COMMON SUBEXPRESSION ISOLATION IN MULTIPLE QUERY OPTIMIZATION

Matthias Jarke

January 1984

Center for Research on Information Systems
Computer Applications and Information Systems Area
Graduate School of Business Administration
New York University

Working Paper Series

CRIS #71

GBA #84-46(CR)

Published in W. Kim, D. Reiner, D. Batory (eds.), Query Processing in Database Systems, Springer-Verlag, 1985 pp.191-205

COMMON SUBEXPRESSION ISOLATION IN MULTIPLE QUERY OPTIMIZATION

Abstract: The simultaneous optimization of multiple queries submitted to a database system may lead to substantial savings over the current approach of optimizing each query separately. Isolating common subexpressions in multiple queries and treating their execution as a sharable resource are important prerequisites. This chapter presents techniques for recognizing, supporting, and exploiting common subexpressions in record-oriented, relational algebra, domain relational calculus, and tuple relational calculus query representations. It also investigates preconditions that transaction management mechanisms must satisfy to make multiple query optimization effective.

1.0 INTRODUCTION

The joint execution of batches of queries and update operations has been a standard technique in the conventional, record-at-a-time file systems of the sixties and early seventies. However, with the introduction of interactive database systems based on direct access to specific subsets of data, the research focus has changed towards optimizing individual, set-oriented data requests. With few exceptions, the art of multiple query processing has not survived the cultural jump from file to database processing. Recently, however, there has been renewed interest in exploiting the potential advantages of resource sharing in query optimization. Two concepts can be distinguished.

Batching. Sharing the cost of operations by jointly executing multiple queries submitted at approximately the same time is viable in a shared database, where a batch of queries can be composed and executed within reasonable response time limits (a few seconds [BARB83]), or in a non-interactive database programming environment. The sharing objective distinguishes batching from simple parallelism of data access, as investigated, e.g., in [CHES83].

Repetitive queries. (Partially) repetitive queries can share common resources even in a one-user environment if these resources (usually called access paths) are kept over an extended period of time. Speaking in business terms, we have an investment problem: the more queries will use a resource, the more initial investment is justified. Support for repetitive queries on a long-term scale, mostly through indexes, has been the focus of much research on the file system level, but less so for high-level queries.

One of the obstacles preventing a more extensive use of these opportunities in database systems has been uncertainty about what constitutes a sharable resource. This chapter assumes (the evaluation of) common subexpressions in queries to be the sharable resource and investigates methods for isolating and exploiting them. A subexpression is a part of a query that defines an intermediate result used during the process of query evaluation. In the relational framework adopted in this paper, subexpressions are defined in relational calculus or as results of relational algebra operations. For the sake of brevity, relational notations as introduced in [JARK84b] will not be repeated here.

In traditional file systems, where records are retrieved one-at-a-time, a query or update is simply defined by a particular key value. Common subexpressions are characterized by the same key value. Section 2 reviews multiple query optimization for such record-oriented systems. Section 3 motivates and defines more general common subexpressions. Sections 4 through 6 present specific methods for common subexpression representation and analysis in three popular query language environments: relational algebra, domain relational calculus (including languages such as QBE and Prolog), and tuple relational calculus (including languages such as SQL and QUEL). For the latter representation, a database programming language construct, called selector, is used to represent subexpressions and access paths supporting their execution.

Finally, section 7 briefly considers a new research problem resulting from multiple query optimization. If the scope of query optimization is extended beyond transaction boundaries, query evaluation strategies may interfere with concurrency control algorithms, leading to an inefficient overall architecture. The need for global transaction optimization integrates the two hitherto separated research areas of query optimization and concurrency control.

2.0 MULTIPLE QUERY OPTIMIZATION IN RECORD-ORIENTED SYSTEMS

In a traditional file system, each query retrieves at most one record, which is described by its file (relation) name and a unique key value. Many such systems are still in use, e.g., in banking applications or reservation systems, in which each user transaction addresses only one data object (e.g., bank account) at a time. One can represent a request for the record of relation 'rel' with the key value 'keyval' in an array-like notation [SCHM83],

rel[keyval].

For example, if social security number is the key for an employee file, a user may ask for 'employee[115-66-3331]'.

In a multiple query environment, information must be provided to determine, to which query the answer should be delivered. A query can be represented by a record

<userid, timestamp, opcode=read, rel[keyval]>.

Consequently, a batch of queries can be stored in a relation, the so-called 'transaction file'. The timestamps become important when the same user submits the same request several times, for example, because of intervening updates. However, we shall ignore this possibility for the moment and will return to it only in section 7.

Under what conditions is batching advantageous? In a paged random access environment, the main profit stems from clustering accesses to the same physical page. Little is gained by batching non-clustered queries which access different pages. (See [SHNE76] for a quantitative analysis of the worst case of random queries to a large file.) Two queries to a relation obviously access the same page if they request the same key value. Therefore, the transaction file should be grouped by relation names and key values; this is typically best achieved by sorting. Sorting has the side benefit of achieving optimal clustering if the database relation to be accessed is sorted by the same criteria (e.g., organized in some indexed sequential fashion).

If, on the other hand, sequential processing is necessary, batching almost always makes sense. As [SHNE76] demonstrate, the expected savings factor of processing a batch of k queries together rather than separately can be approximated by 1-2/(k+1) for large files. For example, a batch of just five

queries will already lead to savings of about 66.7%, as compared to evaluating each of them separately: each of the five queries, separately processed, will require scanning about 1/2 of the file (for a total of 5/2 file scans), whereas only 5/6 of the file have to be scanned on the average to retrieve five (randomly selected) elements.

The advantages of batching in average processing time per query generally grow with batch size, in particular with the size of clusters (i.e., the number of accesses to the same page). On the other hand, batch size is limited by the maximum response time acceptable for the first queries submitted to the batch, as well as by storage constraints.

