An Investigation into Logical Optimization of Relational Query Languages

Steve Talbot

Software Sciences Limited, London & Manchester House, Park Street, Macclesfield, Cheshire SK11 6SR, UK; Database consultant, 8 Lyndhurst Grove, Sittingbourne, Kent, UK

The motivation for this paper comes from an extensive study on relational database optimization techniques applied to query languages. Its aim is to give an introduction to query language optimization and to outline some of the techniques that can be used to improve the performance of a relational database. This investigation also discusses the complexity of maximal optimization and gives certain heuristic measures which lead to better performance in answering queries put to a database.

1. INTRODUCTION

The problem with high level query languages is that they allow queries to be written that take a long time to execute. This problem faces most, if not all, database management systems and may be responsible for a dramatic drop in productivity and in the use of resources. High level query languages allow a query to be expressed in several different ways to get the same result, this aids usability. It would be advantageous to rephrase queries so that they take less time to execute, without changing the original meaning of the query, this would increase productivity and use resources more efficiently.

The rephrasing of queries in this paper is called logical optimization since it is independent of the physical access mechanisms of the database. The access methods used in a database to navigate to the data can also aid optimization. This level of optimization is called physical optimization. It is important to realize that the line drawn between these two levels of optimization is not a clear cut line and that the two approaches can interact. This is borne out in a later discussion.

There are essentially four optimization techniques that span both the logical and physical levels, these are:

(a) Logical optimization:

1. Heuristic optimization: this involves transformations (rephrasing) of queries that are likely to improve execution time.
2. Deterministic optimization: this involves searching for all possible forms of a query in order to execute the best one.
3. Common subexpression recognition: this involves saving the results of an expression so as to avoid executing the same expression again.

(b) Physical optimization:

4. Access path optimization: this involves recognition of storage structures, such as indexes, orders, hash tables, etc. in order to find the fastest path to the data.

This paper addresses the problems encountered in logical optimization of single or batched queries (called the 'query mix'). In this paper a variety of techniques are explored for logical optimization in order to present a design for a general purpose, portable, machine independent optimizer for use with any relational algebra based query languages. The word 'relational' when talking about query languages and databases will be regarded as implicit.

2. LOGICAL OPTIMIZATION TECHNIQUES

2.1 Algebraic transformations

There are two types of transformation:

1. Boolean—as applied to AND, OR, NOT within JOIN and SELECT operators. This is a form of subexpression recognition.
2. Relational—as applied to JOIN, PROJECT, SELECT, UNION, INTERSECT, CARTESIAN PRODUCT, DIVIDE and DIFFERENCE.

It is useful to think of a query in terms of a query tree, much the same as one might expect from parsing a query. The leaf nodes of the tree correspond to the base relations and higher nodes correspond to a relational operator. A SEQUEL query can be directly parsed into a tree containing base relations and relational operators. An example of this is shown in Fig. 1.

Consider the following Database:

```
SUPP(SNO,SNAME,STATUS,CITY)
PARTS(PNO,PNAME,COLOUR,WEIGHT,CITY)
PROJ(PJNO,PJNAME,CITY)
SPJ(SNO,PNO,PJNO,QTY)
```

Suppose we have the following SEQUEL query:

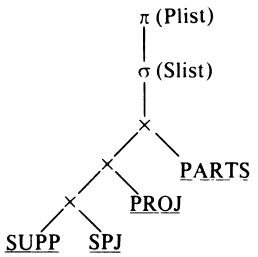
```
SELECT SNAME,PJNAME,PNAME,COLOUR,WEIGHT,
QTY
FROM SUPP,PROJ,PARTS,SPJ
WHERE SUPP·SNO=SPJ·SNO
    AND
    PROJ·PJNO=SPJ·PJNO
    AND
    PARTS·PNO=SPJ·PNO
```

This would form the following parse tree. Note that all the nodes, with the exception of the leaf nodes which are the base relations, are R.A. operators.

CCC–0010–4620/84/0027–0301 $04.50

© Wiley Heyden Ltd, 1984

Downloaded from comjnl.oxfordjournals.org at kingstonuni on September 28, 2010



Where
π ≡ PROJECT
σ ≡ SELECT
× ≡ CARTESIAN PRODUCT

Plist = (SNAME,PJNAME,PNAME,COLOUR,WEIGHT,QTY)
Slist = (SUPP·SNO=SPJ·SNO and PROJ·PJNO=SPJ·PJNO and PARTS·PNO=SPJ·PNO)

Figure 1. Transformation of SEQUEL to relational algebra.

Algebraic transformations concerning relational operators are performed by moving operators up and down the tree as appropriate, although one particular transformation concerns the removal of an operator from the query tree. The basic transformations are listed below, together with an explanation as to why they might lead to a better performance.

