efficient test for determining whether subschemata are "disjoint". The examples have mainly an illustrative purpose, but we do not exclude their practical meaning.

Let DB denote the set of all database instances: DB = FOVST x CON, where FOVST = powerset of( A ), and CON = powerset of( A x N x ( A U V ) ). Let dbl = <fovl, con1> and db2 = <fov2, con2> be database instances. We define operations on database instances as:

intersection: dbl o db2 = <fovl nfov2, con1 n con2>
sum: dbl + db2 = <fovl U fov2, con1 U con2>
complement: db1 = <A - fov1, (A x N x (A U V)) - con1>
difference: db1 - db2 = db1 o ?&?

Assuming that database instance <g,0> is the minimal boolean element and <A,(A x N x (A U V))> is maximal, we can show that the system <DB, o, +, Z > is a boolean algebra; hence we will use its well known properties. Relation o DB x DB is a partial ordering induced by this algebra; it may be defined as db1 < db2 iff fov1 ? fov2 and con1 ? con2.

A schema is a subset of DB. We are not interested in particular facilities (eg. integrity constraints) that are used for determining schemas and reflect just their main semantic property.

Let S ? DB be a schema. A subschema ss of the schema S is a total function ss: S + DB such that for every db E S holds

ss(db) < db

We assume the following notation. Let D1, D2, D3 be some domains (sets) and let f: U1 ? (U2 ? U3). By f[x1,x2], where x1 E D1 and x2 E D2, we denote g(x2) where g is returned by f for argument x1, g = f[x1].

In the sequel Q denotes a syntactic domain for a query language, R denotes a semantic domain of responses returned by queries, and semq: Q ? (DB -> R) a meaning function. By a database update we mean a transformation of a database instance db1 into db2. Let U be a syntactic domain of an update language and let semu: U ? (DB -> DB) be its meaning function. To simplify notation, instead of semu U db we use l1(db) and instead of semu U db we use l1(db).
A query $q \in Q$ is insensitive to update $u \in U$ in schema $S$ iff for every $db \in S$ holds

$$lq(db) = lq(lul(db))$$  \hspace{1cm} (2)

Such a property should be proven if after update $u$ we would like to eliminate stored query $<q, lq(db)>$ from the updating process.

A subschema $ssq$ is sufficient for query $q \in Q$ in schema $S$ iff for every $db \in S$ holds

$$lq(db) = lq(ssq(db))$$  \hspace{1cm} (3)

If $ssq$ is sufficient for query $q$, then instead of (2) it is enough to prove that for every $db \in S$ holds

$$ssq(db) = ssq(lul(db))$$  \hspace{1cm} (4)

A subschema $ssu$ is sufficient for update $u \in U$ in schema $S$ iff for every $db \in S$ holds

$$db - lul(db) < ssu(db)$$  \hspace{1cm} (5)

and

$$lul(db) - db < ssu(lul(db))$$  \hspace{1cm} (6)

The definition for the subschema sufficiency for updates requires that the old (removed) part of a database and the new (inserted) part should "belong" to the subschema. We do not consider other conditions which may be necessary for accomplishing the updating request, e.g. a query occurring in the request may require a wider subschema.

The subschemas $ss1$ and $ss2$ of schema $S$ are weakly disjoint iff for every $db \in S$ holds

$$ss1(db) o ss2(db) = \langle ?, ? \rangle$$  \hspace{1cm} (7)

The subschemas $ss1$ and $ss2$ of schema $S$ are strongly disjoint iff for each $db1, db2 \in S$ holds

$$ss1(db1) o ss2(db2) = \langle ?, ? \rangle$$  \hspace{1cm} (8)

Now assume that $ssq$ is a sufficient subschema for query $q$, $ssu$ is a sufficient subschema for update $u$, and $ssq, ssu$ are weakly (or strongly) disjoint. Further assumptions which allow us to prove that $q$ is insensitive to $u$ are determined by the following theorem.

