ON THE REUSABILITY OF QUERY OPTIMIZATION ALGORITHMS*

D. S. Batory

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-88-02 January 1988

*To appear in *Information Sciences* (special issue on database management systems), 1988.

On the Reusability of Query Optimization Algorithms *

D.S. Batory
Department of Computer Sciences
University of Texas
Austin, Texas 78712

Abstract

We tackle the problem of **software reusability** in this paper as it pertains to an important class of nonrecursive query optimization algorithms. We demonstrate reusability can be achieved by 1) imposing standardized interfaces and 2) expressing algorithms in a DBMS implementation-independent manner. The former is accomplished by generalizing the notion of query graphs, and the latter is accomplished by standardizing algorithm definitions in terms of query graph rewrite rules. Demonstrating algorithm reusability is an essential step toward a building-blocks technology for **extensible database systems.**

---

* This work was supported by the National Science Foundation under grant DCR-86-00738.

definitions are needed.

Consider a simple Employee-Department database and the query to list the names of employees in the Sales department in alphabetical order:

```
SELECT      Employee.Ename
FROM        Employee, Department
WHERE       Employee.D# = Department.D#
            and Department.Dname = 'Sales'
ORDER_BY    Employee.Ename
```

Let R be the above SELECT statement. For each file F listed in the FROM statement, we can define a **local predicate** Q(F,R) and a **local projection list** P(F,R). Q(F,R) is the conjunction of WHERE clauses in R that specifically reference file F but are not join clauses. For example, Q(Department,R) is "Dname = 'Sales'" and Q(Employee,R) = true, i.e., Employee has no local qualification clauses in R. P(F,R) is the list of attributes from file F that are present in the SELECT, WHERE, and ORDER_BY clauses of R. In our example, P(Department,R) = {D#, Dname} and P(Employee,R) = {Ename, D#}.

The expressions that are output by a query optimizer are compositions of five functions. (Another two will be defined in Section 4). Unless otherwise specified, each function takes stored or stream files as input, and produces a stream file as output. A **stored file** is a conceptual file in a database; a **stream file** is a stream of records that is produced by some expression. For notational simplicity, we let local predicates, local projection lists, and SELECT statement R be implicit parameters to these functions:

RET(F,O) - produce the stream of records from stored file F in O order that satisfy selection predicate Q(F,R). All attributes other than those in P(F,R) are removed from output records.

JOIN(F1,F2,J,O) - produce a stream file in O order that is the join of files F1 and F2 on join predicate J. Joined records satisfy the predicate Q(F1,R) & Q(F2,R) & J, and have the attributes P(F1,R) $\cup$ P(F2,R). (& denotes boolean conjunction and $\cup$ denotes list concatenation).

JF(F1,F2,J,O) - produce the stream of F1 records in O order that might participate in a join with file F2 on join predicate J. The attributes of output records are P(F1,R) and all records satisfy Q(F1,R).

PROD(F1,F2,O) - produce the stream file in O order that is the cross product of files F1 and F2. The attributes of output records are P(F1,R) $\cup$ P(F2,R), and all records satisfy Q(F1,R) & Q(F2,R).

END(F) - take stream file F and rearrange its records in the sequence specified in the ORDER_BY clause of R. All attributes that are not present in R's SELECT clause are removed from each record. END is a combination of two primitive functions: sort and project.

Three comments. First, each of the above operations has many different implementations. For example, JOINs can be implemented by sort-merge, nested-loops, hash-joins, and linkset algorithms. As we will explain later, the algorithms used to implement these operations are not relevant to our work. Second, the JF or **join-filter** operation is a generalization of operations that are familiar to most readers. Semijoins and Bloom-semijoins are among the implementations of JF. Although the output of a semijoin and Bloom-semijoin are rarely identical, the essential effect that records of a file have been eliminated because they cannot participate in a join is still present. Third, for the purposes of our work, we do not need to address the level of detail that is common to most research on query optimization. For example, the important optimization strategy of pushing selections before joins, cross products, and join-filters, we presume is encapsulated in the implementations of the JOIN, PROD, and JF operations. Our results on reusability can be demonstrated without introducing many of the finer points of typical optimization methods.