1. *Moving SELECTIONs down the tree as far as possible.* This more than any other transformation tends to save orders of magnitude in the time taken to process a query, since it often leads to smaller intermediate relations.
2. *Cascading SELECTIONs and PROJECTIONs.* A sequence of these operations, each taking just one operand, can be performed all at once while the relation is scanned for the operand.
3. *Combining certain SELECTIONs and CARTESIAN PRODUCT to form a JOIN.* The CARTESIAN PRODUCT can be a very costly operation to perform. If the SELECTION involves an attribute from both operand relations it is really a JOIN. If the operation also involves comparing an attribute of one of the operand relations with a constant then the SELECTION can be moved ahead of the PRODUCT, which makes the JOIN cheaper.
4. *Combining certain PROJECTIONs with a binary operation that precedes or follows it.* There is no need to run through a relation just to eliminate certain fields if we can eliminate those fields either as we create the relation, or the first time we use the relation.
5. *Deleting branches in the query tree.* This transformation involves a user query with superfluous JOINs. The user may only require information relating to a single JOIN but has inadvertently specified two JOINs. This transformation strips out the extra JOIN after the other algebraic optimization techniques have been performed.

Algebraic transformations involving Boolean operators are those that apply to the SELECT and JOIN operators of relational algebra. This type of transformation involves recognizing certain identities. This is accomplished in the same way as subexpression recognition.

2.2 Common subexpression recognition

There are three areas in which one can use common subexpression recognition. Of the three areas two are closely related to single queries whereas the third is concerned with optimization over the 'query mix'. The areas of interest are listed below, a fuller discussion is to be found in Section 4.

1. *As applied to Boolean expressions.* This is used to reduce Boolean expressions which appear in SELECTION and JOIN criteria. The idea is to eliminate redundant subexpressions thus reducing the overall size of the expression.
2. *As applied to relational expressions.* This has two functions:
   (i) The removal of Null relations, for example if we were to JOIN PARTS and SPJ over PNO but SPJ were an empty relation (Null relation) then the result would be a Null relation. Evaluating this explicitly is wasteful since it is possible to filter out the Null relations before execution.
   (ii) The recognition of common subexpressions so as to avoid duplication of effort in their evaluation. This is by far the hardest of single query techniques to implement for reasons stated in Section 4.
3. *Short term storage of views and subexpressions.* A view can be thought of as a virtual relation since it is not stored as data but as some sort of expression to be performed on the base relations (the stored data). Consider the SEQUEL query in Fig. 1; this could be transformed to a view definition by prefixing:

   DEFINE VIEW V1 (SNAME, PJNAME, PNAME, COLOUR, WEIGHT, QTY) AS . . . .

   This has the effect of telling the database system to store the view definition and to evaluate it only when specifically asked to do so.

   The short term storage of views and subexpressions has a two pronged approach, the first entails the storage of data satisfying a view definition, so as to avoid re-evaluating the view definition.

   This type of view, in general, would be periodically updated by re-evaluating the view definition and would therefore be out of date for short periods. Such a view would be queried by the user who wants to know approximately what is in the database. E.g. 'Give me an estimate of the number of Bolts in stock'. Queries of this nature need a very quick response.

   The second prong is to hold the results of certain subexpressions which are used frequently, in the query mix over a period of time so as to increase the performance of the user interface. This technique is perhaps the most difficult to implement successfully, since the criteria for choosing which subexpression(s) to hold are not well defined.

Both these techniques are still under research and need careful thought before any implementation.

3. ALGEBRAIC TRANSFORMATIONS

3.1 Heuristic optimization

The algebraic transformations listed in the previous

Downloaded from comjnl.oxfordjournals.org at kingstonuni on September 28, 2010

section form a small group of 'heuristic transformations', that is they are transformations that are likely to improve the performance of a query language interface. Heuristic transformations have been documented in Refs 1–3.

The basis for all the transformations in this paper comes from the application of certain mathematical rules which show equivalence between certain relational algebraic expressions; these rules are listed together with an algorithm for the manipulation of a query tree. The assumption we have made is that a relation is a set of mappings from attribute names to values and that two relations are equal if they have the same set of mappings.

The algorithm attempts to move SELECTIONs and PROJECTIONs as far down the tree as possible, while preserving any cascade of these operations. Cascades are constructed as one SELECTION followed by one PROJECTION. The algorithm also groups SELECTIONs and PROJECTIONs with the preceding binary operation, such as UNION, CARTESIAN PRODUCT or SET DIFFERENCE. Note that the algorithm uses only the five basic relational algebraic operations. Although the algebraic rules for the transformations include the JOIN, this can be viewed as a CARTESIAN PRODUCT followed by a SELECT. Figures 2(a) to 2(e) illustrate how a query tree is transformed using Algorithm 1.

Algorithm 1. Algebraic transformation of a query tree

Input: A tree of relational algebra operators.
Output: An optimized tree of relational algebra operators.
Method:

1. Use rule 4 to separate each SELECTION $\sigma_{F_1} \wedge \cdots \wedge {}_{F_n}(E)$ into the cascade $\sigma_{F_1}(\ldots(\sigma_{F_n}(E))\ldots)$—(a) to (b).
2. For each SELECTION use rules 4–8 to move the SELECTION as far down the tree as possible—(b) to (c).
3. For each PROJECTION use rules 3, 9, 10 and generalized rule 5 to move the PROJECTION as far down the tree as possible. Note that rule 3 may cause some PROJECTIONs to disappear, whereas the generalized rule 5 splits a PROJECTION into two PROJECTIONs, one of which can be migrated down the tree if possible. Also, eliminate a PROJECTION if it PROJECTs an expression onto all its attributes—(c) to (d).
4. Use rules 3–5 to combine cascades of SELECTIONs and PROJECTIONs into a single SELECTION a single PROJECTION or a SELECTION followed by a PROJECTION—(d) to (e).