3.0 GENERALIZED COMMON SUBEXPRESSIONS

The key-oriented techniques for multiple query processing do not easily generalize to queries retrieving more than one record. In principle, one can decompose a set-oriented query into many record-oriented ones. For multiple query optimization, however, this approach has major disadvantages:

Unless secondary indexes are available, the set of key values for each query is unknown before accessing the database, and hence the comparison of key values cannot be used to determine sharable accesses. One might argue that unknown key values usually require sequential scans, which should make multiple query optimization even more desirable. However, there is now a 'distribution' problem: it is not known in advance, to which of the queries in a batch a certain record will be relevant. Therefore, sharing is limited to the original scan of base data -- none of the intermediate results required for processing complex queries can be shared. This problem arises even if the key set for each query can be enumerated (e.g., because secondary indexes are available).

The solution adopted in this chapter involves access abstraction mechanisms [SCHM83], which reduce the problem of recognizing common physical access requirements to the simpler task of identifying common logical access paths, i.e., subexpressions. Common subexpressions will be used since one can hardly expect two queries to address exactly the same set of tuples as in the record-oriented case.

A few definitions are needed at this point. We define a query as a relation-valued language expression, that is, the evaluation of a query, q, maps a database state into a relation V(q), the value or result of the query. The readset, S(q), of a query is the set of all data to be accessed during the evaluation of q. Note, that S(q) depends on data structures and query evaluation algorithms, whereas V(q) depends only on the state of the database. Let Q = [q1, ..., qn] be a set of queries. Then, a query, c, with non-empty value is called a common subexpression of Q if S(c) is a subset of the intersection of all the S(qi), i = 1, ..., n.

An access path is the value of a query or of a set of queries; access paths are used to support the evaluation of other queries. For example, a secondary index represents the set of results of those queries that ask for all relation elements with a given value in the indexed attribute; the use of the index provides a fast way to process other queries that contain queries on the indexed attribute as subexpressions. Often, access paths are stored in a specific representation form to avoid redundancy and reduce maintenance problems. However, the special representation is usually invisible to the user who just experiences better performance for certain queries. A language construct for the abstract representation of access paths based on this observation will be introduced in section 6.

An access path defined by a query, ap, is applicable to the evaluation of a subexpression, se, if S(se) is a subset of S(ap), or -- in other words -- if the selection predicate of se implies the selection predicate of ap. In this case, we also say somewhat loosely that ap is applicable to a query containing se.

These definitions are very general. The detection of common subexpressions or of the applicability of access paths may be computationally intractable or even undecidable if arbitrary subexpressions are considered. Of course, in a finite database, one can always detect common subexpressions 'after the fact', i.e., by tracing the query execution. But that does not help in ex-ante query optimization where one would like to analyze the query rather than its value. Most published procedures follow a two-step heuristic: (a) decompose each query into a (partially ordered) set of 'suitable' subexpressions, and (b) identify common subexpressions and applicable access paths. What constitutes a 'suitable' subexpression depends on the query language. The next sections discuss three popular representation forms for relational expressions, i.e., algebra, domain calculus, and tuple calculus.

As a running example, we shall use a simplified version of the infection control database in a hospital. Such databases are used for tracing the flow of infecting organisms in case of epidemics, and for identifying persons under risk of infection. A patient is characterized by the ward he/she is located in, by the day of surgery, by observed symptoms indicating an infection, and by the quantity of pathogenic organisms isolated from certain sites of the human body (e.g., surgical wounds, the blood, or the respiratory tract). Employees are characterized by their status (e.g., doctor, nurse, administration) and assigned ward, as well as by their assignment to operating teams on certain days. The schema consists of seven relations:

```
patient  (pname, ward)
isolated (pname, organism, site, qty)
observed (pname, symptom)
relevant (symptom, site)
employee (ename, status, ward)
surgery  (pname, day)
opteam   (ename, day)
```

4.0 COMMON SUBEXPRESSION ANALYSIS IN RELATIONAL ALGEBRA

In relational algebra notation, a single algebra operation is the natural unit of interest. A more complex subexpression corresponds to a sequence of operations. Since an operation is meaningful only if all of its inputs are known, common subexpression detection is a bottom-up procedure, collapsing common subtrees in the operator tree for the algebra query. This procedure, introduced in [HALL76] for single query optimization, can be extended to multiple query optimization by considering a set of operator trees.

As an example, consider the queries: "which wards are members of the Monday operating team assigned to?", and: "what doctors were on the Monday operating team?" Figure 4-1(a) demonstrates collapsing common subtrees for operator trees corresponding to query formulations which follow the heuristic of moving restrictions as far down as possible [SMIT75]. Only the common restriction of opteam by 'day=monday' can be shared in this case. This shows that an unfortunate sequencing of operations may prevent the detection of larger common subexpressions. In Figure 4-1(b), the restriction on employee in the second query has been moved upward beyond the join, such that the sharable subexpression includes the join operation. It is not easy to find a query standardization that automates such algebra transformations. [HALL76] presents some heuristics. In other cases, collapsing small common subexpressions in the beginning of the procedure may prevent the detection of larger ones later on.