Theorem 1

If subschemas $ssq$ and $ssu$ are weakly/strongly disjoint, and they are sufficient respectively for query $q$ and update $u$, then the weakest additional assumptions which allow to prove (4) are the following:

- For weakly disjoint subschemas:
  for every $db \in S$
  $$ssq(db) o db o lul(db) < ssq(lul(db)) + ssu(db)$$
  $$ssq(lul(db)) o db o lul(db) < ssu(db) + ssq(lul(db))$$
  \hspace{1cm} (9)

- For strongly disjoint subschemas:
  for every $db \in S$
  $$ssq(db) o db o lul(db) < ssq(lul(db)) + ssu(db) + ssu(lul(db))$$
  $$ssq(lul(db)) o db o lul(db) < ssq(db) + ssu(db) + ssu(lul(db))$$
  \hspace{1cm} (10)

The sketch of proof of (9); similarly for (10):

Let us denote: $x = db, y = lul(db), a = ssq(db), b = ssq(lul(db)), c = ssu(db), d = ssu(lul(db)), 0 = <?, ?>$. We replace operator $o$ by juxtaposition. According to properties of boolean algebras $z < t$ is equivalent to $z t = 0$, $z = t$ is equivalent to $z < t$ and $t < z$, and $z = 0$ and $t = 0$ is equivalent to $z + t = 0$.

Assumptions:
From (1): $a < x, c < x, b < y, d < y$;
  equivalently: $ax + cx + by + dy = 0$
From (5): $xy < c$
  equivalently: $xyt = 0$
From (6): $yx < d$
  equivalently: $xyd = 0$
From (7): $ac = 0, bd = 0$
  equivalently: $ac + bd = 0$

Summarizing assumptions:
$$ax + cx + by + dy + xyc + xyd + ac + bd = 0$$

The weakest assumption which allows to prove the thesis is $p = 0$, where $p$ is a difference between the left part of the last formula and the left part of the summarizing assumptions, i.e.

$$p = ab + ab -$$

After reduction we obtain: $abcxy + abdy = 0$, or equivalently: $axy < b + c$ and $bxy < a + d$, what completes the proof.

From the weakest assumptions we can easily construct stronger ones. For example, (9) and (10) are implied if $ssq$ satisfies the "subschema continuity property", $[27, 28]$:

$$\forall db1, db2 \in S$$
$$ssq(db1) o db1 o db2 = ssq(db2) o db1 o db2$$

However, this property is sometimes too strong.

Implementation of the elimination method consists in defining a subschema language which allows easy generation of sufficient subschemas for each query and update. A simple method for recognizing that (4) holds should be provided. The efficiency of elimination may depend on the applied subschema language. Subschema languages with better precision (which better approximate necessary parts of the database) allow to eliminate more queries. More precise languages, however, may cause performance difficulties when generating sufficient subschemas and testing whether (4) holds. Note, that a subschema may not be connected with a subschema language. For example, a sufficient subschema
for an update may be considered a function returning an empty database instance for all elements of the schema, with except of two: the database instance before the update (for which the subschema returns deleted elements of $\text{fov}$ and $\text{con}$), and the database instance after the update (for which the subschema returns inserted elements of $\text{fov}$ and $\text{con}$).

As an example consider a subschema language $\text{SS}_1 = \text{powerset of}(\mathbb{N})$, consisting of sets of data names, with the meaning function $\text{MFSS}_1: \text{SS}_1 \rightarrow (\text{DB} \rightarrow \text{DB})$ defined as follows: let $\text{No} \subseteq \mathbb{N}$; then

