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 represented but have to be determined by users during formulation of queries, causing superflous complexity. (The 5th normal form relieves this problem for some types of queries, but it is unreasonable for many reasons.) Current RDM theory is inconsequent in its attitude to duplicates, ordering, updating, arithmetics, aggregates, transitive closures, and so on, refusing to yield general and consistent semantics of these vital concepts. (Existing attempts, eg. [2,9,20,21,32], do not explain all aspects and are ignored by the main stream of the RDM school.) In consequence we prefer to deal with network/semantic approaches which offer more potentiality for future database systems.

There are few papers concerning query optimization addressing network data structures [10,11,15]. 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 "response" 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 [23], 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. [8,17,25,40]. 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, overlaping 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 $con \subseteq A \times N \times A \cup A \times N \times V$. Triples <a1,n,a2> and <a,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 a 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, $<a1,n1,v1> \in con$ and $<a1,n2,v2> \in con$ implies v1 = v2. 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 [5,30,40], 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 data 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 objects should be introduced, which associate data names occuring in queries with proper components of database instances. (eg. names of relations with instantaneous relations). This "environment" (in the programming languages terminology) causes no problems in typical cases; however, non-trivial problems arise from advanced conceptual modelling notions, such as aggregations, "is-a" relationships, 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 EMP record types and EMPS set type from DEPT to EMP. 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 EMP records (Brown,1000), (Casey,1200), (Jones,1000), (Lewis,1500), (Smith,1400), and two occurences 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.

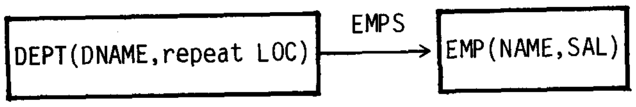

Fig.1. A database structure

This database instance is expressed as a pair <fov,con>, 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 contain 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:

<a1,DEPT,a11>, <a11,DNAME,"Sales">,
<a1,DEPT,a12>, <a12,LOC,"Rome">,
<a1,DEPT,a13>, <a13,LOC,"Paris">,
<a1,DEPT,a14>, <a14,EMPS,a3>,
<a1,DEPT,a15>, <a15,EMPS,a4>,
<a1,DEPT,a16>, <a16,EMPS,a5>

The corresponding graph representation of the whole database instance <fov,con>, 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 "."), operator "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.

EMP where SAL < 1200

The result is a single-column table of record addresses for employees earning less than 1200.