An **access plan** is a functional expression which defines the sequence of operations that processes a query. To specify access plans in a distributed environment, we adorn functions with superscripts to indicate the site at which a function is to be executed. As an example, the access plan $JOIN^c(JF^d(A,B,J1,O1),C,J2,O2)$ performs a join-filter of file A with another file B, and this result is then joined with file C. The join filter is executed at site d and its result is shipped to site c where the join is performed. The shipping of files to different sites is thus implicit in the notation. (For example, files A, B, and C may need to have been shipped in

A partial catalog of robust and nonrobust REDUCING_PHASE algorithms is shown below; a more complete list is given in [Bat87a]:

```
REDUCING_PHASE( G )      =>
                                        ; robust algorithms
        G                               ; identity
        SDD1( G )                       ; SDD-1 algorithm [Ber81b]
        IS( G )                         ; I-Strategy of Kang and Roussopoulos [Kan87]
        LW2( G )                        ; LaFortune and Wong algorithm with semijoins [LaF86]

                                        ; tree queries only
        BC( G )                         ; Bernstein and Chiu [Ber81a]
        YOL( G )                        ; Yu, Ozsoyoglu, Lam [Yu84b]
```

3.3 JOINING_PHASE Algorithms

JOINING_PHASE algorithms are generally robust; i.e., they can reduce all query graphs to executable expressions. The primary distinction among JOINING_PHASE algorithms is if they use heuristic or enumerative strategies for optimization. A partial catalog of JOINING_PHASE algorithms which distinguishes heuristic from enumerative optimizations is shown below. A more complete catalog is given in [Bat87a].

```
JOINING_PHASE( G ) =>
                                ; enumerative algorithms
    LW1( G )                    ; LaFortune and Wong algorithm (without semijoins) [LaF86]
    R_R*( G )                   ; Selinger, et al. [Sel79-81]

                                ; heuristic algorithms
    INGRES( G )                 ; Wong and Youseffi [Won76]
    FS( G )                     ; F-Strategy of Kang and Roussopoulos [Kan86]
    EXODUS( G, RS )             ; Graefe and DeWitt [Gra87] - RS is rule set
    STARBURST( G, RS )          ; Freytag [Fre87] - RS is rule set
```

Note that the EXODUS and STARBURST algorithms are rule-based.

3.3 Algorithm Composition

A class of query optimization algorithms is defined by the cross product of the classes of algorithms that implement the Q_GRAPH, REDUCING_PHASE, and JOINING_PHASE functions. Thus, an implementation of Q_OPT is synthesized by selecting an algorithm from each class and composing them according to eqn. (1). For example, the Bernstein and Chiu (BC) algorithm can be composed with the rule-based EXODUS(G,RS) algorithm, to yield Q_OPT(R) = EXODUS(BC(Q_GRAPH(R)),RS). Another possibility is Q_OPT(R) = INGRES(SDD1(Q_GRAPH(R))), which is a composition of the SDD-1 algorithm with the Wong and Youseffi algorithm of University INGRES. Many other combinations are possible.

4. The Rule Set

The essense of REDUCING_PHASE and JOINING_PHASE algorithms lies in the firing of a small number of query graph rewrite rules. For all but rule-based optimizing algorithms, such as STARBURST [Fre87] and EXODUS [Gra87], rules are applied in a very specific and premeditated order. In general, algorithms are distinguished by the subset of rules that they use, the order in which the rules are fired, and the operations generated by each rule.

In this section, we progressively develop the concepts that underly a rule set for nonrecursive query processing algorithms. We begin by presenting our definition of a query graph, which is central to standardized interfaces for REDUCING_PHASE and JOINING_PHASE algorithms. Without loss of generality, we assume that all join predicates in SELECT statements are equijoins and that there are no nested SELECTs. (Queries with non-equijoins have a similar treatment, and queries with nested SELECTs can be equated to queries without nested SELECTs [Kim82, Gan87]).

4.1 Query Graphs

Constructing a query graph for a SELECT statement R is a well-known process [Ber81a, Yu84a-b]. Standard query graphs have a node for each file referenced in R and a **J edge** (join edge) connecting two nodes if their corresponding files can be joined directly. [1] Nodes are labeled with their file names and edges are labeled by join predicates. Figure 4.1a is a SELECT statement and Figure 4.1b is its corresponding query graph.