$$\text{MFSS}_1\text{E No} \{\text{fov, con}\} = \{\text{fov'}, \text{con'}\}$$

where $\text{con'} = \{a,n,x\in \text{con} \mid n \in \text{No}\}$ contains triples having names from $\text{No}$, $\text{fov'} = \text{fov} \cap \{a \in A \mid (\exists n,x)(a,n,x) \in \text{con'}\}$ contains addresses from $\text{fov}$ where some triples from $\con'$ are located. In our intention, if $\text{N}_1$ determines a sufficient subschema for query $q$, $\text{N}_2$ determines a sufficient subschema for update $u$, and $\text{N}_1 \cap \text{N}_2 = \emptyset$, then $q$ is insensitive to the update. We shall prove this assertion. If $\text{N}_1$, $\text{N}_2 \subseteq \mathbb{N}$ are disjoint sets, then database contents returned by $\text{MFSS}_1\text{N}_1\text{N}_2$ and $\text{MFSS}_1\text{N}_2\text{N}_1$ have no common triples. All changed triples (deleted and inserted) are $\text{MFSS}_1\text{IN}_2$. Hence $\text{MFSS}_1\text{IN}_2$ returns exactly the same triples for database instances before and after updating. From the definition of $\text{MFSS}_1$, returned $\text{fov}$-s functionally depend on returned sets of triples, thus fields of vision returned by $\text{MFSS}_1\text{IN}_1\text{N}_1$ are the same for instances before and after updating. Hence condition (4) holds.

A sufficient subschema for a NETUL query consists of all names occurring in this query. A sufficient subschema for the database update $u$ consists of names from all affected triples (removed and inserted), i.e.

$$(\exists a,x)(a,n,x) \in \text{con} \mid \text{lu}l(\text{con}) - \text{con} - \text{lu}l(\text{con}) \rightleftharpoons (12)$$

where $\text{lu}l(\text{con})$ is the updated database content. Thus, for $\text{SS}_1$ generating sufficient subschemas and testing if (4) holds is quite simple.

Example 3

Consider a schema implied by the diagram of Fig.1 and the mapping rules shown in Example 1. Fig.2 presents a current database instance and Fig.3 presents a current stored queries set. The following sets of names generate subschemas sufficient for queries from Example 2; Fig.4 presents a database sub-instance produced by the subschema for $Q_6$.

$Q_1: \{\text{EMP}, \text{SAL}\}$

$Q_2: \{\text{EMP}, \text{SAL}, \text{NAME}, \text{ELECTS}, \text{DEPT}\}$

$Q_3: \{\text{DEPT}, \text{EMP}, \text{NAME}\}$

$Q_4: \{\text{DEPT}, \text{NAME}, \text{ELECTS}, \text{EMP}, \text{NAME}\}$

$Q_5: \{\text{DEPT}, \text{NAME}, \text{ELECTS}, \text{EMP}, \text{SAL}\}$

$Q_6: \{\text{DEPT}, \text{LOC}, \text{EMP}\}$

$Q_7: \{\text{DEPT}, \text{EMP}, \text{SAL}\}$.

Suppose that all Tokyo locations are changed to Kyoto by a NETUL updating request [37] "update DEPT.((X being LOC where $X$ = Tokyo), Kyoto)." For the instance from Fig.2 it means deleting a triple $<a_{23}, \text{LOC}, \text{Tokyo}>$ and inserting $<a_{23}, \text{LOC}, \text{Kyoto}>$. A set $\{\text{LOC}\}$ determines a sufficient subschema for this update. The empty intersection of this set with the sets of names for $Q_1$, $Q_2$, $Q_4$, $Q_5$ and $Q_7$ shows the insensitivity of these queries to the above update. Thus, $Q_1$, $Q_2$, $Q_4$, $Q_5$ and $Q_7$ may be eliminated, while $Q_3$, $Q_6$ may not. Consider another update changing Smith's salary. $\{\text{SAL}\}$ determines a sufficient subschema, thus $Q_4$, $Q_6$ may be eliminated while the rest may not.

