Common Expression Analysis in Database Applications[1]

Sheldon Finkelstein[2]

Computer Science Department
Stanford University
Stanford, California 94305

Abstract

Independent optimization of database requests overlooks potential savings which can be achieved when they are optimized collectively. An intuitive model for queries called the ***query graph*** supports common expression detection for optimization of a stream of requests. We describe how ad hoc query processing can be improved using intermediate results and answers produced from earlier queries, without significantly impacting processing costs when no common expressions are found. We have written a Pascal program, *COMMON*, which implements a variation of the algorithm which we describe.

1. Introduction

The great advantage of the relational model of database structures is its *physical independence*. Requests specify what information is desired, not how to obtain it. This enables applications to be independent of the secondary access paths, such as indexes, links, and hashed access paths. Even the structure of the relations themselves can be hidden from the user, through the mechanism of views, so applications can be virtually independent of the physical structure of the database. Request *optimizers* make decisions about how to navigate through the maze of access paths, correctly and efficiently, based on a statistical description of the relations, and a cost model of request execution. These decisions may be better than those made by the average user, since the system can (a) estimate all the different possibilities and compare them, and (b) react to changes in the physical structures, without recoding.

[1] This work is part of the Knowledge Base Management Systems Project, under contract #N00039-82-G-0250 from the Defense Advanced Research Projects Agency of the United States Deparment of Defense. The views and conclusions contained in this document are those of the authors and should not be interpreted as representative of the official policies of DARPA or the US Government.

[2] Current address: IBM San Jose Research Lab K55/028, 5600 Cottle Road, San Jose, CA 95193

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-073-7/82/006/0235 $00.75

Once we decide that the system should optimize individual requests, an obvious question is: Why should the optimization be limited to single requests, independent of surrounding requests? Let us give some examples of what we mean by multiple request optimization. When consecutive requests are addressed to a database system

> **Which French ships are in the Mediterranean?**
>
> **Who are the captains of the French ships in the Mediterranean?**

the system will probably independently compute the response to the first query, then compute the response to the second query. It will not recognize that the first response enables fast computation of the second. When a user submits a transaction in which both queries appear, or includes them in a program, the system will probably do no better. Similarly, the requests

> **Which employees in the sales department earn more than $20K, have sold more than 100 personal computers, and have a child who is attending Stanford University?**
>
> **Which employees in the sales department earn more than $20K, have sold more than 100 personal computers, and have a child who is attending New York University?**

might profitably be processed within a transaction by first finding the employees in the personnel department who have sold more than 100 personal computers and earn more than $20K, and then using the temporary to answer both questions.

This is called the *albatross problem* by Daniel Sagalowicz, who regards it as a stigma borne by many database systems. This investigation of the ways in which common expression analysis can improve the optimization of collections of database requests is part of the Knowledge Based Management Systems Project [31] at Stanford and SRI International. Other KBMS work studies optimization using semantic rules [18, 19]; the research described in this paper uses only the structure of the requests and of the database.

A database system may have a repertoire of methods for executing requests, including capabilities for accessing

relations, performing joins, applying restriction predicates, projecting out attributes for results, ordering sets of tuples, calculating arithmetic functions, storing temporary relations (that is, creating new relations whose contents correspond to the results of queries on the original relations), and creating new access paths to relations. We do not consider associated actions such as parsing, authorization checking, concurrency control, and logging.

Optimization of database requests specified in a non-procedural query language involves resolving issues such as in what order relations should be scanned and by what access paths, which join methods should be chosen, how and when restrictions, projections, and sorting should be applied, when new access paths should be created during query processing, and when temporary results should be stored. Although many papers have addressed the problem of query optimization for relational database systems [4, 8, 25, 28, 32, 33, 34, 36], there has been little research on the problem of optimizing collections of requests. As systems with non-procedural query languages, including relational systems, become more prevalent, analysis of this problem will become more important in practice.

Hall [14, 15] uses PRTV (the Peterlee Relational Test Vehicle system) to study techniques for optimizing queries by common subexpression identification, where expressions are regarded as a lattice. His formal techniques are based on the representation of the query as a string of symbols, although associativity is considered. A hill-climbing heuristic is suggested as a means of finding the best combination of common subexpressions. Special attention is paid to recognizing subexpressions *within* a single query. Hall emphasizes the particularly high benefit of recognizing common subexpression when queries involve views. Youssefi and Wong [35, 36] measure the desirability, in INGRES, of building temporaries and access structures during query processing. The PLAIN system [23] enables the user to create tuple identifier lists (called markings), which allows a sufficiently industrious user to optimize explicitly. Kim [17] examines ways of performing scans for multiple purposes simultaneously. Buffer space for temporaries is a critical resource in his analysis. Adiba and Lindsay [1] present a methodology for the creation and use of database snapshots, which can be thought of as stored instances of views. Users explicitly define and refresh snapshots, which have particular value in distributed databases. Grant and Minker [12, 13] use a depth-first algorithm to optimize a collection of conjunctive queries presented in a simplified relational calculus. Disjunction is handled by converting to disjunctive normal form, and treating the disjuncts as separate queries. This approach is applied to deductive database systems. Blaauw, Duyvestijn and Hartmann [6] transform requests into set expressions, and optimize using techniques for circuit minimization in digital switching theory to discover common terms.

We analyze the structure of requests using a formal representation called the ***query graph***, which displays queries in an intuitive manner, and also facilitates comparison and optimization of queries. Based on this representation, we present a methodology for inter-query optimization in the ad hoc query setting:

- Ad hoc requests are presented to the system. These are arbitrary requests, entered in an arbitrary order, by users at their terminals. The system has no predictive capabilities, so ad hoc analysis must determine when stored results are beneficial.

We present the abstract foundations, as well as aspects of a practical implementation designs, for a system that performs common expression analysis in this setting. This system would be able to execute the second French ship query (given at the beginning of this paper) more efficiently, by using the information returned from the first query. (More sophisticated cases will also be examined in this paper.) We have constructed a Pascal program, *COMMON*, which demonstrates the potential practicality of these concepts. This program has been successfully applied to all the examples presented in this paper, as well as more complicated examples.

Elsewhere [10], we study two other settings as well.

- A collection of queries will be repeatedly executed in a transaction. Because the entire collection is available for perusal, and the repeated cost of the transaction dominates compilation cost, sophisticated analysis is possible.

- A program is written in a language that permits embedded database requests, such as PLISQL (PL/I with embedded SQL statements) [3, 26]. This is optimized, using a combination of code optimization techniques and a variation of the approach employed in the other two settings.