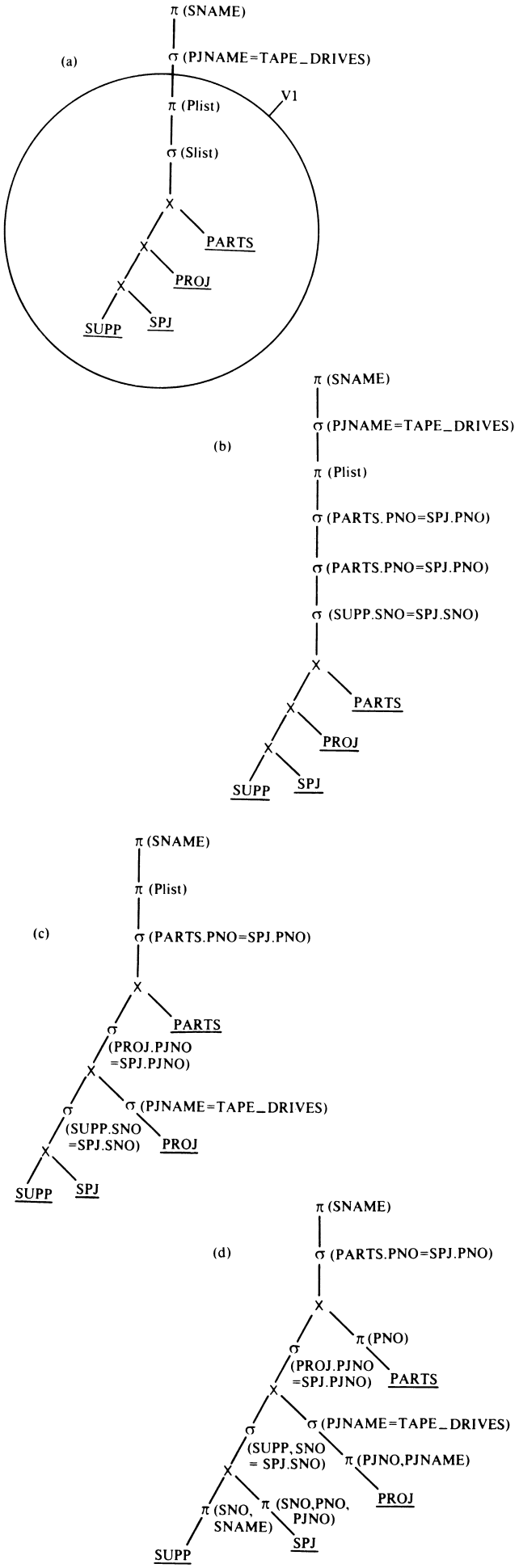
Suppose the query in Fig. 1 is the view V1, this would be done by prefixing the line

DEFINE VIEW V1 (SNAME, PJAME, PNAME, COLOUR, WEIGHT, QTY) AS

to the query in Fig 1. Now suppose we have a query on V1 as follows:

```
SELECT SNAME
FROM V1
WHERE PJNAME='TAPE_DRIVES'
```

this would give the query tree in (a):



Downloaded from comjnl.oxfordjournals.org at kingstonuni on September 28, 2010

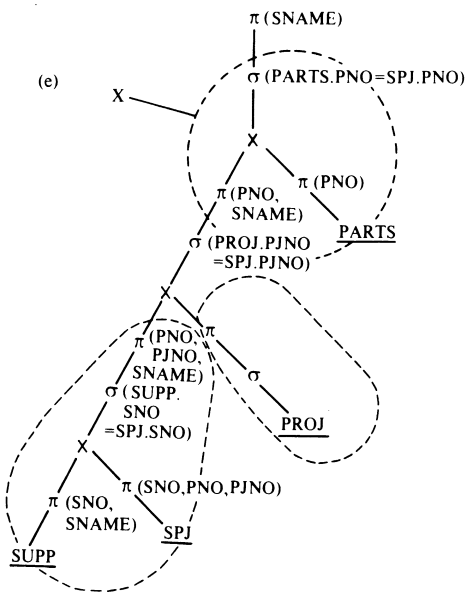


Figure 2. Tree optimization: (a) Combining with the query tree for V1; (b) Splitting up SELECTIONs; (c) Moving SELECTIONs down the query tree; (d) Cascade projections and SELECTIONs; (e) Cascade PROJECTIONs and SELECTIONs. (Normally (e) and (d) are done together.)

The transformed query tree of Fig. 2(e) can now be scanned so as to remove any superfluous JOINs. It is obvious from Fig. 2(e) that the branch labelled X can be removed from the tree since it does not participate in the user's query. In general this can be recognized by looking at the original user query and noting what is to be displayed (in the SELECT clause) and what criteria are to be used (in the WHERE clause). If none of the attributes from above exist in the JOINed relation (as in PARTS) then the relation and its JOIN are candidates for removal. The important thing to note is whether or not tuples will be gained in deleting a JOIN. This is done as follows: If the candidate relation is JOINed on its primary key to another relation on its foreign key (PARTS·PNO and SPJ·PNO), which is related to the primary key, or if the candidate is JOINed on its primary key to another relation on its primary key, the primary keys being the same for every value in their primary domains, then the JOIN can be removed. This is intuitively obvious since the removal of such a JOIN does not affect the cardinality (number of tuples) of the resultant relation. In a sense the deletion of such a JOIN is like a PROJECTION which excludes the attributes in the primary key relation. For a more formal description of this process see Algorithm 2.