A disadvantage of $\text{SS}_1$ is low precision. As a better variant, consider a subschema language $\text{SS}_2$ with expressions being sets $\{m_1, m_2, \ldots, m_k, \langle n_1, V_1 \rangle, \langle n_2, V_2 \rangle, \ldots, \langle n_l, V_l \rangle\}$ where $k > 0$, $l > 0$, $m_1, \ldots, m_k \in \mathbb{N}$, $n_1, \ldots, n_l \in \mathbb{N}$, $V_1, \ldots, V_l$ are subsets of $V$. The idea of such a subschema is similar to the previous one, but we are allowed to restrict values associated with a given name $n_l$ to some set $V_l \subseteq V$. The meaning function $\text{MFSS}_2: \text{SS}_2 \rightarrow (\text{DB} \rightarrow \text{DB})$ for $\text{SS}_2$ is defined as follows:

$$\text{MFSS}_2\{m_1, \ldots, m_k, \langle n_1, V_1 \rangle, \ldots, \langle n_l, V_l \rangle\} \{\text{fov, con}\} = \{\text{fov"}, \text{con"}\}$$

where $\text{con"} = \{a,n,x\in \text{con} \mid n \in \{m_1, \ldots, m_k\}\} \cup \{a,n,v\in \text{con} \mid (\exists n = n_l \land n = n_l \land v \in V_l\}$, $\text{fov"} = \text{fov} \cap \{a \mid (\exists n, x)(a,n,x) \in \text{con"}\}$

With each subschema $\text{ss} \in \text{SS}_2$ we associate a set

Fig.4. Database instance produced by $\text{MFSS}_1\{\text{DEPT, LOC, EMP}\}$

Proceedings of the 13th VLDB Conference, Brighton 1987
representing all possible values which can be associated with data names. We can prove that (4) holds if $ss_2q \in SS_2$ determines a sufficient subschema for query $q$, $ss_2u \in SS_2$ determines a sufficient subschema for update $u$, and

$$\& (ss_2q) \cap \& (ss_2u) = \emptyset \quad (13)$$

Generation of sufficient $SS_2$ subschemas for updates is similar to the previous one, with the modification concerning values: if $<a,n,v>$ is a triple affected by the update (inserted or deleted) then $<n, {...}, v, {...}>$ belongs to the $SS_2$ expression.

A slight difficulty concerning $SS_2$ subschema language is caused by associating sufficient subschemas with queries. Since $SSI \subseteq SS_2$, we could use the previous rules, the precision, however, might be unsatisfactory. More sophisticated rules are necessary, and they may depend on query types.

Example 4

The sufficient $SS_2$ subschemas for queries from Example 2 are:

1. $Q_1$: $EMP, \langle SAL, \{x | x < 1200\}>$
2. $Q_2$: $EMP, NAME, EMP$, $DEPT, \langle SAL, \{x | x < 1200\}>$
3. $Q_3$: $DEPT, EMP$, $LOC$, $EMP, SAL$
4. $Q_4$: $DEPT, EMP$, $EMP$, $NAME$, $DNAME$, $\langle Sales\rangle$
5. $Q_5$: $DEPT, \langle EMP, \langle SAL, \langle DNAME, \langle Sales\rangle\rangle\rangle$
6. $Q_6$: $DEPT, EMP$, $\langle LOC, \langle Paris\rangle\rangle$
7. $Q_7$: $DEPT$, $EMP$, $\langle SAL\rangle$

A sufficient subschema for the update from Example 3 is $\langle LOC, \langle Tokyo, Kyoto\rangle\rangle$. According to (13) $Q_1Q_2Q_4Q_5Q_6Q_7$ may be eliminated from the updating process, while $Q_3$ may not. A sufficient subschema for the update which changes Smith's salary from 1400 to 1500 is $\langle SAL, \{1400, 1500\}\rangle$, thus $Q_1Q_2Q_4Q_6$ may be eliminated, while $Q_3Q_5Q_7$ may not.

$SS_2$ subschemas correspond to the well known concept of "semi-join". A semi-join of relation $R_1$ w.r.t relation $R_2$ is obtained by joining $R_1$ and $R_2$ and projecting the result on the attributes of $R_1$. If $R_1$ is stored in a site $s_1$ and $R_2$ stored in a different site $s_2$, we can calculate the join of $R_1$ and $R_2$ by sending $R_1$ from $s_1$ to $s_2$. To optimize the volume of data to be sent, we can send the semi-join of $R_1$ only. To do this, we must previously send information about $R_2$ from $s_2$ to $s_1$, namely, names of joined attributes and their values in $R_2$. This information may be specified as a $SS_2$ subschema. We think that the concept of sufficient subschema may have some meaning for distributed query processing.