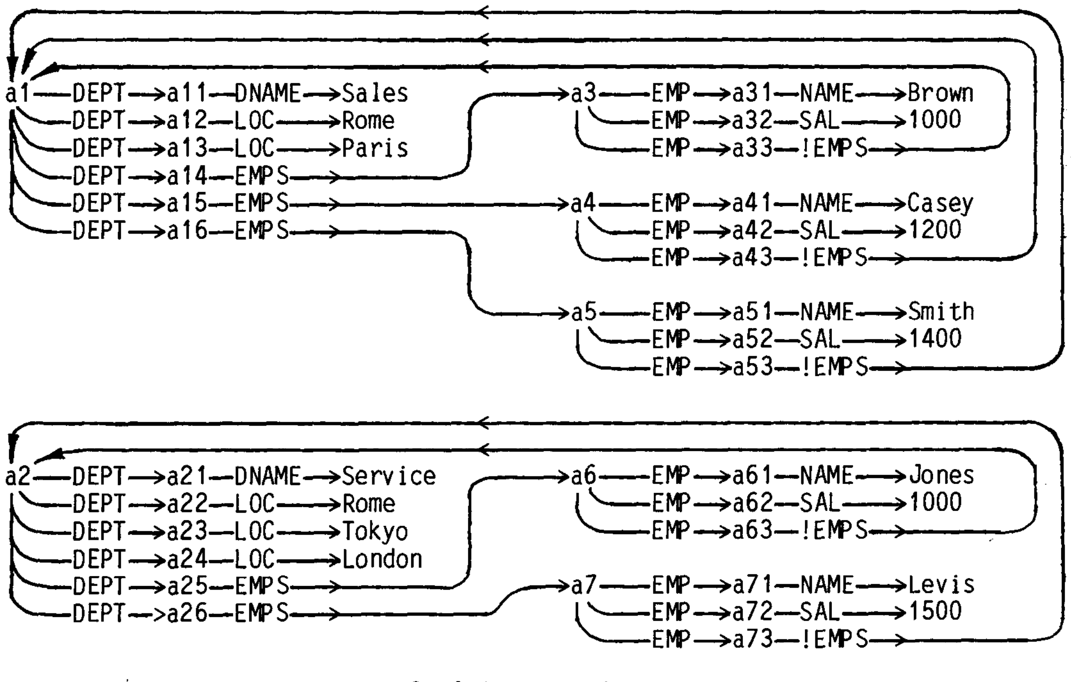

Fig.2. A database instance

Q2. Give names and departments for employees earning less than 1200.
(EMP where SAL < 1200) . (NAME, (!EMPS.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.
(DEPT where count(EMPS) > 2 and count(LOC) > 1)
with (max(EMPS.EMP.SAL), avg(EMPS.EMP.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 <Brown,a3>, <Casey,a4>, <Smith,a5>, <Jones,a6>, <Lewis,a7>. Similarly, the set of stored queries of the form "EMP where NAME = ...". consists of pairs <EMP where NAME = Brown, a3>, <EMP where NAME = Casey, a4>, <EMP where NAME = Smith, a5>, <EMP where NAME = Jones, a6>, <EMP where NAME = Levis, a7>. 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 [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 N1 = N2 and N2 = c" is equivalent to "j where N1 = c and N2 = c", cf. the tableau method [1].

3.4. Decomposition of queries

Some queries may be decomposed into more elementary parts, eg. "j where p1 or p2" is an equivalent of "(j where p1) union (j where p2)" (p1,p2 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",

	Query	Response
Q1	EMP where SAL < 1200	<a3, a6>
Q2	EMP where SAL < 1200 . (NAME, (!EMPS.DEPT))	<<a31,a1>,<a61,a2>>
Q3	DEPT where count(EMPS) > 2 and count(LOC) > 1 with (max(EMPS.EMP.SAL), avg(EMPS.EMP.SAL))	<<a1, 1400, 1200>>
Q4	DEPT where DNAME = Sales.EMPS.EMP.NAME	< a31, a41, a51 >
Q5	avg(DEPT where DNAME = Sales.EMPS.EMP.SAL)	< 1200 >
Q6	DEPT where Paris in LOC and count(EMPS) < 50	< a1 >
Q7	DEPT with (EMPS.EMP where SAL > 0.9 * max(EMP.SAL))	<<a1,a5>, <a2,a7>>

Fig.3. A stored queries set.

i.e. "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 RDM 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, (EMPS.EMP.NAME))"
which in RDM 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(EMPS.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(EMPS.EMP.SAL)"
b2) Give DNAME of Smith's department:
"EMP where NAME = Smith.!EMPS.DEPT. DNAME

Queries a1 and b1 contain the subquery "avg(EMPS.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(EMPS.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(EMPS.EMP.SAL),v> are added to the database content, (where v is the actual average salary in the department). Note that the subquery "avg(EMPS.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(EMPS.EMP.SAL)=max(DEPT.avg(EMPS.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" subschemas. If these subschemas 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 subschemas are "disjoint". The examples have mainly an ilustrative 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 db1 = <fov1, con1> and db2 = <fov2, con2> be database instances. We define operations on database instances as:

intersection:
db1 o db2 = <fov1 $\cap$ fov2, con1 $\cap$ con2>
sum: db1 + db2 = <fov1 U fov2, con1 U con2>
complement:
$\overline{\text{db1}}$ = <A - fov1, (A x N x (A U V)) - con1>
difference: db1 - db2 = db1 o $\overline{\text{db2}}$

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

A <u>schema</u> 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 $\subseteq$ DB be a schema. A <u>subschema</u> ss of the schema S is a total function ss: S $\longrightarrow$ DB such that for every db $\in$ S holds

$$ss(db) \prec db \qquad (1)$$

We assume the following notation. Let D1, D2, D3 be some domains (sets) and let f: D1 $\longrightarrow$ (D2 $\longrightarrow$ D3). By f[x1].x2, where x1 $\in$ D1 and x2 $\in$ 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 $\longrightarrow$ (DB $\longrightarrow$ 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 $\longrightarrow$ (DB $\longrightarrow$ DB) be its meaning function. To simplify notation, instead of semq[q].db we use |q|(db) and instead of semu[u].db we use |u|(db).