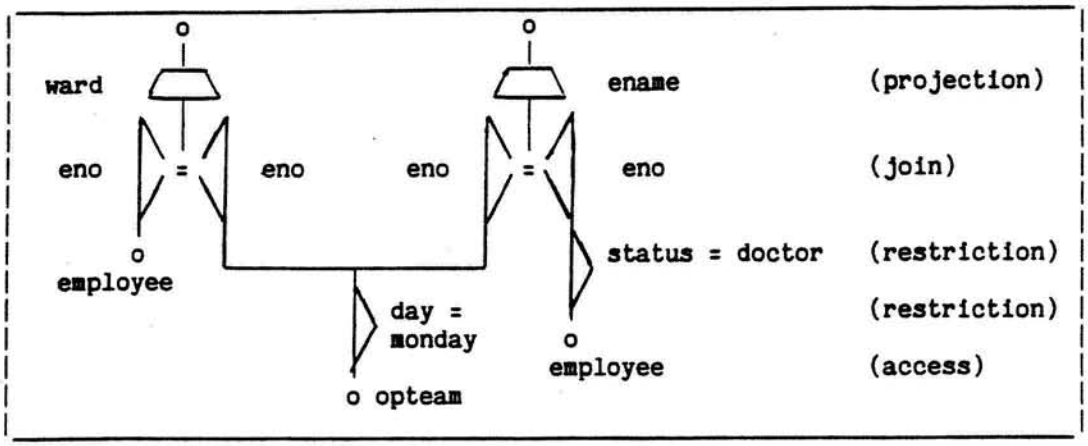

Figure 4-1(a): A simple common algebra subexpression

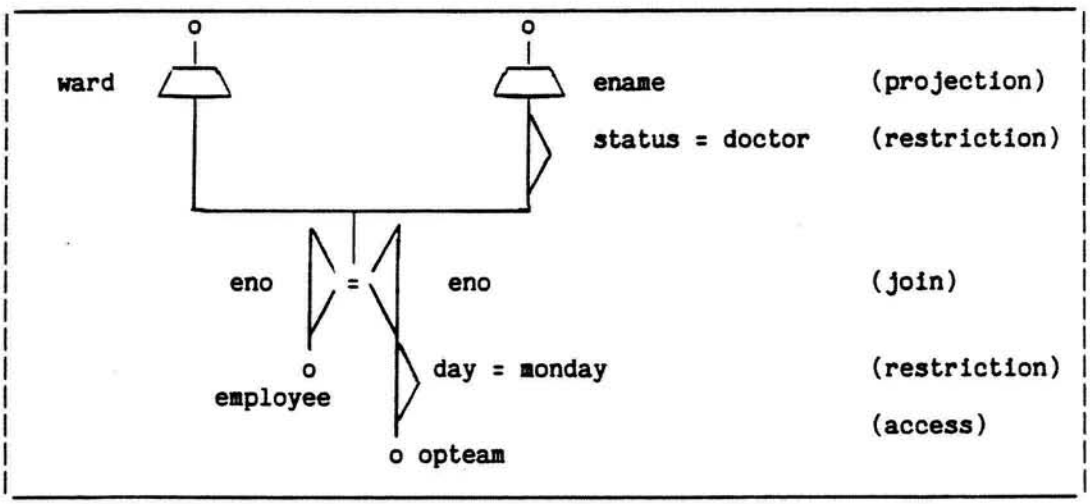

Figure 4-1(b): A better common subexpression

5.0 COMMON SUBEXPRESSION ANALYSIS IN DOMAIN RELATIONAL CALCULUS

Predicate calculus representations offer more control over the level of abstraction, on which common subexpressions can be defined. Domain calculus is used in connection with logic programming, for instance, if Prolog is used as a database language [KOWA81]. In this representation, variables range over attribute domains, and relations are represented as predicates. Relation values are defined as sets of assertions that look like the example schema provided in section 3. An arbitrary hierarchy of parameterized and maybe recursive view definitions can be superimposed on the original database relations through the use of Horn clauses with variables. For example, in the infection database, Prolog definitions of the concepts of an infected patient, and of personnel contact with a patient look as follows:

A patient is said to be infected at a certain body site if pathogenic organisms have been isolated there or relevant symptoms have been observed.

```
infected(Pname, Site) :- isolated(Pname, Organism, Site, Qty).
infected(Pname, Site) :- observed(Pname, Symptom),
                         relevant(Symptom, Site).
```

A employee is said to have been in contact with a patient, if he/she was either on an operating team the same day the patient had an operation, or he/she is a nurse assigned to the patient's ward.

```
contact(Ename, Pname) :- opteam(Ename, Day),
                         surgery(Pname, Day).
contact(Ename, Pname) :- employee(Ename, nurse, Ward),
                         patient(Pname, Ward).
```

In these parameterized view definitions, variable names begin with capital letters and constants begin with lower-case letters. Variables are used to store parts of the relations implicitly, or to represent subsets of stored relations. For example, an assertion

```
relevant(fever, Site).
```

means that fever is a relevant symptom for infections at any site, since the variable 'Site' can assume arbitrary values. A concatenation of predicates by a comma indicates AND-connection; the repetition of a left-hand side predicate indicates that one or the other definition may apply, i.e., OR-connection of the right-hand sides.

A query is an expression consisting of the symbol ':-', followed by AND/OR-connected predicates which refer either to views or directly to base relations. Consider first the case where there are no recursive view definitions applying to the set of queries. Following the two-step heuristic mentioned in section 3, each query is first standardized into disjunctive normal form, i.e., into a set of conjunctive queries to base relations. More than one submitted request will just result in a larger set of conjunctive queries.

To simplify subsequent steps, the query set is then partitioned into components such that the readsets of queries in different components are disjoint. The simplest way to guarantee this is to partition the set of queries by the relations they access [GRAN81]. Within each component, common subexpressions among the queries are identified as follows.

Each query within a component is a conjunction of predicates, where each predicate corresponds to access to one relation. Common subexpressions have to contain at least common predicate (= relation) names. Thus, for each pair of queries, we just have to test if pairs of predicates with common relation names are equivalent. In fact, we can test for containment, rather than for exact equivalence, and use the result of one subexpression as an access path for evaluating the other subexpression [GRAN81]. (Similar methods have also been used to optimize tableaux for non-conjunctive queries [SAGI80].)