5. Correction

The majority of stored queries may be eliminated from the updating by the methods explained in the previous section. Some remaining queries may be corrected. After updates, changes in the database are usually local, hence frequently a part of a query meaning remains valid for the updated database instance. The correction consists in deleting from the old meaning all the "uncertain" elements and augmenting the meaning by elements obtained through evaluation of the query for a part (as small as possible) of the new database instance which was affected by the update.

Previously we did not provide special restrictions concerning NETUL queries. However, correction is a more difficult problem; probably no general method exists. Thus we must assume some semantic constraints. To formalize them we introduce the concept of access function and its transitive closure.

Let $con \subseteq A \times N \times (A \cup V)$ be a database content, and let $Ao \subseteq A$. We define a function "sub", with arguments $Ao$ and $con$, returning addresses "subordinated" to $Ao$:

$$\text{sub}(Ao, con) = \{b \in A \mid \exists a \in Ao \land <a, n, b> \in con\}$$

Then, we define a family of functions ($\text{sub}_0$, $\text{sub}_1$, $\text{sub}_2$, $\ldots$, $\text{sub}(i)$), where $\text{sub}(i)(Ao, con)$ returns the set of addresses that are accessible from $Ao$ in exactly $i$ steps, that is: $\text{sub}(0)(Ao, con) = Ao$

for $i > 0$:

$$\text{sub}(i)(Ao, con) = \text{sub}(\text{sub}(i-1)(Ao, con), con)$$

The transitive closure $\text{sub}^*(Ao, con)$ of $\text{sub}$, defined as $\bigcup_{i=0}^{\infty} \text{sub}(i)(Ao, con)$ (or by a fixpoint: $\text{sub}^*(Ao, con) = X$, where $X = Ao \cup \text{sub}(X, con)$) returns all addresses that are accessible from $Ao$ in an arbitrary number of steps.

Let $<fov, con>$ be a database instance. All triples that are inaccessible from $fov$ in any number of steps may be removed, thus we define a function $\text{red}$, with arguments $<fov, con>$ and $con$, returning addresses "subordinated" to $<fov, con>$:

$$\text{red}(<fov, con>) = <fov, \{<a, n, x> \in con \mid a \in \text{sub}^*(fov, con)\}>$$

Now we can formalize constraints on allowed queries $q$ as follows:

1. $\text{lq}_1(db) = \text{lq}_1(\text{red}(db))$
2. $\text{lq}_n(<fov, con>) = (\text{sub}^*(fov, con))$, $n = 1, 2, 3, \ldots$
(C3) \[ l\bar{q}(f\text{ov}_1 \cup f\text{ov}_2, con) = l\bar{q}(f\text{ov}_1, con) \cup l\bar{q}(f\text{ov}_2, con) \]

Constraint (C1) is obviously satisfied by all NETUL queries. Constraint (C2) states that the meaning of a query is an n-ary relation over addresses accessible from the field of vision. Many queries possess this property, e.g. Q1, Q2, Q4, Q5, Q7 from Fig.3. Substantial parts of others also have it, e.g. in Q5 the argument of avg. Constraint (C3) requires additivity of a query meaning w.r.t. the field of vision. Many typical queries have this property, e.g. Q1, Q2, Q3, Q4, Q6. Q5 does not possess it, but the query being the argument of avg does. The constraint is typically satisfied by queries of the "navigational type"; in general, however, it is not tolerant to nested queries (eq. Q6 and Q7).