Algorithm 2. Detection and deletion of superfluous JOINs

Input: A query tree
Output: A query tree.
Method:

1. Search the tree for JOINs and for each JOIN between a PRIMARY key and a FOREIGN key, or two PRIMARY keys, add the JOIN to the candidate list.
2. Look at the PROJECTION list at the top of the tree (what the user wants to see) and for each JOINED relation in the candidate list which has an attribute mentioned in the PROJECTION delete the JOIN from the candidate list.
3. The candidate list now contains a short list. For each branch containing a relation with no references (Either the PRIMARY keyed relation in a FOREIGN to PRIMARY JOIN or one of the PRIMARY keyed relations in a PRIMARY to PRIMARY), taking note of how the keys are related, remove the branch together with any operations above the branch which reference that relation.

The query tree in Fig. 2(e) exposes possible parallelism in executing the user's query. Each arm of the tree (circled) could be independently evaluated if the base relations involved were stored so as to allow independent access. Further parallelism can be achieved by pipelining the individual operations of each branch wherever possible.

There are 10 rules governing the manipulation of relational algebraic expressions.

1. *Commutative laws for JOINs and PRODUCTs.* If E_1 and E_2 are relational expressions, and F is a condition on named attributes of E_1 and E_2, then:

   1.1 $E_1 \mid\times\mid_F E_2 \equiv E_2 \mid\times\mid_F E_1$
   1.2 $E_1 \mid\times\mid E_2 \equiv E_2 \mid\times\mid E_1$
   1.3 $E_1 \times E_2 \equiv E_2 \times E_1$

   Where $\mid\times\mid_F$ is a θ JOIN, F corresponds to θ, $\mid\times\mid$ is a NATURAL JOIN $\times$ is a CARTESIAN PRODUCT.

2. *Associative laws for JOINs and PRODUCTs.* If E_1, E_2 and E_3 are relational expressions and F_1 and F_2 are conditions, then

   2.1 $(E_1 \mid\times\mid_{F_1} E_2) \mid\times\mid_{F_2} E_3 \equiv E_1 \mid\times\mid_{F_1} (E_2 \mid\times\mid_{F_2} E_3)$
   2.2 $(E_1 \mid\times\mid E_2) \mid\times\mid E_3 = E_1 \mid\times\mid (E_2 \mid\times\mid E_3)$
   2.3 $(E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3)$

3. *Cascade of PROJECTIONs.*

   $$\pi_{A_1,\ldots,A_n}(\pi_{B_1,\ldots,B_m}(E)) \equiv \pi_{A_1,\ldots,A_n}(E)$$

   where $\{A_n,\ldots,A_n\} \subseteq \{B_1,\ldots,B_m\}$.

4. *Cascade of SELECTIONs.*

   $$\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_1 \wedge F_2}(E)$$

   where $F_1 \wedge F_2$ is the logical AND of F_1 and F_2.

5. *Commuting SELECTIONs and PROJECTIONs.*

   $$\sigma_F(\pi_{A_1,\ldots,A_n}(E)) \equiv \pi_{A_1,\ldots,A_n}(\sigma_F(E))$$

   More generally, if condition F also involves attributes $B_1,\ldots,B_m$ that are not among the $A_1,\ldots,A_n$ then

   $$\pi_{A_1,\ldots,A_n}(\sigma_F(E)) \equiv \pi_{A_1,\ldots,A_n}(\sigma_{F_1}(\pi_{A_1,\ldots,A_n,B_1,\ldots,B_m}(E)))$$

6. *Commuting SELECTIONs with CARTESIAN PRODUCT.* If all the attributes mentioned in F are attributes of E_1, then

   $$\sigma_F(E_1 \times E_2) \equiv \sigma_F(E_1) \times E_2$$

   Further, if F is of the form $F_1 \wedge F_2$, where F_1 relates to E_1 and F_2 relates to E_2, then we can use rules 1, 4 and 6 to get

   $$\sigma_F(E_1 \times E_2) \equiv \sigma_{F_1}(E_1) \times \sigma_{F_2}(E_2)$$

Downloaded from comjnl.oxfordjournals.org at kingstonuni on September 28, 2010

But if F_1 relates to E_1 and F_2 relates to E_2 and E_1 then,

$$\sigma_F(E_1 \times E_2) \equiv \sigma_{F_2}(\sigma_{F_1}(E_1) \times E_2)$$

thereby pushing part of the SELECTION ahead of PRODUCT.

7. *Commuting SELECTION with a UNION.* Let $E = E \cup E_2$, then

$$\sigma_F(E_1 \cup E_2) \equiv \sigma_F(E_1) \cup \sigma_F(E_2)$$

where $\cup$ = UNION. If the attribute names for E_1 and/or E_2 differ from those of E, then the formulae F on the right must be modified to use the appropriate names.