To illustrate the above procedure, consider the three view queries:

R1. "what patients with Monday surgery have wound infections?"

```
:- infected(Who, wound), surgery(Who, monday).
```

R2. "what patients had the same organisms isolated as Smith?"

```
:- isolated(smith, Commorg, Site1, Qty1),
   isolated(Pat,   Commorg, Site2, Qty2).
```

R3. "who was in contact with wound-infected patients?"

```
:- infected(Infpat, wound), contact(Pers, Infpat).
```

Using the view definitions given above, the translation of these view queries into queries to the base relations given in section 3 yields the seven conjunctive queries listed below. Q1 and Q2 come from R1, Q3 from R2, and the remaining four queries are derived from R3.

```
Q1. :- isolated(Who, Organism, wound, Qty), surgery(Who, monday).
Q2. :- observed(Who, Symptom), relevant(Symptom, wound),
       surgery(Who, monday).
Q3. :- isolated(smith, Commorg, Site1, Qty1),
       isolated(Pat,   Commorg, Site2, Qty2).
Q4. :- isolated(Infpat, Organism, wound, Qty),
       surgery(Infpat, Day), opteam(Pers, Day).
Q5. :- isolated(Infpat, Organism, wound, Qty),
       patient(Infpat, Ward), employee(Pers, nurse, Ward).
Q6. :- observed(Infpat, Symptom), relevant(Symptom, wound),
       surgery(Infpat, Day), opteam(Pers, Day).
Q7. :- observed(Infpat, Symptom), relevant(Symptom, wound),
       patient(Infpat, Ward), employee(Pers, nurse, Ward).
```

All seven queries form one component and are therefore candidates for simultaneous query optimization. Common subexpressions can be identified by comparing predicates with equal names: (a) The first predicate of Q1 is equivalent to the first predicates of Q4 and Q5, and its value is a subset of the value of the second predicate of Q3; (b) Similarly, the conjunction of the first two predicates of Q2 also appears in Q6 and Q7; (c) the surgery predicates in Q1 and Q2 are identical; (d) the second rows of queries Q4 and Q6, as well as of Q5 and Q7 are sharable.

Could we have detected the common subexpressions more efficiently? Yes; we could have predicted many of the common subexpressions by comparing the original view queries directly. For example, that R1 and R3 have a lot in common is obvious from the common predicate 'infected(X, wound)'. This observation could have reduced the set of queries to be investigated. A more systematic analysis of this view-oriented approach will be presented in section 6.3.

An additional opportunity for multiple query optimization by common subexpression analysis, not found in the other two representations used in this chapter, presents itself in the evaluation of queries that are defined on recursive views. Assume that our infection control database contains a base relation 'met(Gname, Sname)' which describes the fact that an individual named Gname met a potential carrier of pathogenic organisms, named Sname. Note, that the Sname--Gname relationship must be hierarchical (i.e., it may not contain cycles) to be evaluable with the depth-first approach of Prolog. We can now define the risk that organisms have been transmitted from a person another one recursively as

```
risk(Pname, Gname) :- met(Gname, Pname),
                      infected(Pname, Anysite).
risk(Pname, Gname) :- met(Gname, Intermediate),
                      risk(Intermediate, Pname).
```

One can now ask for persons exposed to risk by a particular patient, or for the patients putting a particular person at risk. In both cases, the query evaluation generates a sequence of queries, in which each subsequent query contains the previous one as a subquery. For example, the query,

```
:- risk(smith, Gname),
```

is evaluated by the sequence of non-recursive database queries

```
:- met(Gname, smith), infected(smith, Site).
:- met(Gname, C1),
   met(C1, smith), infected(smith, Site).
:- met(Gname, C1),
   met(C1, C2), met(C2, smith), infected(smith, Site).
etc.
```

Each query appears as the second part of the following one; vice versa, the evaluation of each query can use the result of the previous one [JARK84a]. Additionally, it is possible to rephrase recursive queries in a manner that keeps the size of intermediate results small [MARQ84]. A detailed discussion of these approaches, as well as of a large number of additional problems related to recursive query processing, is beyond the scope of this paper; more material can be found in [MINK83] and [HENS84].

6.0 COMMON SUBEXPRESSION ANALYSIS IN TUPLE CALCULUS

6.1 Nested Expressions And The Selector Language Construct

In the tuple relational calculus representation as used in [JARK84b], an interesting set of subexpressions can be generated using range-nested expressions. Recall that the following transformations can be applied to generate nested expressions where p1 and p2 are predicates [JARK83].

```
[EACH r IN rel: p1 AND p2]     <==> [EACH r IN [EACH r IN rel: p1]: p2]
 SOME r IN rel (p1 AND p2)     <==>  SOME r IN [EACH r IN rel: p1] (p2)
  ALL r IN rel (NOT(p1) OR p2) <==>   ALL r IN [EACH r IN rel: p1] (p2)
```

Here, we shall not deal with universally quantified variables. Therefore, we can assume without loss of generality that all queries are conjunctive. Each of the inner nestings represents a potential common subexpression. For the definition of access paths, there is a need to abstract from specific subexpressions. A language construct called selector [MALL84] serves as an abstract representation of subexpressions and their access paths. Let

```
EACH r IN rel:  p(r, s1, ..., sm, t1, ..., tn)
```

be a relational expression where p is a well-formed formula of the relational calculus in which quantified variables t1, ..., tn appear. The s1, ..., sm are formal parameters representing constants in terms of the selection predicate. A selector sp representing the subexpression can be declared in a function-like fashion [1]:

```
SELECTOR sp(s1, ..., sm) FOR rel;
BEGIN
  EACH r IN rel: p(r, s1, ..., sm, t1, ..., tn)
END;
```

Selectors are used in selected variables that appear in relation-valued expressions, using an array-like notation

```
rel[sp(S1, ..., Sm)],
```

where the Si are actual parameter values. For example, the notation 'rel[keyval]', which was introduced in section 2 for identifying single tuples in relations, uses an implicitly defined key selector,

```
SELECTOR sk(keyval) FOR rel;
BEGIN
 EACH r IN rel: r.key = keyval
END;
```

[1] The actual selector definition originally introduced in [MALL84], [SCHM83] refers to relation types instead of relation variables. In the query optimization context considered in this paper, however, the simpler notation presented here is sufficient.

Selectors have two uses in multiple query optimization: naming common subexpressions and supporting their evaluation. From a language viewpoint, a selector can be regarded as the definition of a (possibly parameterized) view, similar to the ones we saw in the domain calculus examples. However, from a system's viewpoint, a selector can also be perceived as the abstract representation of an access path, that -- if provided with appropriate parameters -- returns a set of relation elements. Following the definitions given in section 3, we say that a selector is applicable to a subquery, if actual parameters S1, ..., Sm can be found, such that the predicate of the subquery implies the selection expression of the selector, with S1, ..., Sm substituted for the formal parameters. Furthermore, we say that a selector is supported if an actual physical access path has been created for all queries represented by the selector definition.