Constraint (C3) implies

\[ l\bar{q}(f\text{ov}, con) = \bigcup_{a \in f\text{ov}} l\bar{q}(\{a\}, con) \quad (14) \]

that is, the meaning of a query may subsequently be evaluated for single elements of f\text{ov}. By addr(con) we denote addresses occupied by con, i.e.

\[ \text{addr}(con) = \{a | (\exists n,x) <a,n,x> \in con\} \]

Now assume that an update changes a database instance \( f\text{ov}_1, con_1 \) into \( f\text{ov}_2, con_2 \). We determine elements of the field of vision which are "uncertain" after this update, since according to (14) they may produce elements of the old meaning which may not be valid any longer. These are:
- addresses removed from f\text{ov}, i.e. f\text{ov}_1-f\text{ov}_2,
- addresses inserted into f\text{ov}, i.e. f\text{ov}_2-f\text{ov}_1,
- addresses from f\text{ov} from which deleted triples are accessible, i.e. F1 = \( \{a \in f\text{ov}_1 | \sub{\{a\}, con_1} \cap \text{addr}(con_1-con_2) \neq \emptyset\} \),
- addresses from f\text{ov} from which inserted triples are accessible, i.e. F2 = \( \{a \in f\text{ov}_2 | \sub{\{a\}, con_2} \cap \text{addr}(con_2-con_1) \neq \emptyset\} \).

Hence the set X of "uncertain" f\text{ov} elements is:

\[ X = f\text{ov}_1-f\text{ov}_2 \cup f\text{ov}_2-f\text{ov}_1 \cup F1 \cup F2 \quad (15) \]

Then, "uncertain" elements of the old meaning of query q are \( l\bar{q}(f\text{ov} \cap X, con_1) \) and they should be removed from the meaning. On the other hand, q should be reevaluated for addresses \( X \cap f\text{ov}_2 \), and the result should be appended to the meaning.

However, after removing some elements from the old meaning it may happen that we have removed too much. Indeed, assume \( a_1, a_2 \in f\text{ov}, a_1 \in X, a_2 \notin X \) and \( l\bar{q}(\{a_1\}, con_1) = l\bar{q}(\{a_2\}, con_2) \). Thus, we should remove all elements determined by the left part of the last formula, but at the same time we remove elements determined by the right part, which should not be removed. To avoid this problem, we must reevaluate q for a slightly wider part of f\text{ov}, namely, for the set \( X \cap X \cup Y \cap f\text{ov}_2 \), where Y contains additional elements of f\text{ov} from which tuples of addresses removed from the old meaning are accessible, i.e.

\[ Y = \{a \in f\text{ov}_1 | \sub{\{a\}, con_1} \} \cap \bigcup_{i=1}^{n} l\bar{q}(f\text{ov}_i \cap X, con_i) \quad (16) \]

where n is the size of tuples returned by q.
Now we are ready to formulate the following theorem:

**Theorem 2**

If query q satisfies conditions (C1), (C2) and (C3), then for any database instances \( f\text{ov}_1, con_1 \), \( f\text{ov}_2, con_2 \) holds

\[ l\bar{q}(f\text{ov}_2, con_2) - l\bar{q}(f\text{ov}_1, con_1) \quad (17) \]

\[ - l\bar{q}(f\text{ov}_1 \cap X, con_1) \cup l\bar{q}(f\text{ov}_2 \cap (X \cup Y), con_2) \]

where X and Y are given by formulas (15), (16).

A variant of this theorem is proven in [273]. An easy (but paper consuming) proof may be done according to the informal considerations preceding the theorem. Its kernel consists of application of (C1): we have to show that \( \text{red}(f\text{ov} \cap X, con_1) = \text{red}(f\text{ov} \cap X, con_2) \); thus \( l\bar{q}(f\text{ov} \cap (X \cup Y), con_1) = l\bar{q}(f\text{ov} \cap (X \cup Y), con_2) \). The rest follows from the additivity of the query meaning.

There may be many variants of this theorem, depending on the assumptions on query meaning or allowed database instances; a few are considered in [273]. Below we discuss two such variants.