The effects of insertions, updates, and deletions are also discussed there [10]. In this paper, only queries are considered. In section 2 we define query graphs, and give examples of query graphs. We also explain informally, using these graphs, how a system might use the answer for one query to compute the answer for another. In section 3 we formally define when it means for a query to be a subexpression of another, and give an algorithm for testing for this. Section 4 is the conclusion.

2. Query graphs

In this section, we introduce a notation for queries called the ***query graph***. Queries are represented as undirected graphs. The nodes correspond to occurrences of relations in the query; the edges correspond to joins between the relations in the nodes that they connect. There have been many similar approaches to structuring queries, including QBE [37], tableaux [5], qual graphs [7], and the work in many of the optimization papers cited in section 1, especially the papers by Wong and Youssefi [32, 33, 36].

Welty and Stemple [29] suggest that explicit indications of join operations assist the user in writing and understanding queries. In that paper, the authors' human factors studies provide evidence that **structured introduction and presentation** of queries is useful. But we feel that they do not go far enough; most predicates can be associated with a relation in the query, or identified as join predicates. This approach is basic to our common expression analysis methodology, but it also structures queries for human use.

We shall assume that the following relations exist in the database, with attributes including those listed.

Relation	*Attributes*
People	name, employer, age, experience, salary, commission, education
Companies	corpname, location, earnings, president, business
Schools	schoolname, level

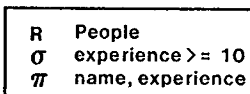
2.1. Some query graph examples

These examples represent six queries in SQL [3] and as query graphs. (Our implementation actually uses SODA queries [16, 20], but SODA is not intended as a user interface.) We discuss the examples, which we also follow through the paper, using standard graph and database terminology in this section, and define terms rigorously in the next section. A query corresponds to an expression in a query language, and a snapshot is the interpretation of a query corresponding to a database instance. A temporary is a relation stored in the database whose tuples are those in a snapshot. In what follows, we assume that there is an arbitrary fixed database instance, and that Snapshot 1 and Temporary 1 correspond to Query 1, etc.

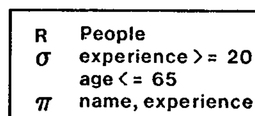
Query 1

```
SELECT name, experience
FROM   People
WHERE  experience >= 10
```

Query 1 is a single node, on the relation People. All people with at least 10 years of experience are selected, and the name and experience attributes for each are projected.



QUERY 1



QUERY 2

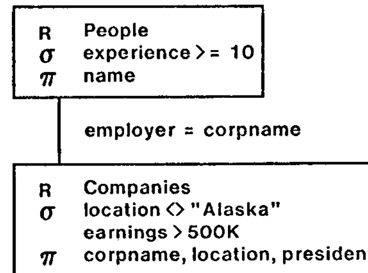
Query 2

```
SELECT name, experience
FROM   People
WHERE  experience >= 20
  AND  age <= 65
```

Query 2 is also on the People relation. Snapshot 2 should contain only people who have at least 20 years of experience, and who also are 65 or younger. It is a subset of Snapshot 1, since anyone who satisfies the Query 2 predicates must satisfy the weaker conditions for Query 1. (Equivalently, we can observe that the selection predicate for Query 2 *implies* the selection predicate for Query 1.) Fortunately, the projected attributes are identical, and since experience is projected, we can easily test to see whether a person in Snapshot 1 has 20 years experience. Age, however, is not available in Snapshot 1. If we had known about both queries in advance, we could have projected age, as well as name and experience, when we answered Query 1. (Age would not be displayed as part of the answer.) Let us assume that name uniquely identifies tuples of the People relation. If there is an index, or some other fast access path to the tuples of People, based on experience or name, it is possible, and may be worthwhile, to obtain Snapshot 2 by performing a join between Temporary 1 and the People relation, and restricting further. (It is likely that an index exists on unique identifiers.)

Query 3

```
SELECT p.name, c.corpname,
           c.location, c.president
FROM   People p, Companies c
WHERE  p.experience >= 10
  AND  p.employer = c.corpname
  AND  c.location <> 'Alaska'
  AND  c.earnings > 500K³
```



QUERY 3

Query 3 involves two nodes, on two different relations, People and Companies. Not only must internal selection predicates hold, but the Cartesian product of People and Companies is restricted by the join predicate requirement that the person's employer be the company. If Snapshot 1 is available, we can join it with the indicated subset of Companies. But since the employer field of People was not projected in Snapshot 1, this would again involve having to extract values from the People relation itself (so Snapshot 1 may not be beneficial).

Query 4

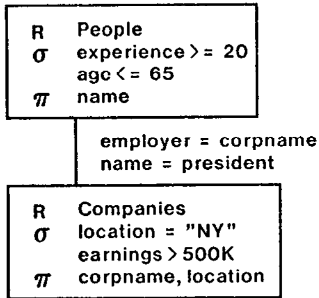
```
SELECT p.name, c.corpname, c.location
FROM   People p, Companies c
WHERE  p.experience >= 20
  AND  p.age <= 65
  AND  p.employer = c.corpname
  AND  p.name = c.president
  AND  c.location = 'NY'
  AND  c.earnings > 500K
```

Query 4 has two nodes on the same relations as Query 3. Moreover, each node of Query 4 can be obtained by restricting the corresponding Query 3 node further. Also, the join predicate for Query 4 is stronger than that of Query 3. Hence, we can restrict Snapshot 3 to obtain Snapshot 4. (Age and employer were not projected into Snapshot 3, so joining People and Temporary 3 is necessary, if the temporary is to be used. However, we need not perform a join with Company, since the

---

[3] We regard earnings as given in thousands; the use of 500K to represent 500 thousand is not part of SQL, but "earnings > 500" looks meager. Also, the table variables *p* and *c* may be omitted, but we include them to emphasize the associations of attributes with relations.

attributes corpname, location, and president are projected into Snapshot 3, and the earnings predicate is already known.)



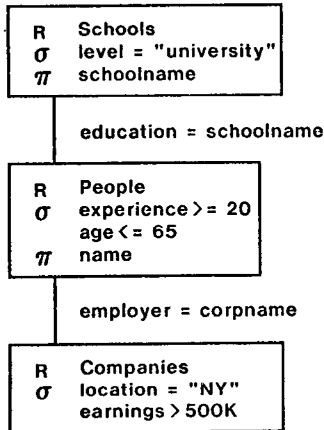
QUERY 4

Query 5

```
SELECT p.name, s.schoolname
FROM   People p, Companies c, Schools s
WHERE  p.experience >= 20
  AND  p.age <= 65
  AND  p.employer = c.corpname
  AND  c.location = 'NY'
  AND  c.earnings > 500K
  AND  p.education = s.schoolname
  AND  s.level = 'university'
```

Query 5 has three nodes on different relations. Either Snapshot 1 or Snapshot 2 could be used to help obtain Snapshot 5, since the People node in Query 5 further restricts that of Query 1 and Query 2. Snapshot 2 is probably preferable, since Query 2 has the same selection predicates on People that Query 5 has. The education and employer attributes are not available in either Temporary, so People tuples must still be accessed.