Since testing applicability is undecidable in general first-order predicate calculus, and computationally intractable even in some cases where it is decidable [ROSE80], we shall explore several classes of selectors (and consequently of nested subexpressions), for which efficient tests or good heuristics are known.

Selectors can be classified by the values of m and n in the above definition. First, selectors without parameters (m=0) will be investigated; they correspond to traditional database views. This discussion will be subdivided into the cases of extended range expressions (n=0, section 6.2) and general nested expressions (n>0, section 6.3). Afterwards, selectors with parameters (m>0) will be analyzed (section 6.4). This discussion will be brief since one part of it is covered by the other subsections, another part is covered by the literature on index selection, and the remainder is largely unresearched.

6.2 Common Extended Range Expressions

The early execution of one-variable operations, such as restriction and projection, is a well-known heuristic for query transformation [SMIT75], [WONG76]. In the relational calculus framework, this has been modeled by introducing nested expressions that extend the range definition of variables from simple relation names to relational expressions (i.e., queries) that contain restrictive terms over the base relation [JARK82]. Consider the following three queries and their disjunctive prenex normal form representation in Pascal/R [SCHM80]:

R4. "what analyses yielded more than 1000 organisms/ml?"

```
[EACH i IN isolated: i.qty > 1000]
```

R5. "what patients of which wards had at least 2000 organisms isolated?"

```
[EACH p IN patient: SOME i IN isolated
     (p.pname = i.pname AND i.qty >= 2000)]
```

R6. "list doctors in wards with patients who had fever in connection with the isolation of at least 2000 organisms in wounds."

```
[EACH e IN employees:
 SOME o IN observed SOME p IN patient SOME i IN isolated
  (e.ward = p.ward AND p.pname = o.pname AND i.pname = p.pname
   AND o.symptom = fever AND i.qty >= 2000 AND i.site = wound)]
```

Applying the nesting transformation given in section 6.1 converts the two queries R5 and R6 to:

```
R5'. [EACH p IN patient:
      SOME o IN [EACH i IN isolated: i.qty >= 2000]
          (o.pname = p.pname)]

Q6'. [EACH d IN [EACH e IN employees: e.status = doctor]:
      SOME f IN [EACH o IN observed: o.symptom = fever]
       SOME p IN patient
        SOME w2 IN [EACH i IN isolated:
                          i.qty >= 2000 AND i.site = wound]
       (w2.pname = p.pname AND p.pname = f.pname AND p.ward = e.ward)]
```

Nested expressions are conveniently represented by object graphs [FINK82], [JARK83]. Each inner expression corresponds to a node, and each join term corresponds to an edge. Thus, while R4 is represented by only one node, R5' and R6' are represented by trees with two respectively four nodes (figure 6-1).

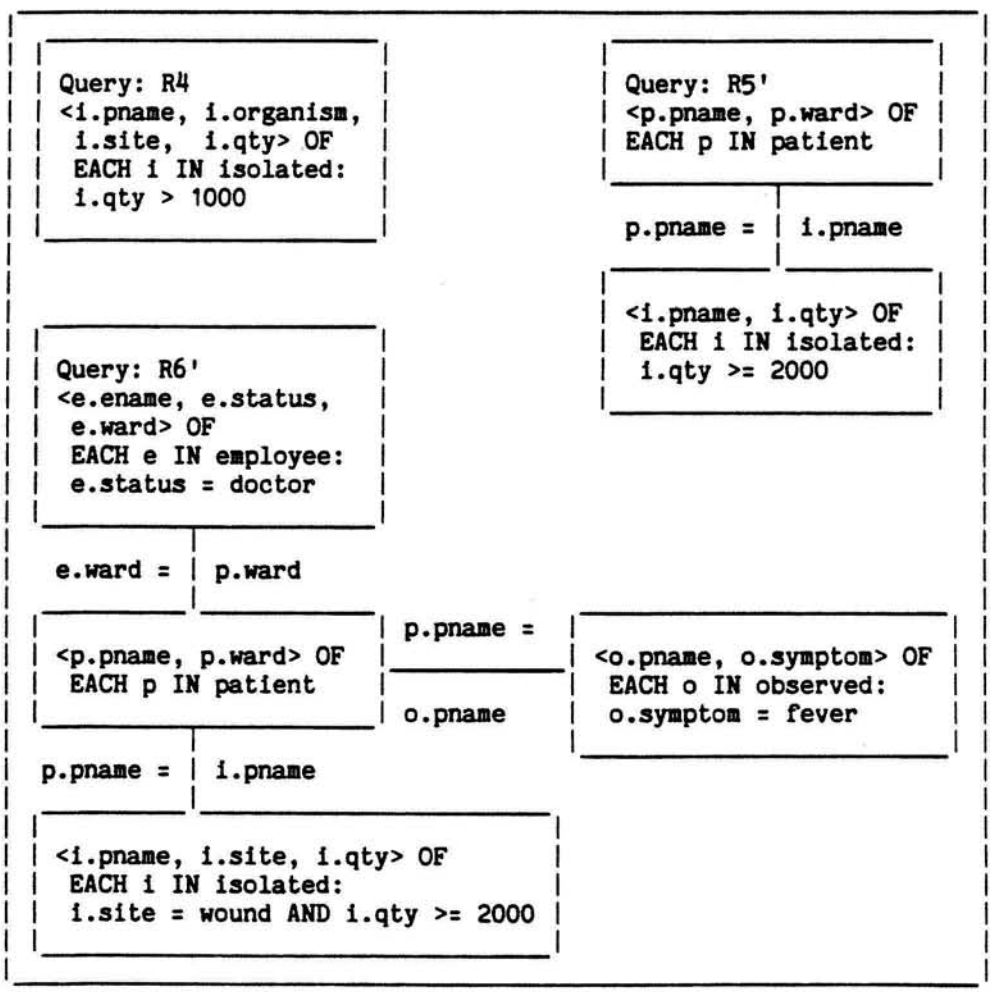

Figure 6-1: Query graphs for extended range expressions

Users often refer to previous queries; for example, the infection control nurse might 'zoom in' on the objects of interest by issuing a sequence of queries R4 to R6. [FINK82] therefore suggests storing object graph representation and query value -- V(q) in the notation of section 3, not s(q)! -- of certain queries in a buffer to be used for evaluating subsequent queries.