The set Y occurring in formula (17) may be removed if we assume that for query q there exists a sufficient hierarchical subschema. A subschema ssq of schema S is said to be hierarchical if for every db \( \in S \) holds:

\[ \text{ssq}(db) = <f\text{ov}ssq, conssq> \]

\[ (\forall a_1, a_2 \in f\text{ov}ssq) a_1 \neq a_2 \implies \sub{\{a_1\}, conssq} \cap \sub{\{a_2\}, conssq} = \emptyset \]

That is, clusters of addresses accessible from different addresses of f\text{ovssq} do not overlap. A hierarchical subschema exists for each query from Fig.3. If in (C1), (C2), (C3), (15), (16) and (17) for each db we replace db by ssq(db), which is quite reasonable if ssq is a sufficient subschema, then Y = \( \emptyset \). Indeed, in this case the described situation leading us to the definition of Y never arises.

The second case concerns removing (C2) and Y totally. The reason which has led us to (C2)
is the same as the reason for which we consider set $Y$: for different addresses of $fov$, query $q$ may return the same tuple. Thus, instead of a stored query $<q, lq(fov, con)>$, we can store $<q, I>$, where $I$ is a table defined as

$$T = \bigcup_{a \in fov} \{a\} \times lq(\{a\}, con)$$

We augment the table returned by $q$ by an additional column storing elements of $fov$ for which $q$ returns the given tuple. Due to additivity, the meaning of $q$ may be obtained through removing this column. Now for different $fov$ elements $q$ never returns the same tuple of $T$, thus constraint (C2) is unnecessary. However, this possibility is achieved at expense of additional storage space and processing time.

The straightforward implementation of the correction method according to formula (17) may be described by the following algorithm:

1. Determine addresses of removed triples, i.e. $\text{addr}(\text{con1-con2})$.
2. Navigate in $\text{con1}$ in the opposite direction, from these addresses to the field of vision, to establish set $F_1$ of uncertain elements.
3. Determine addresses of inserted triples, i.e. $\text{addr}(\text{con2-con1})$.
4. Navigate in $\text{con2}$ in the opposite direction, from these addresses to the field of vision, to establish set $F_2$ of uncertain elements.
5. Calculate uncertain elements of the query meaning, i.e.

$$lq(fov_1 - fov_2 \cup F_1 \cup fov_1 \cap F_2, \text{con1})$$

6. Remove uncertain elements from the meaning.
7. Navigate in $\text{con1}$ in the opposite direction, from these uncertain elements to the field of vision, to establish set $Y$.
8. Calculate $lq(fov_2 - fov_1 \cup fov_2 \cap (F_1 U Y) \cup F_2, \text{con2})$ and append the result to the query meaning.

Navigation in opposite direction requires special organization of physical data. Usually, data organizations assumed in classical network databases permit such a possibility. For example, in CODASYL databases set occurrences are organized as linked rings of pointers causing no problems in navigation from an owner to a member and otherwise.

6. CONCLUSION

The relational data model has unacceptable intellectual drawbacks in vital database aspects thus we believe that much more effort is necessary to develop network/semantic data models. Query optimization methods developed for the relational data model do not work for network/semantic data models since these methods are strongly directed towards optimization of joins (or Cartesian products) which are not specific for query languages addressing network data structures. Besides, the methods are unapplicable to query languages having the power of algorithmic programming languages, such as NETUL. Hence the need for new approaches to query optimization. In this paper we have proposed a group of methods based on the idea of stored queries. Stored queries give a possibility of performance improvement regardless of the data model assumed and of the expressive power of query language. We have outlined basic properties of the idea, and have sketched two complementary methods, elimination and correction, for solving the problem of updating stored queries.

The efficiency of the elimination method, based on the concept of subschema, depends to a great extent on the features of the subschemas used. The proposed method, based on subschemas generated by sets of data names associated with allowed values, seems to be convenient for implementation and efficient.

Correction works particularly well when the updated part of the database is small, which is quite a typical situation. Two possibly more efficient variants of the main method are also proposed.

The formalism outlined in this paper has many advantages over well known database formalisms. In particular, it provides a consistent explanation of all persistent features of database systems and query languages, such as duplicates, ordering, updating, and so on. We hope that it will appear a convenient tool also for studying other aspects of the database theory.

The stored queries data organization was experimentally implemented and some essential intuitions of this idea were examined in [27].

REFERENCES