A query $q \in Q$ is <u>insensitive to update</u> $u \in U$ in schema S iff for every $db \in S$ holds

$$|q|(db) = |q|(|u|(db)) \qquad (2)$$

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

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

$$|q|(db) = |q|(ssq(db)) \qquad (3)$$

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

$$ssq(db) = ssq(|u|(db)) \qquad (4)$$

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

$$db - |u|(db) \prec ssu(db) \qquad (5)$$

$$\text{and} \quad |u|(db) - db \prec ssu(|u|(db)) \qquad (6)$$

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, eg. a query occuring in the request may require a wider subschema.

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

$$ss1(db) \circ ss2(db) = <\emptyset,\emptyset> \qquad (7)$$

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

$$ss1(db1) \circ ss2(db2) = <\emptyset,\emptyset> \qquad (8)$$

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 $db \in S$

$$ssq(db) \circ db \circ |u|(db) \prec ssq(|u|(db)) + ssu(db)$$
$$ssq(|u|(db)) \circ db \circ |u|(db) \prec ssq(db) + ssu(|u|(db)) \qquad (9)$$

- For strongly disjoint subschemas:
  for every $db \in S$

$$ssq(db) \circ db \circ |u|(db) \prec ssq(|u|(db)) + ssu(db) + ssu(|u|(db))$$
$$ssq(|u|(db)) \circ db \circ |u|(db) \prec ssq(db) + ssu(db) + ssu(|u|(db)) \qquad (10)$$

The sketch of proof of (9); similarly for (10): Let us denote: $x = db$, $y = |u|(db)$, $a = ssq(db)$, $b = ssq(|u|(db))$, $c = ssu(db)$, $d = ssu(|u|(db))$, $0 = <\emptyset,\emptyset>$. We replace operator o by juxtaposition. According to properties of boolean algebras $z \prec t$ is equivalent to $z\overline{t} = 0$, $z = t$ is equivalent to $z \prec t$ and $t \prec z$, and $z = 0$ and $t = 0$ is equivalent to $z + t = 0$.

Assumptions:
From (1): $a \prec x$, $c \prec x$, $b \prec y$, $d \prec y$;
  equivalently: $a\overline{x} + c\overline{x} + b\overline{y} + d\overline{y} = 0$
From (5): $x\overline{y} \prec c$
  equivalently: $x\overline{y}\overline{c} = 0$
From (6): $y\overline{x} \prec d$
  equivalently: $\overline{x}y\overline{d} = 0$
From (7): $ac = 0$, $bd = 0$
  equivalently: $ac + bd = 0$
Summarizing assumptions:
  $a\overline{x} + c\overline{x} + b\overline{y} + d\overline{y} + x\overline{y}\overline{c} + \overline{x}y\overline{d} + ac + bd = 0$
Thesis from (4): $a = b$
  equivalently: $a\overline{b} + \overline{a}b = 0$

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.
$p = a\overline{b} + \overline{a}b -$
  $(a\overline{x} + c\overline{x} + b\overline{y} + d\overline{y} + x\overline{y}\overline{c} + \overline{x}y\overline{d} + ac + bd) = 0$
After reduction we obtain: $a\overline{b}\overline{c}xy + \overline{a}b\overline{d}xy = 0$, or equivalently: $axy \prec b + c$ and $bxy \prec a + d$, 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]:

$$(\forall db1,db2 \in S) \qquad (11)$$
$$ssq(db1) \circ db1 \circ db2 = ssq(db2) \circ db1 \circ db2$$

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 fov and con), and the database instance after the update (for which the subschema returns inserted elements of fov and con).