Consider first single-node queries such as R4. Such queries are of interest, not only because they occur frequently, but also because most indexes in database systems can be regarded as collections of query results of this type. The stored value of a one-variable query can be used directly as an access path for a new one if two preconditions are satisfied: (a) the selection expression of the new query implies the one of the stored query; (b) the output attributes of the stored query are a superset of the attributes appearing anywhere in the new query. For example, R4 can be used directly for evaluating the second node of R5 since 'i.qty >= 2000' implies 'i.qty > 1000' and both attributes of the relation 'isolated' appearing in R5 also appear in the output of R4.

When condition (b) is violated, using the stored query may still be justified, but a 'backjoin' is required between the stored query result and the base relation to recover missing attribute values. This would happen, for instance, if R4 only requested patient names. The join with a stored relation may be cost-justified if there are fast access paths to perform the backjoin. The backjoin problem may also occur in multi-node queries. The set of patients retrieved in R5' is clearly a superset of the ones retrieved for R6'. However, since the site attribute does not appear in the output of R5', a backjoin of the query value V(R5') with the relation 'isolated' is required.

[FINK82] does not perform general implication tests for multi-node queries (such tests are studied in [MUNZ79] and [ROSE80]) but uses a heuristic, which not necessarily detects all stored query results usable for a given query. In addition, the algorithm also determines eventual backjoins, and compares their costs to the savings expected from using the old query value as an access path. However, [FINK82] does not assume exact foresight of the query optimizer in terms of what future queries to expect.

6.3 Common Nested Expressions

In processing batches of queries, such precise knowledge does exist. Common subexpressions can be supported in a pre-planned fashion by defining appropriate selectors and creating physical access paths supporting them. For example, the batch R4, R5, R6 of the previous subsection could be supported by the selector

```
S1.  SELECTOR org1000 FOR isolated;
     BEGIN
      EACH i IN isolated: i.qty > 1000
     END;
```

This reduces R4 to a single access to an existing query result via the selector

```
isolated[org1000]
```

and improves the performance of the other queries accordingly. However, it is obvious that the limitation of selector definitions to one-variable expressions does not permit the level of sophistication required for multi-relation queries. This subsection will therefore consider more general selector definitions. For example, with the additional definition of a selector

```
S2.  SELECTOR pat2000 FOR patient;
     BEGIN
      EACH p IN patient: SOME i IN isolated
                         (i.qty >= 2000 AND i.pname = p.pname)
     END;
```

query R6' can be replaced by (notice the backjoin between pi and wi)

```
[EACH doc IN [EACH e IN employees: e.status = doctor]:
 SOME fev IN [EACH o IN observed: o.symptom = fever]
  SOME pi IN patient[org2000]
   SOME wi IN [EACH i IN isolated: i.site = wound]
  (wi.pname = pi.pname AND pi.pname = fev.pname AND pi.ward = doc.ward)]
```

Even this notation is not completely satisfactory since it does not show that one selector can be used to improve the evaluation of another one. We therefore introduce several levels of nesting in selectors.

```
S2'. SELECTOR pat2000 FOR patient : ...;
     BEGIN
      EACH p IN patient: SOME i IN isolated[org1000]
                              (i.qty >= 2000 AND i.pname = p.pname)
     END;
```

Many selectors could be defined for supporting any given query. One can generate these selectors by applying the general range nesting transformations given in section 6.1. In terms of the object graph representation (Figure 6-1), multi-variable selectors can be defined for any subgraph that does not contain the target nodes and is connected to the rest of the query graph by a single edge. For example, in query R6', we could define one selector containing the patient/observed subgraph, one containing the patient/isolated subgraph, or one that contains all three nodes.

All queries in a batch can be described by partially ordered sets of selectors, using a uniform naming scheme for equivalent selectors. The system can then identify common selectors among queries. This looks very similar to the algebra approach of section 4. An important difference is that it is not necessary to trace all possible sequences of operations if other means exist for establishing equivalence among subexpressions.

A similar structure has been proposed as a logical access path schema for database design [ROUS82b]. Roussopoulos introduces an object graph that is the exact complement to an algebra operator graph; that is, the nodes represent the results of algebraic operations and groups of edges the operations themselves. [ROUS82b] presents algorithms similar to [HALL76] but goes further by assigning a weight to each node, based on the frequency of reference to the corresponding selector in a set of queries. The higher the weight, the more profitable is the creation of a special physical access path to support the selector. The latter method is called "view indexing" in [ROUS82a].

6.4 Common Query Structures

The disadvantage of access paths defined through parameter-free selectors as described in the previous two subsections is that they essentially represent only one (sub)query. The usual understanding of indexes is quite different: the exact query is defined by specifying a certain parameter value. For example, most users of the infection control database may be interested only in one type of infecting organism at a time but this type may differ from query to query. Therefore, it pays to define a selector corresponding to a secondary index:

```
SELECTOR the-organism(ORG) FOR isolated;
BEGIN
 EACH i IN isolated: i.organism = ORG
END;
```

A query requesting wards where Klebsiella bacteria were found would then be converted to something like:

```
[<kp.ward> OF EACH kp IN patient:
              SOME k IN isolated[the-organism('klebsiella')]
                  (k.pname = kp.pname)]
```

While secondary indexes such as this one are available in most database systems, the selector definition can be more general. For example, if one is always interested in the patients from whom an organism was isolated in certain quantities rather than in the isolation itself, one can define and support a selector, the definition of which spans more than one relation.

```
SELECTOR with-org(ORG, QMIN, QMAX) FOR patient;
BEGIN
 EACH p IN patient: SOME i IN isolated
     (p.pname = i.pname AND i.organism = ORG AND
      i.qty >= QMIN AND i.qty <= QMAX)
END;
```

Obviously, definition and maintenance of such a selector require more sophisticated data structures and algorithms than conventional one-dimensional indexes [NIEV84].