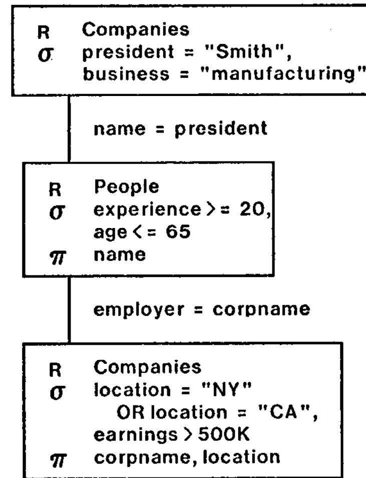
Snapshot 3 has two of the three nodes, and each can be restricted further to obtain the conditions in Query 5. Moreover, the join predicate between those nodes in Query 5 is a further restriction of the join predicate of Query 3 (trivially, since they are the same). Hence, we can answer Query 5 by joining Temporary 3 with Schools (and with People, since the education attribute was not projected). Snapshot 4 cannot be used for Query 5, because its join predicate is not implied by the join predicate (between People and Companies) in Query 5.



QUERY 5

Query 6

```
SELECT p.name, c.corpname, c.location
FROM   People p, Companies c, Companies c1
WHERE  p.experience >= 20
  AND  p.age <= 65
  AND  p.employer = c.corpname
  AND  ( c.location = 'NY'
         OR c.location = 'CA' )
  AND  c.earnings > 500K
  AND  p.name = c1.president
  AND  c1.president = 'Smith'
  AND  c1.business = 'manufacturing'
```



QUERY 6

Query 6 has three nodes, where one relation has two occurrences. Nodes must be distinguished from relations, since multiple occurrences are possible. Note that there are several possible ways in which we can try to match Query 3 and Query 4 with Query 6. But node *c1* (which has the test on company business) is not a further restriction of any node that has appeared in an earlier query, so that no match which involves *c1* can succeed. Snapshot 3 is potentially beneficial, however, if the *c* node of Query 6 is matched with the Companies node of Query 3.

Our program, *COMMON*, detects all of the uses of snapshots that are described above. Although our approach enables detection of even less obvious cases, we believe that the most likely and fruitful case is illustrated by the French ships queries that appear in the introduction, where further information was requested after the ships were identified. This is also illustrated by the further restriction of Temporary 1 to obtain the answer to Query 2.

2.2. Query graph definition

We assume that a base set of *relation schemas* has been specified, as in the relational model; this is called the *database schema*. *Attributes* (sometimes called fields or columns) are defined for each relation. An attribute consists of a *name* and a *domain*. Attribute names must be distinct within a relation. If R is a relation schema, and A is the name of an attribute of R, we write *R.A* to refer to that attribute, and write A when the relation R is clear from context. A *relation instance* is a set of *tuples*, where each tuple associates each attribute with a value in its domain. A *database instance* associates each relation schema in D with a relation instance. As is customary, we often use the words *relation* and *database* for both schema and instance, trusting context to disambiguate.

For simplicity, assume that the domain for all attributes is the integers. The integers have special properties, such as order, arithmetic operation and discreteness, but the modifications to support other domains are usually straightforward (e.g., limiting terms to single attributes or constants).

A *term* is a linear sum $c_0 + c_1A_1 + c_2A_2 + \ldots + c_nA_n$, where the A's are attributes, and the c's are integer constants. An *atomic selection predicate* has the form τ_1 *op* τ_2, where τ_1 and τ_2 are terms, and *op* is one of the comparison operators $=, \neq, <, \leq, >, \geq$. The *selection predicates* are the smallest class of predicates containing the atomic selection predicates and closed under the Boolean operations: *NOT*, *AND*, and *OR*. For example, *experience* $\geq$ *10* is an atomic selection predicate, and *experience* $\geq$ *20 AND age* $\leq$ *65* is a selection predicate.

A *projection set on relation R* is a subset of the attributes of R. The projection set corresponds to the attributes from the relation required in the answer; these are sometimes referred to as *projected*, *selected*, or *targeted* attributes. In practice, the order in which the attributes appear is obviously of importance in report generation and other applications. We shall ignore this property, since it does not affect our results here.

Let G be an undirected graph (without self-loops) with nodes N and edges E. In a query graph, nodes correspond to occurrences of relations in the query (as with appearances in the *FROM* clause in SQL [3], or the *RANGE* clause in QUEL [27]). Node labels give the relation, the internal selection criteria, and the attributes requested in the response to the query. We identify σ with the conjunction of its elements. Each of the individual conjoined predicates is called a *clause*.

Definition 1: A *node label* is a triple $\langle R,\sigma,\pi \rangle$, where:

1. R is a relation, referred to as the *underlying relation* of the label, or of the node with which the label is associated.
2. σ is a set of selection predicates in which all attributes are in R. (They are referred to as the *internal or local predicates* for the node.)
3. π is a projection set on R.

A function NL which associates a node label with each node of G is called a *node labelling* for G.

Given G and a node labelling for G, if n_1 and n_2 are (distinct) nodes of G which have underlying relations R_1 and R_2 respectively, a *join predicate* between n_1 and n_2 has the form $n_1.A_1 = n_2.A_2$, where A_1 and A_2 are attributes (of R_1 and R_2, respectively). When the underlying relation identifies the node, we write the join predicate as $R_1.A_1 = R_2.A_2$.

Definition 2: Given a node labelling for G, if e is an edge of G then an *edge label* for e has the form $\langle n_1,n_2,X \rangle$, where:

1. n_1 and n_2 are the nodes connected by the edge.
2. X is a non-empty set of join predicates between n_1 and n_2.

A function EL which associates edge labels with each edge in E is called a *edge labelling*.