As an example consider a subschema language SS1 = powerset_of(N), consisting of sets of data names, with the meaning function MFSS1: SS1 $\longrightarrow$ (DB $\longrightarrow$ DB) defined as follows: let No $\subseteq$ N; then

MFSS1[No].<fov,con> = < fov', con' >

where con' = $\{$<a,n,x>$\in$con $|$ n$\in$No$\}$ contains triples having names from No,

fov' = fov $\cap$ $\{$a$\in$A $|$ ($\exists$n,x) <a,n,x> $\in$ con'$\}$ contains addresses from fov, where some triples from con' are located. In our intention, if N1 determines a sufficient subschema for query q, N2 determines a sufficient subschema for update u, and N1 $\cap$ N2 = $\emptyset$, then q is insensitive to the update. We shall prove this assertion. If N1, N2 $\subseteq$ N are disjoint sets, then database contents returned by MFSS1[N1] and MFSS1[N2] have no common triples. All changed triples (deleted and inserted) are returned by MFSS1[N2]. Hence MFSS1[N1] returns exactly the same triples for database instances before and after updating. From the definition of MFSS1, returned fov-s functionally depend on returned sets of triples, thus fields of vision returned by MFSS1[N1] are the same for instances before and after updating. Hence condition (4) holds.

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

$$\{n \mid (\exists a,x)\ \langle a,n,x\rangle \in (|u|(con) - con \cup con - |u|(con))\} \tag{12}$$

where |u|(con) is the updated database content. Thus, for SS1 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 Q6.

Q1: { EMP, SAL }
Q2: { EMP, SAL, NAME, !EMPS, DEPT }
Q3: { DEPT, EMPS, LOC, EMP, SAL }
Q4: { DEPT, DNAME, EMPS, EMP, NAME }
Q5: { DEPT, DNAME, EMPS, EMP, SAL }
Q6: { DEPT, LOC, EMPS }
Q7: { DEPT, EMPS, EMP, SAL }.

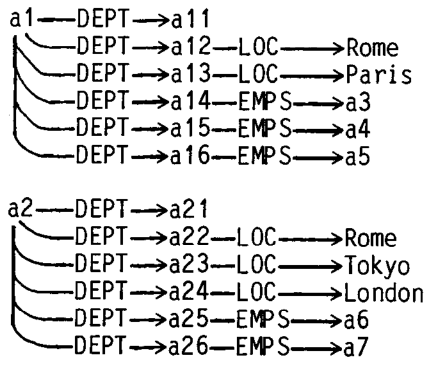

Fig.4. Database instance produced by MFSS1[{DEPT, LOC, EMPS}]

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 <a23,LOC,Tokyo> and inserting <a23,LOC,Kyoto>. A set {LOC} determines a sufficient subschema for this update. The empty intersection of this set with the sets of names for Q1, Q2, Q4, Q5 and Q7 shows the insensitivity of these queries to the above update. Thus, Q1, Q2, Q4, Q5, Q7 may be eliminated, while Q3, Q6 may not. Consider another update changing Smith's salary. {SAL} determines a sufficient subschema, thus Q4, Q6 may be eliminated while the rest may not.

A disadvantage of SS1 is low precision. As a better variant, consider a subschema language SS2 with expressions being sets

{m1, m2,..., mk, <n1,V1>, <n2,V2>,..., <nl,Vl>}

where k >= 0, l >= 0, m1,...,mk $\in$ N, n1,..,nl $\in$ N, V1,...Vl 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 ni to some set Vi $\subseteq$ V. The meaning function MFSS2: SS2 $\longrightarrow$ (DB $\longrightarrow$ DB) for SS2 is defined as follows:

MFSS2[{m1,..,mk,<n1,V1>,..,<nl,Vl>}].<fov, con> = <fov", con">

where con" = $\{$<a,n,x>$\in$con $|$ n$\in$$\{$m1,...,mk$\}$ $\}$ $\cup$ $\{$<a,n,v>$\in$con $|$ ($\exists$i=1,..,l) n = ni and v$\in$Vi$\}$,

fov" = fov $\cap$ $\{$ a $|$ ($\exists$n,x) <a,n,x> $\in$ con" $\}$

With each subschema ss $\in$ SS2 we associate a set

$$\&(ss) = \{<m,v> \mid m \in ss \text{ and } v \in V\} \cup \{<n,v> \mid (\exists Vo \subseteq V)\ <n,Vo> \in ss \text{ and } v \in Vo\}$$

representing all possible values which can be associated with data names. We can prove that (4) holds if ss2q $\in$ SS2 determines a sufficient subschema for query q, ss2u $\in$ SS2 determines a sufficient subschema for update u, and

$$\&(ss2q) \cap \&(ss2u) = \emptyset \qquad (13)$$

Generation of sufficient SS2 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 SS2 expression.