8. *Commuting SELECTION with a SET DIFFERENCE.* Let $E = E_1 - E_2$, then

$$\sigma_F(E_1 - E_2) \equiv \sigma_F(E_1) - \sigma_F(E_2)$$

where $-$ = SET DIFFERENCE. The restrictions above apply to SET DIFFERENCE as well as to the UNION operator.

9. *Commuting a PROJECTION with a CARTESIAN PRODUCT.* Let E_1 and E_2 be two relational expressions. Let $A_1, \ldots, A_n$ be a list of attributes, of which $B_1, \ldots, B_m$ are attributes of E_1, and the remaining attributes, $C_1, \ldots, C_k$ are from E_2, then

$$\pi_{A_1,\ldots,A_n}(E_1 \times E_2) \equiv \pi_{B_1,\ldots,B_m}(E_1) \times \pi_{C_1,\ldots,C_k}(E_2)$$

10. *Commuting a PROJECTION with a UNION.*

$$\pi_{A_1,\ldots,A_n}(E_1 \cup E_2) \equiv \pi_{A_1,\ldots,A_n}(E_1) \cup \pi_{A_1,\ldots,A_n}(E_2)$$

As in rule 7, if the names of attributes for E_1 and/or E_2 differ from those in $E \cup E_2$ we must replace $A_1, \ldots, A_n$ on the right by the appropriate names.

3.2 Exact optimization

In this paper we have examined ways of transforming queries into equivalent queries which are more efficient than the original. However we have said nothing of the optimality of the transformed queries, and in general the methods outlined in Section 3.1 are only 'rules of thumb' or heuristics. It may well be the case that the transformed queries are suboptimal and that there exists a better solution.

There are several problems in finding an exact solution in query language optimization. First we have not been specific about what cost measures we are trying to optimize and therefore it is not clear what 'efficiency' is for this problem. In the previous section we have tried to achieve a time/memory trade-off; by making intermediate results small less memory is taken up and scanning becomes less of a problem, and so on. Given that we can pin-point what cost measures to use, it is interesting to consider exact optimization. There is in fact an extensive class of queries called 'conjunctive queries' for which it is possible to minimize the number of JOINs and PRODUCTs, thereby getting an exact solution. Conjunctive queries in SEQUEL are of the form 'SELECT–FROM–WHERE' such that the WHERE clauses are the conjunction (logical AND) of terms that equate two components of a tuple or equate a component to a constant. The method for doing this, greatly simplified, is to search among all those queries that are equivalent and to evaluate them according to some cost criterion so as to determine which is the least-cost query. Although it is possible to implement an optimizer for this class of queries which would give exact results it is not practical since the optimizer would be much too slow for long queries.

For a more extensive study on exact optimization of conjunctive queries see Ref. 4.

4. COMMON SUBEXPRESSION RECOGNITION

4.1 For single queries

The techniques for common subexpression recognition in single queries can be divided into two.

1. *Reduction of Boolean expressions.* As with all of the techniques, for single query reduction, we rely on the recognition and application of certain identities. These are listed in Table 1. This transformation can involve several passes over an expression in order to reduce the expression fully.

Table 1

	Expression	Transformation
1	*A* and TRUE = TRUE and *A*	*A*
2	*A* and FALSE = FALSE and *A*	FALSE
3	*A* or TRUE = TRUE or *A*	TRUE
4	*A* or FALSE = FALSE or *A*	*A*
5	*A* and *A*	*A*
6	*A* or *A*	*A*
7	*A* and not *A* = not *A* and *A*	FALSE
8	*A* or not *A* = not *A* or *A*	TRUE

2. *Reduction of relational expressions.* The techniques for relational expressions can be divided into two parts.

   (i) *The removal of Null relations.* This is similar to the transformations applied to Boolean expressions as mentioned above. The laws governing recognition and subsequent reduction are listed in Tables 2 and 3, where Null represents the empty relation, and F is a Boolean expression.

Table 2

	Expression	Transformation
9	Null UNION *A* = *A* UNION Null	*A*
10	Null INTERSECT *A* = *A* INTERSECT Null	Null
11	*A* MINUS Null	*A*
12	Null MINUS *A*	Null
13	Null PROJECT (Column list)	Null
14	Null SELECT WHERE *F*	Null
15	*A* SELECT WHERE TRUE	*A*
16	*A* SELECT WHERE FALSE	Null
17	*A* JOIN Null = Null JOIN *A*	Null
	F *F*	

Downloaded from comjnl.oxfordjournals.org at kingstonuni on September 28, 2010

Table 3

	Expression	Transformation
18	*A* UNION *A* = *A* UNION *A*	*A*
19	*A* INTERSECT *A* = *A* INTERSECT *A*	*A*
20	*A* MINUS *A* = *A* MINUS *A*	Null

The philosophy behind these rules (9 to 20) is the removal of a Null relation as an argument in an expression. Application of these rules should be done from the bottom of a large expression to the top. This bottom up reduction is expressible recursively, by first using a recursive invocation to remove Null relations from an expression, followed by other reductions until no more Null relations are left in the expression.

(ii) *Subexpression recognition.* This involves recognizing subexpressions that occur several times in an expression. At one extreme this may be a view which is referenced several times in an expression, or it may be more atomic. In either case the saving can be quite substantial. The obvious cost criterion is related to the size of the result of the subexpression, since if space is critical it may not be possible to hold the result for future use. Subexpression recognition also involves recognizing certain forms of expression that would not be picked up in (1). An artificial example of this is:

((*A* UNION *B*) MINUS (*B* UNION *A*)) UNION *B*

This gives Null UNION *B* which would be picked up in (1).