All the predicates in labels for nodes and edges are conjoined to form the restriction criterion for the query. By associating clauses with nodes and edges, rather than regarding the predicates as restricting the Cartesian product of the nodes' relations, we separate the restriction criteria in a way that is easy to grasp, and also assists analysis of query structure in an optimizer or common expression analyzer. Unfortunately, not all clauses can be associated with a node, as a selection predicate, or with an edge, as a join predicate. For example, *p.salary* $\geq$ *30K OR c.earnings* $\geq$ *200K* does not fit into the framework described so far, nor does *p.age* = *p.experience + 20*,[4] nor does *p.salary + d.budget < c.earnings.* Any selection clause (over the attributes in the underlying relations of the nodes of G) which does not qualify as a selection predicate (because attributes come from two or more nodes), nor as a join predicate (because it does not have the right form) is called a *global predicate*. Since global predicates cannot be associated with a node or an edge, they require special handling. Requests frequently have empty global predicates.

Definition 3: A *query graph* is a 4-tuple $\langle G,NL,EL,\gamma \rangle$, where:

1. G is an undirected graph (without self-loops).
2. NL is a node labelling for G.
3. EL is an edge labelling for G and NL.
4. γ is a set of global predicates for G.

We shall not always distinguish between queries and their query graphs.

The definitions given above exclude the operations union and difference, aggregate functions such as *MIN* and *MAX*, and the quantifiers *SOME* (*EXISTS*) and *NONE* (which, curiously, is more convenient for us than is *ALL*). Elsewhere [10], we extend the query graph definition to permit these operations, and consider a more general class of join predicates.

As we have already mentioned, a snapshot is the interpretation of a query corresponding to a database instance, and a temporary is a snapshot stored as a relation in the database.

---

[4] We consider the possibility of regarding this as a join predicate in [10].

3. Ad hoc query setting

When a transaction is known in advance and pre-compiled, access path selection can be performed once, before processing. In certain applications, requests are not known in advance. These ad hoc requests are unpredictable; it is not, however, unreasonable to assume that such requests will sometimes resemble each other, with questions becoming more detailed based on information returned in previous requests.

For example, after getting a response, a user might want to see more attributes from the relations involved in the question. A new request might demand a join of the previous answer to another relation. A question might be repeated. An answer might have included too much information, and additional restrictions might be added. (This may happen when a count of items, such as bibliographical references, is returned, and the user decides whether or not to see them based on that count.) When there is more than one user on the system who is interested in a current topic, those users may be asking questions that are closely related. In a military application, several users might simultaneously be interested in foreign ships in a crisis area. In a business application, several users might want to know about today's shipments. In a ticket sales application, several users might be interested in the same upcoming events.

We want to find situations in which there are large savings, without significantly adding to the cost of normal processing. Moreover, we are not assuming any semantic knowledge of the contents or sequencing of these ad hoc requests. There may exist particular ways of processing requests that produce temporaries that will be very useful in the future. But because we do not know the future, we simply decide how to process each request independently, based on the request and existing temporaries.

3.1. Required capabilities in ad hoc query setting

The following capabilities are required in a system which uses common expression analysis to optimize ad hoc queries.

1. *Creation of Temporaries*
   The system must be able to create and store temporary relations, which may correspond to answers to previous requests, or to intermediate results materialized (that is, physically created) as part of the query processing for previous requests. A query graph description of the snapshot must also be saved, and descriptive information about the snapshot should be entered in the system catalog (or schema), as a (temporary) relation, so that requests can refer to it as they can to the base relations.

   Access paths to the temporary (such as indexes) might also be created at this time. (Youssefi [35] studies creation of auxiliary access paths as a strategy for query processing.)

2. *Use of Temporaries*
   The system should determine which temporaries are potentially beneficial as subexpressions for the current request. The query has to be reformulated using the snapshot (or perhaps using several snapshots), and the system must compare the expected costs of the original and the revised requests. If the revised form turns out to be less expensive to use, the system should execute that instead.

3. *Background Support*
   The system must maintain a storage discipline for temporaries. Since space for temporaries is limited, storing a temporary may entail the destruction of some other temporary. By attaching a utility value to each temporary, based on factors such as the temporary's size, usage frequency, user priority, and the complexity of the description of the snapshot, the system decides when temporaries must be replaced.[5] Also, when the base relations on which the snapshot is defined are updated, the temporary relation must be either updated or destroyed, unless users are willing to see answers based on obsolete data.

In the rest of this section, we describe a method, based on the query graph formalism, for determining the snapshots whose temporaries might be beneficial for processing the current query. These auspicious snapshots are sub-expressions, in a sense which we formally define below, of the current query.

3.2. Upper bounds

The common subexpression detection problem in compiler optimization of straight-line code for arithmetic expressions is well known. Since the associative and distributive laws are false on finite precision machines,[6] operations are analyzed in a canonical form that reflects the commutativity of + and *. The problems of finding minimal programs for collections of arithmetic expressions, and optimizing program code with assignments and branches, have a long history of analysis [2, 9].

In order to explain what it means for one query to be a subexpression[7] of another, or for an expression to be a subexpression of two queries, we begin by comparing nodes. If T and Q are single node queries (with no global predicates), what relationship must T and Q have so that we can obtain the answer to query Q merely by restricting the answer to query T further?

Definition 4: If $n_T = \langle R_T, \sigma_T, \pi_T \rangle$ and $n_Q = \langle R_Q, \sigma_Q, \pi_Q \rangle$ are nodes, then n_T is an *upper bound* for n_Q if

1. $R_T = R_Q$, and
2. $\sigma_Q \rightarrow \sigma_T$.

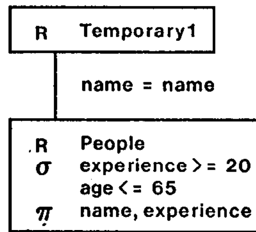
[5] More advanced strategies, based on prediction of user's area of interest and subsequent requests, are outside the scope of this paper.

[6] Some language standards permit the assumption of associativity.

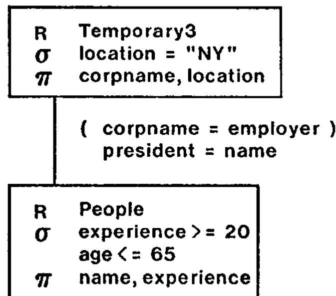
[7] The term *subquery* has a standard meaning in [3]. We shall identify queries with their expression as query graphs, and speak of subexpressions.

For example, Query 1 and Query 2 are both single node queries, and Query 1 is an upper bound on Query 2, but not conversely. If n_T is an upper bound for n_Q, and T is a stored temporary for n_T, we may be able to use T to obtain the answer to the query represented by n_Q, by *further restricting* T. Every tuple in the original relation that is needed for n_Q participates in n_T. But this is not enough, since some attributes which do not appear in π_T may be needed to perform the further restriction to σ_Q. For example, *age* is not in the projection set for Query 1. Moreover, π_Q may contain attributes not in π_T.