A slight difficulty concerning SS2 subschema language is caused by associating sufficient subschemas with queries. Since SS1 $\subseteq$ SS2, 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 SS2 subschemas for queries from Example 2 are:

Q1: {EMP, <SAL,{x|x<1200}>}
Q2: {EMP, NAME, !EMPS, DEPT, <SAL,{x|x<1200}>}
Q3: {DEPT, EMPS, LOC, EMP, SAL}
Q4: {DEPT, EMPS, EMP, NAME, <DNAME, {Sales}>}
Q5: {DEPT, EMPS, EMP, SAL, <DNAME, {Sales}>}
Q6: {DEPT, EMPS, <LOC, {Paris}>}
Q7: {DEPT, EMPS, EMP, SAL}

A sufficient subschema for the update from Example 3 is {<LOC, {Tokyo, Kyoto}>}. According to (13) Q1,Q2,Q4,Q5,Q6,Q7 may be eliminated from the updating process, while Q3 may not. A sufficient subschema for the update which changes Smith's salary from 1400 to 1500 is {<SAL, {1400,1500}>}, thus Q1,Q2,Q4,Q6 may be eliminated, while Q3,Q5,Q7 may not.

SS2 subschemas correspond to the well known concept of "semi-join". A semi-join of relation R1 w.r.t relation R2 is obtained by joining R1 and R2 and projecting the result on the attributes of R1. If R1 is stored in a site s1, and R2 stored in a different site s2, we can calculate the join of R1 and R2 by sending R1 from s1 to s2. To optimize the volume of data to be sent, we can send the semi-join of R1 only. To do this, we must previously send information about R2 from s2 to s1, namely, names of joined attributes and their values in R2. This information may be specified as a SS2 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 x N x (A U 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:

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

Then, we define a family of functions (sub0, sub1, sub2,..,sub(i),..), where sub(i)(Ao, con) returns the set of addresses that are accessible from Ao in exactly i steps, that is:

$$\text{sub0}(Ao,con) = Ao$$

for i > 0:

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

The transitive closure sub* of sub, defined as

$$\text{sub*}(Ao, con) = \bigcup_{i=0}^{\infty} \text{sub}(i)(Ao, con)$$

(or by a fixpoint: sub*(Ao,con) = X, X = Ao U 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 red: DB $\longrightarrow$ DB perfoming such operation:

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

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

(C1) $\quad |q|(db) = |q|(\text{red}(db))$

(C2) $\quad |q|(<fov,con>) \subseteq (\text{sub*}(fov,con))^n, \quad n = 1,2,3,...$

(C3) $|q|(<fov1 \cup fov2, con>) = |q|(<fov1,con>) \cup |q|(<fov2,con>)$

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, eg. Q1, Q2, Q4, Q6, Q7 from Fig.3. Substantial parts of others also have it, eg. 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, eg. 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 (eg. Q5 and Q7).

Constraint (C3) implies

$$|q|(<fov,con>) = \bigcup_{a \in fov} |q|(<\{a\}, con>) \qquad (14)$$

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

$$addr(con) = \{a \mid (\exists n,x) <a,n,x> \in con\}$$

Now assume that an update changes a database instance <fov1, con1> into <fov2, con2>. We determine elements of the field of vision which are "uncertain" after this update, since according to (14) they may produce elements of the query meaning which may not be valid any longer. These are:
- addresses removed from fov, i.e. fov1-fov2,
- addresses inserted into fov, i.e. fov2-fov1
- addresses from fov from which deleted triples are accessible, i.e. F1 = $\{a \in fov1 \mid sub^*(\{a\},con1) \cap addr(con1-con2) \neq \emptyset\}$
- addresses from fov from which inserted triples are accessible, i.e. F2 = $\{a \in fov2 \mid sub^*(\{a\},con2) \cap addr(con2-con1) \neq \emptyset\}$.