It is not enough, however, to recognize certain subexpressions first and then apply Tables 2 and 3, because there are some expressions that would still not be reduced. Another somewhat contrived example is:

((*A* UNION *B*) MINUS ((*B* UNION Null) UNION *A*)) UNION *B*

Which is equivalent to Null UNION *B* (or simply *B*). It is therefore necessary to do (i) and (ii) concurrently so as to cover all possibilities.

The process of common subexpression recognition was first used by Hall.[1,5] The method used is outlined in Algorithm 3. This method works by transforming a tree into a lattice in which no node is redundant (see Fig. 3).

Algorithm 3. Common subexpression identification, idempotency and null relation removal

Input: A query tree.
Output: A lattice.
Method:

1. Starting at the leaf nodes and working up, remove any repeated common subexpression.
2. Apply the various rules in Tables 1, 2 and 3 to remove Null relations and simple booleans.

Note: steps 1 and 2 of algorithm 3 are performed in parallel.

Suppose we have the following query

INTERSECT(UNION(EQUI-JOIN(A,B),EQUI-JOIN(B,A)),
UNION(EQUI-JOIN(C,D),EQUI-JOIN(A,B)))

then the query tree would be as follows: (We have included EQUI-JOIN instead of PRODUCT/SELECT to make the diagram simpler) A, B, C, D are all base relations.

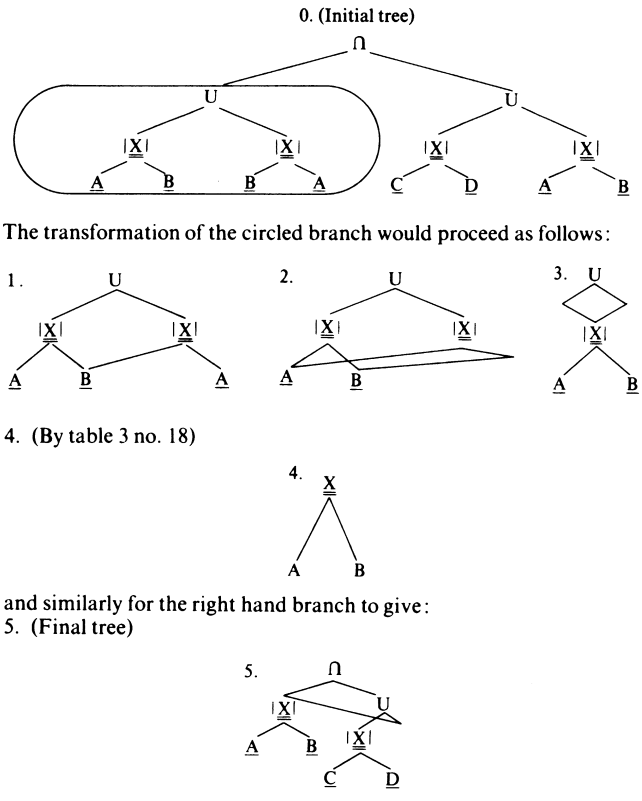


Figure 3. Transforming a tree to a lattice |x| ≡ EQUI-JOIN, ∩ ≡ INTERSECTION, ∪ ≡ UNION).

Evaluating a subexpression once, storing the result, and using it many times has a cost overhead in storing and retrieving the intermediate result. This must be compared to the saving obtained by not evaluating the expression repeatedly. We now have a complicated discrete optimization problem.

The identification of common subexpressions is a necessary step since we cannot determine whether or not to take advantage of such commonality without knowing on which expressions we can operate. We must identify a cost criterion to enable us to choose which expressions to evaluate and which expressions we should hold. In formulating a cost model we have two considerations.

(a) How much do we save by evaluating a given subexpression only once?
(b) If we have many possible common subexpressions which do we select?

A cost model for this problem is given in Ref. 1.

4.2 For a query mix

As stated in Section 2, this can be split into two sections:

1. '*Quick Views*'. The idea of a 'quick view' is to allow the user quick access to data on the understanding

Downloaded from comjnl.oxfordjournals.org at kingstonuni on September 28, 2010

that the data may be out of date. This type of view would be used by people wanting approximate answers to a query. E.g. 'Approximately how many Bolts have we in stock?' The problem in catering for such a view is that the control over what views should be stored as 'quick views' cannot be given to the DBMS since there is no way of predicting which views will be referenced in this way. The only sensible alternative is to push the responsibility onto the database administrator, and as such it is outside the scope of this paper.

2. *Temporary storage of intermediate relations.* This technique is used to compensate for a badly designed database schema, where the designer has failed to factor out common view structures. Although in its infancy and in need of much research, it seems to be promising for non-volatile databases. The principle is to look at the query-mix over a period of time or over a number of queries, either of which may be adjusted, and by comparing the queries in the mix determine any common subexpressions, which can then be evaluated and stored on a semipermanent or temporary basis. If there exists a common element in the query mix satisfying some cost criterion it can be temporarily stored for quicker reference. A possible method for this could be geared towards recognizing common branches among several query trees. The following example is not to be thought of as an implementation guide but as an illustration of what is meant by commonality of subexpressions over the query mix.