The *name* attribute in the People relation *uniquely identifies* People tuples, and may be indexed, or accessible using hashing. (This is likely, since uniqueness must be guaranteed.) Query 2A is the revised query for Query 2, using Temporary 1. We must join the temporary to one of its base relations to obtain the value of age. This join back to one of the defining relations of the temporary is called a *back-join*. Although Query 2A requires a back-join to People, it might be processed more quickly than Query 2, especially if Temporary 1 had few tuples. Query 4A is the revised query for Query 4, using Temporary 3. It requires a back-join to People, but does not require a back-join to Companies. The appearance of the join predicate *corpname = employer* is unnecessary, and will be discussed in section 3.3.



QUERY 2A



QUERY 4A

In general, the expected cost of accessing the required tuples, as measured by evaluation formulas [25, 30, 34] may decrease when the system follows a *fast access path* to the tuples of the base relation using attributes in the projection set of the temporary.

If $\sigma_Q \rightarrow \sigma_T$, where σ_Q and σ_T are sets of clauses, then σ_Q/σ_T denotes the clauses in σ_Q that are not implied by σ_T. Hence, $\sigma_Q \equiv (\sigma_T \textit{ AND } \sigma_Q/\sigma_T)$, and σ_Q/σ_T is the minimal subset of σ_Q which has the property that *AND*'ing it with σ_T yields (the equivalent of) σ_Q. *Further restriction of T by any subset of σ_Q having this property would allow us to get the answer to Q;* we shall take advantage of that in section 3.4, since we do not perform complete theorem proving.

Let $ATTRIBS(\alpha)$ represent the attributes which appear in predicates α.

Definition 5: If $n_T = \langle R_T, \sigma_T, \pi_T \rangle$ and $n_Q = \langle R_Q, \sigma_Q, \pi_Q \rangle$ are nodes, then n_T is an *effective upper bound* for n_Q if n_T is an upper bound for n_Q, and either

1. $\pi_T \supseteq ATTRIBS(\sigma_Q/\sigma_T)$, and $\pi_T \supseteq \pi_Q$, or
2. π_T includes both a unique identifier for R_T and a fast access path to R_T.

Definition 6: If n_T is an effective upper bound for n_Q, then the *revised query* for n_Q using a temporary T corresponding to n_T is defined based on whether the first or the second case of Definition 5 applies.

If the first applies, the revised query has one node, $\langle T, \sigma_Q/\sigma_T, \pi_Q \rangle$.

If the second applies, the revised query has two nodes, $\langle T, TRUE, \emptyset \rangle$ and $\langle R_Q, \sigma_Q/\sigma_T, \pi_Q \rangle$, with a join predicate (which is the back-join referred to above) equating the unique identifier (which must appear in both nodes) to itself.

Note that the definition above does not mention the fast access path; that qualification is required as a (very good) heuristic, to eliminate inefficient revised queries from consideration. *We rely on the optimizer to choose the fast access path, when the revised query is submitted, as well as to furnish the costs of the original and revised query.* This approach is also used in the physical database design tool DBDSGN [11]. Definition 5 is imprecise (we prefer to say system-dependent), since the meaning of *fast* depends heavily on the capabilities of the system, and on the evaluation formulas applied by the optimizer procedure. We shall not be more precise, preferring to regard the optimizer as a black box which delivers predictions. Its judgments decide how queries are actually processed. Both the original and the revised query must be submitted for evaluation, and the version with the smallest expected cost should be executed.

3.3. Subexpressions

When one node is an effective upper bound on another, we can define a revised query, which may have a lower processing cost. We now generalize this to testing whether a temporary T (which has t nodes) may be useful for processing a new query Q (which has q nodes). Assume that there is a particular mapping from the nodes of T to the nodes of Q. (Many mappings may be possible, but underlying relations of mapped nodes must be the same, as in the previous section.) The node mapping must be one-to-one, but it need not be onto, since there may be additional nodes in Q that are not in T. Hence $t \leq q$. We assume that the nodes of Q and T are numbered, where Q_i and T_i correspond for $i = 1, 2, ..., t$. X_{ij}^T is the edge label (set of join predicates) for the edge between T_i and T_j, and similarly for X_{ij}^Q. (When there is no edge, we have the empty set of join.

predicates, which is trivially *TRUE*.) We identify mapped nodes from T with the corresponding nodes in Q. If α_Q and α_T are sets of predicates for Q and T, which may involve multiple nodes, and $\alpha_Q \rightarrow \alpha_T$, then α_Q/α_T may be defined as in the previous section, since a mapping between nodes which preserves underlying relations has been specified.

In a query, there are join predicates and global predicates, as well as local predicates. We need to further restrict the predicates of T to obtain Q. Hence we need a variation of another definition of the previous section.

Definition 7: If $\alpha_Q \rightarrow \alpha_T$, then *T enables further restricting of α_T to α_Q* if for any node Q_k mentioned in α_Q, either

1. $\pi_k^T \supseteq$ the attributes in $ATTRIBS(\alpha_Q/\alpha_T)$ that come from node Q_k's relation appearance, or

2. π_k^T includes both a unique identifier for R_k and a fast access path to R_k.

When the first alternative does not hold and the second one does, the revised query for Q using T will require a back-join to Q_k, since attributes of Q_k which are not in T_k are required. Note that the definition of *effective upper bound*, in the previous section, ensures that T enables further restricting of σ_T to σ_Q, and may also cause back-joins in the revised query.

Definition 8: T is a *subexpression* of query Q if there are node orderings:

$T_1, T_2, \ldots, T_t$ for T,

$Q_1, Q_2, \ldots, Q_q$ for Q,

where $t \leq q$, and:

1. Nodes: T_k is an effective upper bound for Q_k, for $k=1, 2, \ldots, t$.

2. Edges: $X_{ij}^Q \rightarrow X_{ij}^T$, and T enables further restricting of X_{ij}^T to X_{ij}^Q, for $i=1, 2, \ldots, t$, and $j=1, 2, \ldots, q$.

3. Global: $\gamma_Q \rightarrow \gamma_T$, and T enables further restricting of γ_T to γ_Q.

Definition 9: If T is a subexpression for Q (using the node mapping given in the previous definition), the *revised query* Q' for Q using T is defined as follows.