Hence the set X of "uncertain" fov elements is:

$$X = fov1-fov2 \cup fov2-fov1 \cup F1 \cup F2 \qquad (15)$$

Then, "uncertain" elements of the old meaning of query q are $|q|(<fov1 \cap X, con1>)$ and they should be removed from the meaning. On the other hand, q should be reevaluated for addresses $X \cap fov2$, 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 a1, a2 $\in$ fov, a1 $\in$ X, a2 $\notin$ X and $|q|(<\{a1\}, con1>) = |q|(<\{a2\}, con2>)$. 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 fov, namely, for the set $(X \cup Y) \cap fov2$, where Y contains additional elements of fov1 from which tuples of addresses removed from the old meaning are accessible, i.e.

$$Y = \{a \in fov1 \mid (sub^*(\{a\},con1))^n \cap |q|(<fov1 \cap X, con1>) \neq \emptyset\} - X \qquad (16)$$

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 <fov1, con1>, <fov2, con2> holds

$$|q|(<fov2,con2>) = |q|(<fov1,con1>) - |q|(<fov1 \cap X,con1>) \cup |q|(<fov2 \cap (X \cup Y),con2>) \qquad (17)$$

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

A variant of this theorem is proven in [27]. 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 red(<fov1-(XUY),con1>)=red(<fov1-(XUY),con2>); thus $|q|(<fov1-(X \cup Y), con1>) = |q|(<fov1-(X \cup Y), con2>)$. 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 [27]. Below we discuss two such variants.

The set Y occuring 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 <u>hierarchical</u> if for every db $\in$ S holds:

ssq(db) = <fovssq, conssq> and

$$(\forall a1, a2 \in fovssq)\ a1 \neq a2 \Longrightarrow sub^*(\{a1\},conssq) \cap sub^*(\{a2\},conssq) = \emptyset$$

That is, clusters of addresses accessible from different addresses of fovssq 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, |q|(<fov,con>)> we can store <q, T>, where T is a table defined as

$$T = \bigcup_{a \in fov} \{a\} \times |q|(<\{a\},con>)$$

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. addr(con1-con2)
2. Navigate in con1 in the opposite direction, from these addresses to the field of vision, to establish set F1 of uncertain elements.
3. Determine addresses of inserted triples, i.e. addr(con2-con1)
4. Navigate in con2 in the opposite direction, from these addresses to the field of vision, to establish set F2 of uncertain elements.
5. Calculate uncertain elements of the query meaning, i.e. $|q|(fov1-fov2 \cup F1 \cup fov1 \cap F2, con1)$
6. Remove uncertain elements from the meaning.
7. Navigate in con1 in the opposite direction, from these uncertain elements to the field of vision, to establish set Y.
8. Calculate $|q|(fov2-fov1 \cup fov2 \cap (F1 \cup Y) \cup F2, con2)$ 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 occurences 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

[1] Aho, A.V., Sagiv, Y., and Ullman, J.D. Efficient optimization of a class of relational expressions. ACM Trans. on Database Syst. 4, 4 (1979) 435-454

[2] Aho, A.V., and J.D.Ullman. Universality of data retrieval languages. Proc. 6th ACM Symp. on Principles of Programming languages, San-Antonio, Texas 1979, 110-117

[3] Bernstein, P.A., and Blaustein, B. A simplification algorithm for integrity assertions and concrete views. Proc. 5th Computer Software and Application Conf.,

Chicago, Nov. 1981

[4] Blakeley, J. A., Larson, P. -A., and Tompa, F. W. Efficiently updating materialized views. Proc. ACM SIGMOD Conf. Washington D.C. 1986, 61-71

[5] Buneman, O.P., and R.E.Frankel. FQL - A functional query language. Proc. ACM SIGMOD Conf., Boston 1979, 52-57

[6] Chen, P.P.S. The entity-relationship model: towards a unified view of data. ACM Trans. on Database Syst. 1, 1 (1976) 9-36

[7] Comer, D.: The difficulty of optimum index selection. ACM Trans. on Database Syst. 3, 4 (1978)

[8] Dayal, U. Schema-mapping problems in database systems. TR-11-79, Center for Research in Computing Technology, Harvard University, Cambridge, 1979

[9] Dayal, U., N.Goodman, and R.H.Katz. An extended relational algebra with control over duplicate elimination. Proc. ACM Symp. on Principles of Database Systems, Los Angeles, March 1982, 117-123

[10] Dayal, U., and Goodman, N. Query optimization for CODASYL database systems. Proc. ACM SIGMOD Conf., Orlando 1982, 138-150