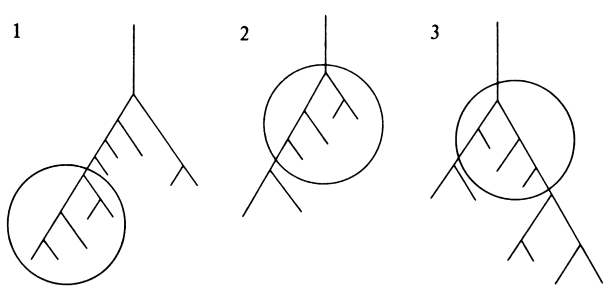


Figure 4. Commonality of subexpressions over the query mix.

Example: suppose we have the query trees shown in Fig. 4. Further, suppose that the circled branches are the same, then we could evaluate the common branch and store the intermediate result. The DBMS would have the job of determining which branches, if any, to store at any time and should be able to drop intermediate relations for new intermediate relations dynamically. In some ways this technique can be thought of as an aid to automatic database design, since it analyses what the users want, over a period of time or over a number of queries, and alters the structure of the database, albeit invisibly to the user.

There are several unanswered questions in evaluating such a mechanism.

(a) What happens if someone wants to update the data used in evaluating part of the branch? The only solution to this is to re-evaluate the intermediate relation.

(b) How do we determine when to store an intermediate relation, what criterion do we use?

These are complex questions to answer and need both theoretical and practical research, although the considerations expressed in formulating a cost model in single query subexpression recognition obviously apply.

5. THE DESIGN OF A GENERAL PURPOSE LOGICAL OPTIMIZER

5.1 Problems in designing

There are 3 areas of concern in pointing out the problems in designing a logical optimizer. These are categorized below.

1. *Problems with order.* We have looked at four techniques concerned with optimization of single queries, these are:

   (i) Algebraic manipulation of a query tree. Algorithm 1.
   (ii) Removal of Null relations from a query tree. Algorithm 3.
   (iii) Common subexpression recognition. Algorithm 3.
   (iv) Removal of superfluous JOINs. Algorithm 2.

   The question we are faced with is 'In which order should we perform these four techniques'. Ideally we want the three algorithms to be both associative and commutative, so that we can perform them in any order, but in practice we cannot guarantee this to hold.

   If we apply Algorithm 1 before we apply (iii) we can radically reduce the number of common subexpressions, since moving SELECTIONs can destroy possible subexpressions. However, because of the bigger savings that are likely to be gained by moving SELECTIONs down the tree as opposed to detecting subexpressions it is probably better to perform (i) before (iii).

   As for the order of the three algorithms as a whole, it is fairly obvious that we should perform them as follows; (i) → (iv) → (iii). The reason for performing (iv) before (iii) is that there is no point searching along a branch for common subexpressions or in order to remove nulls if that branch is to be removed. Techniques (ii) and (iii) are performed concurrently for the reasons given in Section 4 and are combined in Algorithm 3.

2. *Problems in separating the logical and physical levels.* We mentioned earlier in this paper that the line drawn between the physical and the logical levels of optimization is not a clear cut line. It is better to think of the dividing line as a wall with a window, through which the two levels may interact. There are two major problems which arise as a result of the logical optimization techniques and which affect the physical level.

   (i) The removal of PROJECTIONs in Algorithm 1 often shields existing indexes which could otherwise be used in a JOIN or SELECT operation. Therefore there is a need to look at the physical

Downloaded from comjnl.oxfordjournals.org at kingstonuni on September 28, 2010

details in order to determine whether or not we should delete a PROJECTION.

(ii) The storage of intermediate results through common subexpression recognition is dependent upon many factors, one of which is the amount of memory available; again we need to look at physical details in order to evaluate the cost model for subexpression recognition.

The alternative is not to implement those parts of Algorithms 1 to 3 that need access to the physical details.

3. *Problems of acceptability*. Until all of these techniques, for both single and multiple queries, have been implemented and tested on large databases it is not possible to assess their importance or their practical value. It is the opinion of the author that as the demand for relational databases grows we will see solutions to the optimization problem of the relational database interface.

5.2 The design

In presenting a design for a logical optimizer (Fig. 5) we have used a box like structure to represent the different algorithms and have annotated the diagram to indicate possible versions for an optimizer.