All nodes Q_k of Q that require back-joins (on the basis of local, join or global predicates) are in Q'. The relation and projection set are as they are in Q. The local predicates for Q' are those in σ_k^Q/σ_k^T.

All nodes Q_k of Q that do not require back-joins correspond to the same special new node in Q'. The relation for that node is T. The local predicates for that node are the union of σ_k^Q/σ_k^T for all such k. The projection set for that node is the union of π_k^Q for all such k.

All nodes Q_k, where $k > t$, appear in Q' just as they are in T.

Whenever X_{ij}^Q/X_{ij}^T is non-empty, it appears as (part of) an edge label in Q', between the nodes of Q' corresponding to i and j.[8] All edges involving a node Q_k, where $k > t$, must appear in the revised query.

The global predicate for the revised query is γ_Q/γ_T.

There are many ways to implement the concepts embodied in the above definition. Which way is best depends on how much of the system's resources can be devoted to common expression analysis. We shall outline one concrete realization of these concepts, most of which is embodied in a program, *COMMON*, which detects common expressions and produces revised queries corresponding to them.

3.4. Algorithm for ad hoc query setting

Practical considerations (we wish to avoid the execution time burden of a general purpose theorem prover) prevent us from solving the decision problem for:

$$\sigma_Q \rightarrow \sigma_T$$

completely, in the sense that we answer affirmatively only when the implication is valid, but we respond negatively for some valid statements. This is acceptable. Although some optimizations escape us, all the subexpressions which we do detect correctly support transformation of queries.

Thus we do not test whether the conjunction of all the predicates in the query necessarily implies the conjunction of all the predicates in the temporary. Instead, we perform a weaker test based on the query graph structure, and might speak of *formal implication*, rather than implication. The details of this test appear (implicitly) in the algorithm later in this section. However, with the understanding that we do perform this weaker test, we shall continue to use terms from the previous sections, such as upper bound and subexpression, without prefacing them with the adjective "formal."

The steps in our algorithm, each of which is described in detail below, are *Normalization* (organize query and create query graph), *Signature* (efficiently reject most temporaries that cannot be used), *Propagation* (movement of predicates to detect additional subexpression possibilities, and join predicate checking), *Matching* (map temporary nodes to query nodes), *Direct Elimination* (detecting clauses that are trivially implied), *Connected Components* (partitioning implication problem into independent subproblems), *Resolution* (more general implication proving), *Global predicates testing*, *Revised query generation*, and *Cost comparison* (between alternatives).

INPUT: Query graphs for a query and a set of temporaries.

PROBLEM: Decide whether or not some temporary is a subexpression of the query. For each such temporary, find the cost of the revised form of the query using the temporary, so that the version of the query with the least expected cost is chosen for execution.

---

[8] The predicate *corpname = employer* of Query 4A is not required, since it is a join predicate for Query 3. It may, however, provide a useful access path in processing Query 4A. If it does not, it should be eliminated.

Step 1: Normalization. *NOT* is eliminated using De Morgan's laws. The clauses of the query are identified as local, join or global predicates, and they are expressed in the normal form for selection predicates, with terms as linear sums. Other simplifications to reduce work in later steps include elimination of > and ≥ (since < and ≤ always suffice), and elimination of < (replacing a < b with a + 1 ≤ b, using the discreteness of the integers).

Step 2: Signature. Since we want to reject most temporaries quickly, we associate a signature with each temporary. The signature indicates which relations appear in the temporary, and which attributes must appear in any query that can use the temporary. For example, any attribute which appears in a clause of the temporary's definition that does not contain *OR* must appear in any query that can use the temporary. The temporary will be considered further only if all the relations and attributes mentioned in its signature also appear in the query. A hashed representation of the temporary's signature, stored with the temporary, allows us to quickly compare it with a similarly hashed representation of all the relations and attributes that appear in the query.

Most of the time, irrelevant temporaries will be quickly discarded in this step. Subsequent steps will catch other temporaries that cannot be used.

For example, Temporary 5 cannot be used for Query 6 because the Schools relation does not appear in Query 6, and Temporary 2 cannot be used for Query 3, because the age attribute does not appear in Query 3.

Step 3: Propagate query's predicates. This step has to be performed only once for each query, no matter how many temporaries are compared with it. Since we want to show that the query's predicates imply the temporary's predicates, we try to build up all the information that we can about the query's restriction. If a clause appears as a local predicate for a node of the query, and some edge from that node has all the attributes that appear in the clause in its label, propagate that clause to the other side of the edge (replacing attributes according to the equijoin specified in the edge label). For example, in Query 6, the clause c1.president = 'Smith' propagates to node p as p.name = 'Smith'.

Similarly edges may "propagate across edges." For example, if $a = b$ and $b = c$ label edges, we would generate $a = c$. No other interaction takes place across edges. As a byproduct of this process, the join predicates of the query are enhanced with other information describing equality of attributes (and constants) - an equivalence class. We check that the enhanced join predicates labelling edges of the query imply the labels on corresponding edges (that is, the edges between corresponding nodes) of the temporary. If not, the temporary is deemed unusable.

Step 4: Match query to temporary. For the temporary to be a subexpression of the query, there must be node orderings for the temporary and the query that demonstrate this. Perform Steps 5 to 10 for each different ordering that matches underlying relations. There may be exponentially many matches possible (if many nodes are on the same relation), but in practice the number of matches to consider is small.

Step 5: Direct elimination. Steps 5, 6, and 7 are performed to compare the σ's in corresponding nodes of the temporary and the query. In the direct elimination step, we find and eliminate clauses of the temporary that are implied by a clause in the corresponding node of the query. The direct elimination table in the Appendix gives the conditions for this elimination among atomic predicates; this can also be generalized to selection predicates involving *AND* and *OR*.

For example, if p.salary < 10K appears in the query, we can eliminate p.salary < 20K from the temporary. Other examples given in the Appendix show that clauses involving multiple attributes, and even non-atomic clauses, can be eliminated from the temporary by direct elimination. This simple table-driven test often reduces the number of predicates in the temporary significantly. Note that the elimination is just for the purposes of testing implication; we do not alter the expression for the temporary itself.

Direct elimination tests for implication using the structure of the selection predicates. To test if Temporary 3 can be used for Query 4, using the obvious matching between nodes, we find that each of the predicates in Temporary 3 is directly eliminated by a predicate in Query 4. Hence steps 6 and 7 are unnecessary, since direct elimination demonstrates the implication. A rudimentary implementation of our approach could omit steps 3, 6, and 7. (The implementation of the program *COMMON* includes step 3, but does not include steps 6 and 7.)