[11] Dayal, U. Query processing in a multi-database system. In [19], 81-108

[12] Finkelstein, S. Common expression analysis in database applications. Proc. ACM SIGMOD Conf., Orlando 1982, 235-245

[13] Frost, R.A. SCHEMAL: Yet another conceptual schema definition language. The Computer Journal 26, 3 (1983) 228-234

[14] Hammer, M., and Chan, A. Index selection in a self-adaptive data base management system. Proc. ACM SIGMOD, Washington D.C. 1986, 1-8

[15] Hwang, H., and Dayal, U. Using the entity-relationship model for implementing multi-model database systems. Proc. 2nd Entity-Relationship Conf., Washington D.C., 1981

[16] Inmon, W.H. Why large on-line relational systems don't (and may not ever) yield good performance. EDP Performance Review (USA), Oct. 1986, Vol.14, No 10, 5-8

[17] Jacobs, B. On database logic. Journal of the ACM 29, 2 (1982) 310-332

[18] Kent, W. Limitations of record-based information models. ACM Trans. on Database Syst. 4, 1 (1979) 107-131

[19] Query processing in database systems. (Kim, Reiner, Batory, Eds.) Springer-Verlag, Berlin 1985

[20] Klausner, A., and N.Goodman. Multirelations - Semantics and Languages. Proc. 11th VLDB Conf., Stockholm 1985, 251-304

[21] Klug, A. Equivalences of relational algebra and relational calculus query languages having aggregate functions. Journal of the ACM 29, 3 (1982) 699-717

[22] Koenig, S., and Paige, R. A transformational framework for the automatic control of derived data. Proc. 7th VLDB Conf., Cannes 1981, 306-318

[23] Larson, P., and Yang, H. Computing queries from derived relations. Proc. 11th VLDB Conf., Stockholm 1985, 259-269

[24] Lindsay, B. et al. A snapshot differential refresh algorithm. Proc. ACM SIGMOD Conf., Washington D.C., June 1986, 53-60

[25] Manola,F., and A.Pirotte. An approach to multimodel database systems. Proc. 2nd Conf. on Databases, Cambridge, England, 1983, 53-75

[26] IEEE Database Engineering, Special issue on query processing. Sep. 1982, (Reiner, Ed.)

[27] Rzeczkowski, W.: Query files as a tool for improving database systems. Ph.D. thesis, Institute of Computer Science Polish Academy of Sciences, 1985, (in Polish)

[28] Rzeczkowski,W., and Subieta,K. Stored queries - a data structure for query optimization. Institute of Coputer Science Polish Academy of Sciences Report 583, Warsaw, May 1986

[29] Schkolnick, M. The optimal selection of secondary indices for files. Information Systems 1 (1975) 141-146

[30] Shipman, D.W. The functional data model and the data language DAPLEX. ACM Trans. on Database Syst. 6, 1 (1981) 140-173

[31] Shmueli, O., and Itai, A. Maintenance of views. Proc ACM SIGMOD Conf., 1984, 240-255

[32] Stonebraker, M., et al. Document processing in a relational database system. ACM Trans. on Office Inf. Syst. 1, 2 (1983) 143-158

[33] Subieta, K. High-level navigational facilities for network and relational databases. Proc. 9th VLDB Conf., Florence 1983, 380-386

[34] Subieta, K. Semantics of query languages for network databases. ACM Trans. on Database Syst. 10, 3 (1985) 347-394

[35] Subieta,K., and Missala, M. Semantics of query languages for the Entity-Relationship Model. Proc. 5th Conf. on Entity-Relationship Approach, Dijon, Nov. 1986, 291-310

[36] Subieta, K. Denotational semantics of query languages. Information Systems 12, 1 (1987)

[37] Subieta, K., and Missala, M. Data manipulation in NETUL. Submitted to 6th Intl. Conf. on Entity-Relationship Approach, New York, 1987

[38] Wong, E. and K. Youssefi: Decomposition - strategy for query processing. ACM Trans. on Database Syst. 1, 3 (1976) 223-241

[39] Young, J. Relational databases - benefits and drawbacks. Data Processing 28, 6 (1986) 312-313

[40] Zlatuska, J. The HIT data model. Data bases from the functional point of view. Proc. 11th VLDB Conf., Stockholm 1985, 470-477