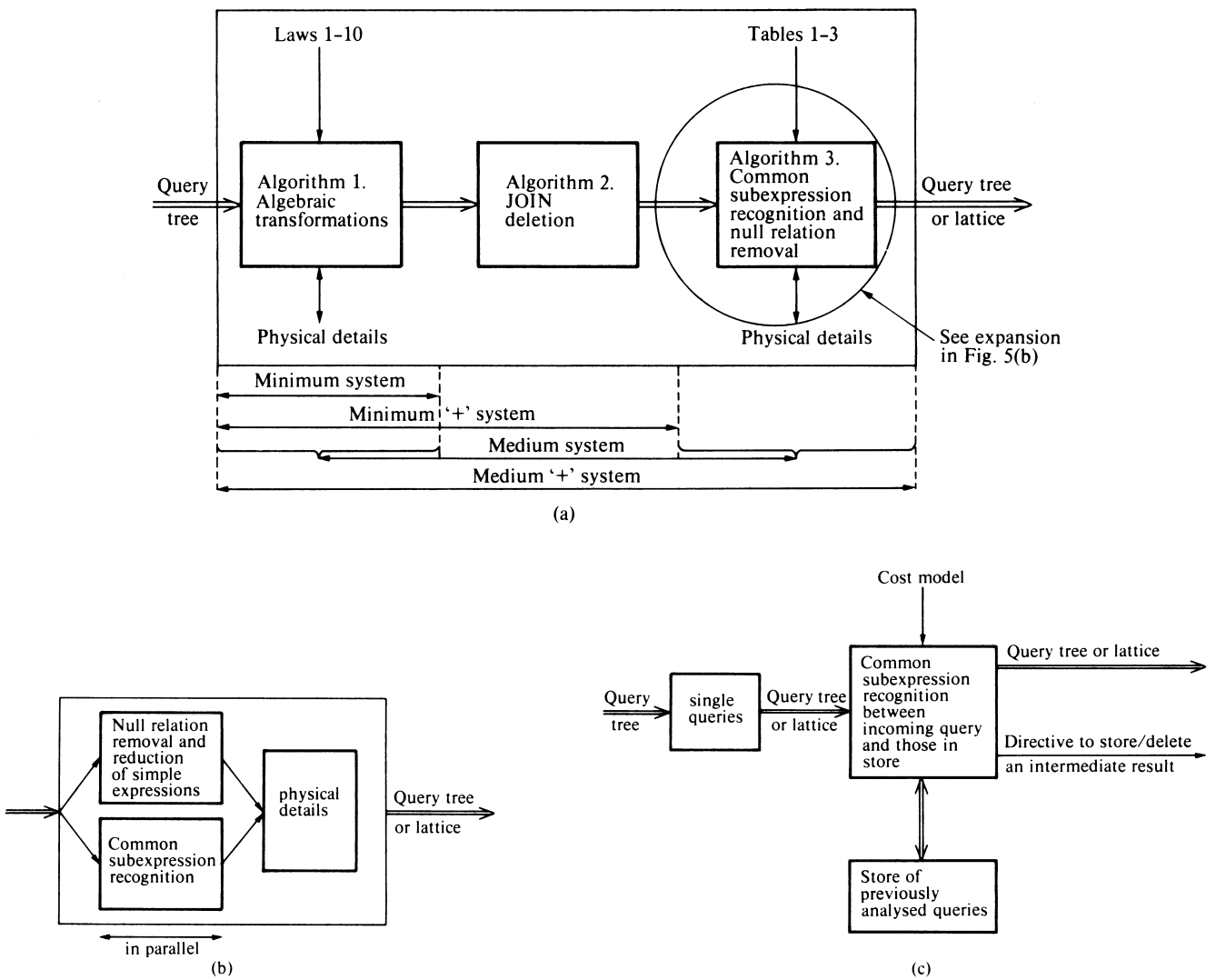


Figure 5. The design: (a) for single queries; (b) expansion of Algorithm 3; (c) for the query mix.

6. CONCLUSIONS

6.1 Summary

We have presented several techniques which optimize a relational database interface and have attempted to isolate the physical aspects of optimization in order to present a portable logical optimizer. Although it has not been possible to competely ignore the physical details of a relational database implementation we have been able to point out those techniques which rely heavily on the physical details of a database.

The techniques that have been presented and illus-

Downloaded from comjnl.oxfordjournals.org at kingstonuni on September 28, 2010

trated in the functional design of a logical optimizer, including both old and new methods for optimization, are listed below:

1. Old methods
   1.1 Algebraic transformations.
   1.2 Common subexpression recognition.
   1.3 Null relation removal.

2. New methods
   2.1 Removal of superfluous JOINs.
   2.2 Identification and storage of subexpressions over the query mix.

6.2 Further work

Obviously further work needs to be done on the newer methods for optimization, in particular a sound theoretical background needs to be established for 2.1 and 2.2 above, as well as a practical implementation of both new and old in order to assess their worth in a real D.P. environment.

Acknowledgements

The author thanks his colleagues of the data management group in the Research and Development department at Systime in Leeds especially David Iosson and Jim McKinnell.

This paper is dedicated to Joe Stevens.

REFERENCES

1. P. A. V. Hall, Optimisation of single expressions in a relational data base system. *IBM Journal of Research and Development* **20** (3), 244–257 (1976).
2. J. D. Ullman, *Principles of Database Systems,* Chapter 6, Computer Science Press, Washington D.C. (1979).
3. J. M. Smith and P. Y. Cheng, Optimizing the performance of a relational algebra database interface. *Communications of the ACM* **18** (10), 568–579 (1975).
4. A. K. Chandra and P. M. Merlin, Optimal implementation of conjunctive queries in relational data bases. *Proceedings of the 11th Annual Symposium on the Theory of Computing,* 77–90 (1977).
5. P. A. V. Hall, Common subexpression identification in general algebraic systems. *Report UKSC0060,* IBM UKSC, Peterlee, Co. Durham, England (1974).

Received June 1983

Downloaded from comjnl.oxfordjournals.org at kingstonuni on September 28, 2010