Step 6: Connected components. A predicate that does not involve that attribute age cannot help in implying a predicate that does mention age, unless there is some way of "connecting" its predicates with age. For example *experience ≥ 10* does imply *age ≤ 65 OR experience ≥ 10*. This suggests a way of breaking the implication problem into smaller subproblems, or perhaps deciding immediately that implication is impossible.

For each corresponding pair of nodes in the query and the temporary, define an *attribute connection* graph. The vertices of this graph correspond to the attributes that appear in the query or the temporary, annotated with the number of the nodes (in the node ordering) in which they appear. We do not distinguish whether they come from the temporary or the query. Connect these vertices if they appear together in some clause of either the temporary or the query. Attributes which appear in different connected components of the attribute connection graph cannot influence each other. Step 7 can be performed independently for each connected component. If some component consists entirely of attributes that do not appear in the query, then we can immediately reject use of the temporary.

Step 7: Resolution. We have reduced testing requirements considerably in steps 5 and 6, but will still refer to the remaining clauses (in one component, for one node) as σ_Q and σ_T. Testing whether $\sigma_Q \rightarrow \sigma_T$ is equivalent to determining whether (σ_Q *AND NOT* σ_T) is unsatisfiable. We may apply a variation of the Fourier-Motzkin elimination method for checking infeasibility of conjunctions of linear inequalities (over the integers). This algorithm is described and analyzed by C. G. Nelson [21] under the assumption of two attributes per clause. Disjunction is handled by showing infeasibility no

matter which disjunct is chosen. The algorithm resembles resolution, in that each pair of linear forms which provide an upper bound and a lower bound for the value of some attribute are combined, eliminating that attribute and relating the lower and upper bounds. The resulting clauses are satisfiable if and only if the original clauses were satisfiable.

Rosenkrantz and Hunt [24] show that the satisfiability problem is NP-hard when ≠ (which produces a disjunction of linear inequalities) is permitted, even if comparisons involve no more than 2 attributes. In practice, however, step 7 will usually be fast, because the number of clauses to examine has been reduced in earlier steps.

For example,

$$(a \geq b \ \textit{AND}\ b \geq c) \rightarrow a \geq c$$

will be verified very quickly. Heuristic cut-off rules such as:

- Never produce a resolvent with more attributes than the maximum number in the clauses resolved.

- Limit the total number of resolvents produced, giving up on the temporary if unsatisfiability hasn't been shown in a limited time.

are desirable, since we seek fast detection of obvious consequences, rather than completeness.

A variation of the simplex algorithm, as described by Nelson [22], is also suitable for this problem.

Step 8: Global predicates. We expect that global predicates occur infrequently. To test whether $\gamma_Q \rightarrow \gamma_T$, we see whether each clause of γ_T is implied by some clause of γ_Q. Although more complicated strategies may be attempted, we prefer to optimize for the simpler, more frequent cases, because global predicates are hard to deal with. Temporaries with complicated global predicates should have low probabilities of being stored. It is reasonable to perform this step early in the processing, immediately after step 2 (node matching) to allow fast rejection when the global predicate of the temporary is not implied by that of the query.

Step 9: Revise query. We described query revision in the section 3.3, and showed two examples of revised queries. Node matches that require back-joins can be rejected as soon as the need for the back-join is detected, if the unique identifier and fast access path required for the back join are not in the projection set of the temporary.

Step 10: Compare costs. If the cost of execution has been estimated for the original query, and all the revised queries, as well as for revisions using more than one temporary (where the temporaries cover disjoint parts of the query), then we can choose the version which has the smallest expected cost, and execute it.

An optimization system configured for common expression analysis might recognize that many of the values calculated during optimization of the original query will also be valuable during optimization of revisions of the query. The query graph lends itself to annotation with values such as filter factors for selection predicates and join predicates, best access paths to nodes, and descriptions of plans for query processing.

4. Conclusion

In this paper, we have presented the query graph representation, and described how it can be used for optimization of a sequence of ad hoc queries using common subexpression analysis.

The query graph representation organizes a query description so that many aspects of the query's structure are apparent. Unlike the representations of most query languages, which display all predicates together, the query graph associates clauses with nodes (relation occurrences) and edges (joins), whenever possible. This has advantages for entry and display of queries, as well as for internal representation.

For processing a stream of ad hoc queries, the query graph enables us to recognize providentially available temporaries which were stored during processing of previous requests. We provide an abstract framework and a concrete algorithm for exploiting these temporaries.

We also examine [10] more general queries, and the effects of updates. For batches of requests in a transaction, and requests embedded in a programming language, we show how the same concepts can be applied when we have complete information about the operations to be performed.

We have implemented a Pascal program, *COMMON*, which can detect temporaries that may be useful for processing a new query in the ad hoc context, and which can formulate the revised query for such temporaries. The time for performing these functions for the six example queries presented in this paper range from 7 milliseconds (to decide that a query could not be used, on the basis of signature) up to 45 milliseconds (to decide that Query 6 could use Temporary 1). This includes neither optimization nor any form of cost estimation. Considerably more time is required to read (and normalize) queries. It is also clear that common expression analysis can yield large savings. We believe that as the use of relational databases systems becomes common, tools for processing applications will become more accepted and important.

Acknowledgement

The author would like to thank Gio Wiederhold, Daniel Sagalowicz, Jim Davidson, Arthur Keller, and the other members of the KBMS project for their generous support, encouragement and assistance in preparing this paper.

Appendix: Direct elimination

	=	<	>
=	=	<	>
≠	≠	N	N
<	<	<	N
≤	≤	≤	N
>	>	N	>
≥	≥	N	>

Table 1: Direct elimination table

Let $\tau\ op_Q\ v_Q$ be a clause in the query, and let $\tau\ op_T\ v_T$ be a clause in the temporary, where v_Q and v_T are constants. We want to decide if the former directly eliminates the latter. Look at the row of Table 1 for op_Q.[9] Look at the column of Table 1 corresponding to the result of comparing v_Q with v_T. The entry in the Table gives the strongest condition that we can have for op_T which permits direct elimination. [If the entry is < or =, then op_T could be the weaker ≤; the dual statement for ≥ is also true.] N means that elimination cannot be performed in this case.

For example, if *p.salary < 10K* appears in the query, we can eliminate *p.salary < 20K* from the temporary, since the entry for row < and column < is <. The clause *p.salary ≤ 20K* could also be eliminated. If *p.salary = p.commission + 500* appears in the query, we can eliminate *p.salary ≥ p.commission + 300* from the temporary. A clause in the temporary which is a disjunction may be eliminated if there is a clause in the query such that every disjunct in the temporary clause is implied by a disjunct in the query clause. For example, if *p.salary ≤ 20K OR p.salary ≥ 60K* is in the query, it can eliminate *p.salary < 10K OR p.salary > 80K*. A recursive definition of direct implication allowing arbitrary nestings of conjunction and disjunction is left to the reader.