The detection of subexpressions that are supportable by a particular parameterized selector hardly differs from the non-parameterized case; in all places where the formal parameter appears in the selector definition, any domain value may appear in the corresponding position of the query to make the selector applicable.

On the other hand, the problem of choosing a good set of parameterized selectors to be supported by physical access paths is much more difficult than for non-parameterized ones. For single-attribute selectors, and even for general one-variable selectors, the literature on index selection (see [MARC84]) applies. For multiple variable selectors, however, one faces the double problem of choosing a good level of abstraction, and of deciding which query conditions should be parameterized. No solution to this problem is known as of this writing, but several heuristics based on the range nesting procedure of [JARK83] are under study.

7.0 MULTIPLE QUERY OPTIMIZATION AND TRANSACTION MANAGEMENT

When we reviewed batched query processing in record-oriented systems, we excluded update operations from the discussion. In reality, this separation often cannot be accepted; queries and updates may be combined in transactions submitted by the same user or by multiple users. This may lead to problems like inconsistent reads or lost updates which have to be prevented by concurrency control mechanisms [ESWA76].

The only requirement supported by almost all concurrency control methods is serializability, i.e., the outcome of a set of concurrently running transactions has to be equivalent to that of some serial execution of the transactions. Therefore, read and write accesses can follow each other in arbitrary sequence if each transaction concerns only one access to one record. Alternatively, record-oriented file systems often combine batched update with interactive retrieval; the file system is locked completely during the update process. However, certain backout mechanisms for erroneous update transactions have to be provided even in this case [ARDI79].

Once more, consideration of more complex common subexpressions complicates the problem. This section summarizes additional research questions that have to be answered if multiple query optimization is to be integrated into an transaction processing environment that includes interactive update operations.

Consider first a one-user environment (or one in which each user transaction is completely shielded from the others by exclusive locking mechanisms). In such an environment, multiple query optimization is possible only within transactions which consist of more than one query. [HEVN81] studied the influence of general programming language constructs, such as conditional statements and loops, on the composition of query sets to be optimized jointly. As a simple example, consider the following program:

```
IF cond1 THEN Q1; Q2
         ELSIF cond2 THEN Q2; Q3
                     ELSE Q1; Q3
```

Although each query appears twice in the program, any of them will be executed at most once. Creating sophisticated access paths, cost-justified only by multiple usage of any one query, would be a waste of resources.

A second problem in the one-user context is addressed in [KIM81]: the interference of update operations with queries. If the value of a subexpression is changed, the value of queries before and after the update operation will be different. Common subexpressions are defined in [KIM81] on the relation level. As an example, consider a database program that works with queries Qi and altering operations Ai on two relations, R1 and R2:

```
Q1.R1;
A1.R1; A2.R2;
Q2.R1 (Q3.R2);
Q4.R2
```

One can optimize Q3 and Q4 simultaneously but not Q1 and Q2. It is, however, feasible to optimize Q1 and the read part of A1 jointly. Notice that it is quite possible that the actual write set of A1 may be disjoint from the read sets of Q1 and Q2; a detailed analysis of common subexpressions going below the relation level may confirm this.

This argument carries over to the multi-user case. It is well known in concurrency control theory that finer granularity of locks can result in more concurrency [GRAY75], [RIES77]. To identify potential conflicts on a subrelation level, the first requirement is therefore that query optimization and concurrency control mechanisms use the same language, i.e., that predicative concurrency control methods are employed [ESWA76], [REIM83]. Moreover, the focus of multiple query optimization is on sharing rather than just parallelism of data access. Therefore, the introduction of multiple query optimization on a global level creates new problems of integrating query optimization and concurrency control mechanisms.

In the locking approach to concurrency control, shared locks have been proposed to increase concurrency [KEDE83] but sharing is not supported actively. On the other hand, optimistic methods [KUNG81], [REIM83] assume that conflicts are unlikely to occur and validate transactions ex-post; hence, full global query optimization is possible. However, a whole batch of transactions may have to be repeated if one of them cannot be validated.

More flexible scheduling strategies combining locking and optimistic methods are required [BRAE83]. For example, in order to create a safe environment for multiple query optimization, one can schedule read transactions pessimistically (i.e., use locking) while allowing altering transactions to run optimistically.

In addition, we are investigating strategies on the query optimization side that limit the damage done by transaction abortion without sacrificing correctness. Finally, the use of multiversion databases [BERN83] is considered in this context. A radical solution based on multiple versions completely decouples read and write transactions. It allows queries to access only data versions guaranteed to be too old to have been created by any currently active transaction. However, such an extreme solution is hardly acceptable in general.

8.0 SUMMARY AND CONCLUSION

Extending multiple query processing from record handling systems to set-oriented database systems is based on the recognition of and support for common subexpressions in queries. The review of three different language types for relational queries showed that common subexpression analysis is not very hard if only one-relation subexpressions are considered and appropriate tools exist for isolating them. Common multi-relation subexpressions can only be addressed in a heuristic manner. More powerful heuristics, made possible through the introduction of programming language abstractions, allow more than detecting common subexpressions in a purely bottom-up fashion. Where simplistic radical solutions for concurrency control are not acceptable, multiple query optimization requires integrating the hitherto separate areas of query optimization and concurrency control, with the final goal of developing a unified optimization concept for database implementation.

Acknowledgments

The author is grateful to Richard Braegger, Henning Eckhardt, Manuel Reimer, Juergen Koch, and Joachim Schmidt for stimulating discussions, and to the editors for their detailed comments on a previous draft of this paper.

REFERENCES

[ARDI79] Arditi, J. "An optimized backout mechanism for sequential updates", Proceedings 5th VLDB Conference, Rio de Janeiro 1979, 147-154.

[BARB83] Barber, R.E., Lucas, H.C. "System response time, operator productivity, and job satisfaction", Commmunications of the ACM 26, 11 (1983), 972-986.