References

1. Adiba, M.E. and Lindsay, B.G. Database snapshots. Proc. Sixth International Conference on Very Large Data Bases, Montreal, Oct., 1980, pp. 86-91.

2. Aho, A.V., Johnson, S.C., and Ullman, J.D. "Code generation for expressions with common subexpressions." *J. ACM 24*, 1 (Jan. 1977), 146-160.

3. Astrahan, M.M. and Chamberlin, D.D. "Implementation of a structured English query language." *Comm. ACM 18*, 10 (Oct. 1975), 580-588.

4. Astrahan, M.M., Kim, W., and Schkolnick, M. Performance of the System R access path selection mechanism. In *Information Processing 80*, Lavington, S.H., Ed.,North Holland, 1980, pp. 487-491.

5. Aho, A.V., Sagiv, Y., and Ullman, J.D. "Efficient optimization of a class of relational expressions." *ACM Transactions on Database Systems 4*, 4 (Dec. 1979), 435-454.

6. Blaauw, G., Duyvestijn, A., and Hartmann, R. Optimisation of relational expressions using a logical analogon. Submitted to *IBM Systems Journal*

7. Bernstein, P.A. and Goodman, N. "Power of natural semijoins." *SIAM J. of Computing 10*, 4 (1981), 751-771.

8. Blasgen, M.W. and Eswaran, K.P. On the evaluation of queries in a relational data base system. Research Report RJ1745, IBM, April, 1976.

9. Cocke, J. and Schwartz, J. Programming Languages and Their Compilers. Tech. Rept. Second Revised Version, Courant Institute of Mathematical Sciences, New York University, April, 1970.

10. Finkelstein, S. J. Common expression analysis for optimization of database applications. Computer Science Department, Stanford U., 1982. To appear.

11. Finkelstein, S. J., Schkolnick, M., and Tiberio, P. A physical database design tool for relational databases. IBM, 1982. To appear.

12. Grant, J. and Minker, J. On optimizing the evaluation of a set of expressions. Tech. Rept. TR-916, Computer Science Department, U. of Maryland, College Park, July, 1980.

[9] Over the integers, we don't use the rows for < AND >, since $A < v_Q$ is equivalent to $A \leq v_Q - 1$, and similarly for >.

13. Grant, J. and Minker, J. Optimization in deductive and Relational Databases. In *Advances in Database Theory, Volume I*, Gallaire, H., Minker, J., and Nicolas, J-M., Ed.,Plenum Press, 1981.

14. Hall, P.A.V. Common subexpression identification in general algebraic systems. Tech. Rept. UKSC0060, IBM UK Scientific Centre, Nov., 1974.

15. Hall, P.A.V. "Optimisation of a single relational expression in a relational data base system." *IBM Journal of Research and Development 20*, 3 (May 1976), 244-257.

16. Hendrix, G. G., *et al.* "Developing a natural language interface to complex data." *ACM Transactions on Database Systems 3*, 2 (June 1978), 105-147.

17. Kim, W. Query optimization for relational database systems. Tech. Rept. UIUCDCS-R-80-1034, Computer Science Department, U. of Illinois, Urbana-Champaign, Oct., 1980.

18. King, J.J. Query optimization by semantic reasoning. Tech. Rept. STAN-CS-81-857, Computer Science Department, Stanford U., May, 1981.

19. King, J.J. QUIST: a system for semantic query optimization in relational databases. Proc. Seventh International Conference on Very Large Data Bases, Cannes, Sept, 1981, pp. 510-517.

20. Moore, R. C. Handling complex queries in a distributed data base. Technical Note 170, Artificial Intelligence Center, SRI International, Oct., 1979.

21. Nelson, C. G. An $n^{\log n}$ algorithm for the two-variable-per-constraint linear programming satisfiability problem. Technical Note STAN-CS-78-689, Computer Science Department, Stanford U., Nov., 1978.

22. Nelson, C. G. Techniques for program verification. Xerox Palo Alto Research Center, June, 1981.

23. van de Riet, R.P., *et al.* "High-level programming features for improving the efficiency of a relational data base system." *ACM Transactions on Database Systems 6*, 44 (Sept 1981), 464-485.

24. Rosenkrantz, D.J. and Hunt, H.B. III. Processing conjunctive predicates and queries. Proc. Sixth International Conference on Very Large Data Bases, Montreal, Oct., 1980, pp. 64-72.

25. Selinger, P.G., *et al.* Access path selection in a relational database management system. Proc. ACM-SIGMOD International Conference on the Management of Data, Boston, May, 1979, pp. 23-34.

26. *SQL/Data System Application Programming.* IBM, 1981. SH24-5018-0.

27. Stonebraker, Michael, *et al.* "The design and implementation of INGRES." *ACM Transactions on Database Systems 1*, 3 (Sept. 1976), 189-222.

28. Todd, S.J.P. "The Peterlee relational test vehicle - a system overview." *IBM Systems Journal 15*, 4 (1976), 285-308.

29. Welty, C. and Stemple, D.W. "Human factors comparison of a procedural and a nonprocedural query language." *ACM Transactions on Database Systems 6*, 4 (Dec. 1981), 626-649.

30. Wiederhold, G.. *Database Design.* McGraw-Hill, 1977.

31. Wiederhold, G., Kaplan, J., and Sagalowicz, D. "The Knowledge Based Management Systems project." *ACM SIGMOD Record 11*, 3 (April 1981), 26-54.

32. Wong, E. and Youssefi, K. "Decomposition - a strategy for query processing." *ACM Transactions on Database Systems 1*, 3 (Sept. 1976), 223-241.

33. Yao, S.B. "Optimization of query evaluation algorithms." *ACM Transactions on Database Systems 4*, 2 (June 1979), 133-155.

34. Youssefi, K. Query processing for a relational database system. Electronics Research Laboratory Memorandum UCB/ERL M78/3, U. of California, Berkeley, Jan., 1978.

35. Youssefi, K. and Wong, E. Query processing in a relational database management system. Proc. Fifth International Conference on Very Large Data Bases, Rio de Janeiro, Oct., 1979, pp. 409-417.

36. Zloof, M.M. "Query-by-Example: a data base language." *IBM Systems Journal 16*, 4 (1977), 324-343.