[BERN83] Bernstein, P.A., Goodman, N. "Multiversion concurrency control - theory and algorithms", ACM Transactions on Database Systems 8, 4 (1983), 465-483.

[BRAE83] Braegger, R.P., Reimer, M. "Predicative scheduling: integration of locking and optimistic methods", Bericht 53, ETH Zuerich 1983.

[CHAK82] Chakravarthy, U.S., Minker, J. "Processing multiple queries in database systems", IEEE Database Engineering 5, 3 (1982), 38-43.

[CHES83] Chesnais, A., Gelenbe, E., Mitrani, I. "On the modelling of parallel access to shared data", Communications of the ACM 26, 3 (1983), 196-202.

[ESWA76] Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L. "The notions of consistency and predicate locks in a database system", Communications of the ACM 19, 11 (1976), 624-633.

[FINK82] Finkelstein, S. "Common expression analysis in database applications", Proceedings Acm-SIGMOD Conference, Orlando 1982, 235-345.

[GRAN81] Grant, J., Minker, J. "Optimization in deductive and conventional relational database systems", in Gallaire, H., Minker, J., Nicholas, J.-M. (eds.), Advances in Database Theory, Plenum, New York 1980, 195-234.

[GRAY75] Gray, J.N., Lorie, R.A., Putzolu, G.R. "Granularity of locks in a shared database", Proceedings 1st VLDB Conference, Framingham, Mass. 1975, 428-451.

[HALL76] Hall, P.A.V. "Optimization of single expressions in a relational data base system", IBM Journal of Research and Development 20, 3 (1976), 244-257.

[HENS84] Henschen, L.J., Naqvi, S.A. "On compiling queries in recursive first-order databases", Journal of the ACM 31, 1 (1984), 47-85.

[HEVN81] Hevner, A.R., Yao, S.B. "Transaction optimization on a distributed database system", Technical Report HR-81-259, Honeywell Corporate Computer Science Center, Bloomington, MN, 1981.

[JARK82] Jarke, M., Schmidt, J.W. "Query processing strategies in the Pascal/R relational database management system", Proceedings ACM-SIGMOD Conference, Orlando 1982, 256-264.

[JARK83] Jarke, M., Koch, J. "Range nesting: a fast method to evaluate quantified queries", Proceedings ACM-SIGMOD Conference, San Jose 1983, 196-206.

[JARK84a] Jarke, M., Clifford, J., Vassiliou, Y. "An optimizing Prolog front-end to a relational query system", Proceedings ACM-SIGMOD Conference, Boston 1984.

[JARK84b] Jarke, M., Koch, J., Schmidt, J.W. "Introduction to query processing", this volume.

[KEDE83] Kedem, Z.M., Silberschatz, A. "Locking protocols: from exclusive to shared locks", Journal of the ACM 30, 4 (1983), 787-804.

[KIM81] Kim, W. "Query optimization for relational database systems", IBM Technical Report RJ3081, San Jose 1981.

[KOWA81] Kowalski, R. "Logic as a database language", Imperial College, London 1981.

[MALL84] Mall, M., Reimer, M., Schmidt, J.W. "Data selection, sharing, and access control in a relational scenario", in Brodie, M., Mylopoulos, J., Schmidt, J.W. (eds.): On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages, Springer 1984, 411-436.

[MARC84] March, S.T. "Physical database design: techniques for improved database performance", this volume.

[MARQ84] Marque-Pucheu, G., Martin-Gallausiaux, J., Jomier, G. "Interfacing Prolog and relational data base management systems", in Gardarin, G., Gelenbe, E. (eds.), New Applications of Data Bases, Acadamic Press, to appear 1984.

[MINK83] Minker, J., Nicholas, J.-M. "On recursive axioms in deductive databases", Information Systems 8, 1 (1983), 1-13.

[MUNZ79] Munz, R., Schneider, H.-J., Steyer, F. "Application of sub-predicate tests in database systems", Proceedings 5th VLDB Conference, Rio de Janeiro 1979, 426-435.

[NIEV84] Nievergelt, J., Hinterberger, H., Sevcik, K.C. "The grid file: an adaptable, symmetric multi-key file structure", ACM Transactions on Database Systems 9, 1 (1984), 38-71.

[REIM83] Reimer, M. "Solving the phantom problem by predicative optimistic concurrency control", Proceedings 9th VLDB Conference, Florence 1983, 81-88.

[RIES77] Ries, D.R., Stonebraker, M. "Effects of locking granularity in a database management system", ACM Transactions on Database Systems 2, 3 (1977), 233-246.

[ROSE80] Rosenkrantz, D.J., Hunt, H.B. "Processing conjunctive queries and predicates", Proceedings 6th VLDB Conference, Montreal 1980, 64-72.

[ROUS82a] Roussopoulos, N. "View indexing in relational databases", ACM Transactions on Database Systems 7, 2 (1982), 258-290.

[ROUS82b] Roussopoulos, N. "The logical access path schema of a database", IEEE Transactions on Software Engineering SE-8, 6 (1982), 563-573.

[SAGI80] Sagiv, Y., Yannakakis, M. "Equivalences among relational expressions with the union and difference operators", Journal of the ACM 27, 4 (1980), 633-655.

[SCHM80] Schmidt, J.W., Mall, M. "Pascal/R Report", Report 66, Fachbereich Informatik, University of Hamburg 1980.

[SCHM83] Schmidt, J.W., Mall, M. "Abstraction mechanisms for database programming", Proceedings ACM-SIGPLAN Symposium on Programming Languages in Software Systems, San Francisco 1983, SIGPLAN Notices 18, 6 (1983).

[SHNE76] Shneiderman, B., Goodman, V. "Batched searching of sequential and tree structured files", ACM Transactions on Database Systems 1, 3 (1976), 268-275.

[SMIT75] Smith, J.M., Chang, P.Y.T. "Optimizing the performance of a relational algebra database interface", Communications of the ACM 18, 10 (1975), 568-579.

[WONG76] Wong, E., Youssefi, K. "Decomposition - a strategy for query processing", ACM Transactions on Database Systems 1, 3 (1976), 223-241.