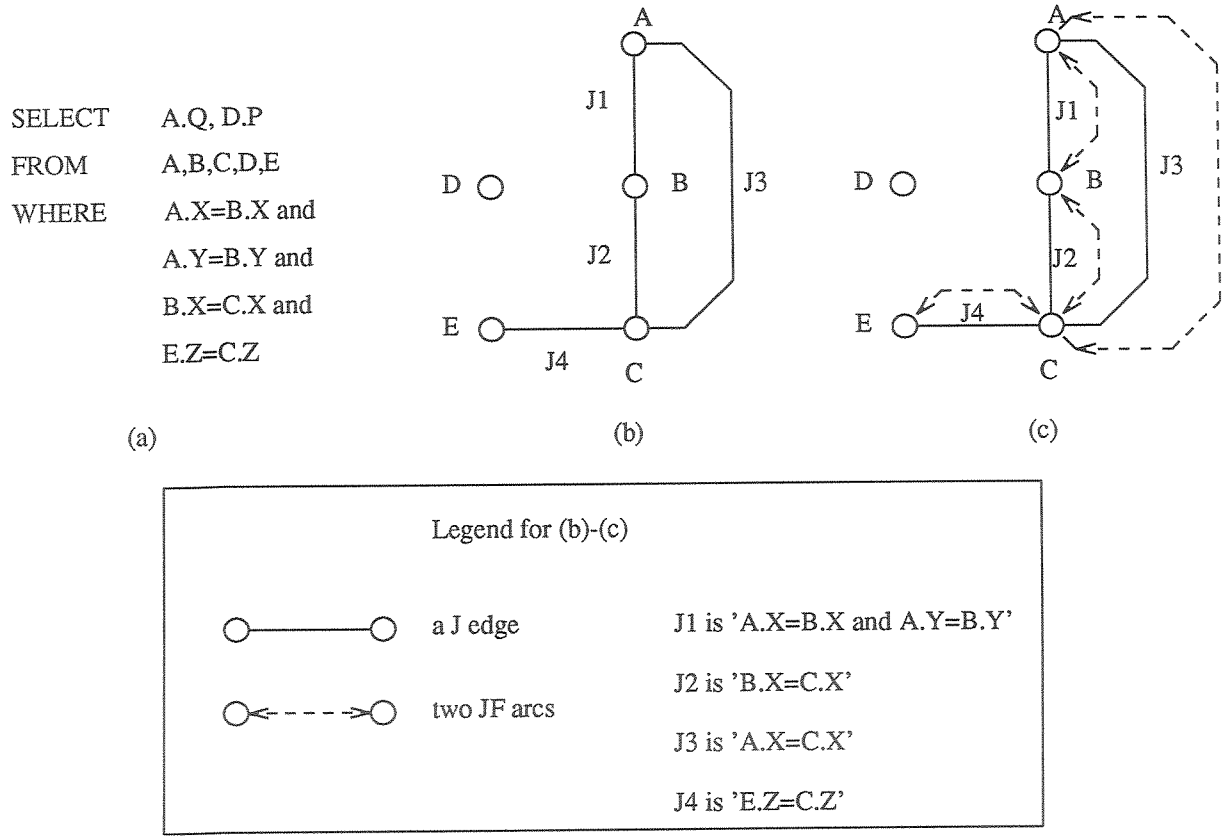

Figure 4.1 A Q_GRAPH Mapping Example

---

[1] Pairs of files that can be joined directly are listed in R's selection predicate as join clauses. Occasionally, equijoin predi-

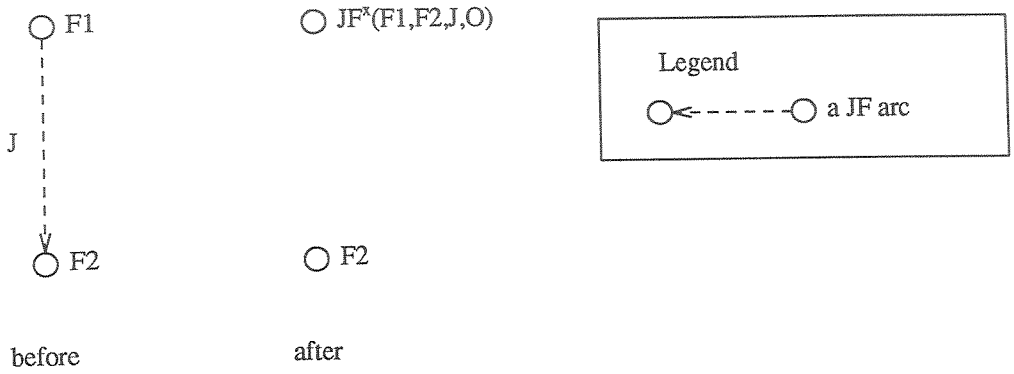

Figure 4.3 JF Arc Removal Rewrite (G2)

As in the case with local processing rewrites, (G2) suggests multiple ways to reduce file F1; the variables are the ordering $O \in \{*\} \cup P(R,F1)$ and the JF site x. x can be f1, the site of file F1, or the wild card site (a special site possibly different than f1 [Loh85]). Denoting the wild card site by **, the site restrictions are $x \in \{f1,**\}$. Note that the removal of a JF arc from a graph does not affect other arcs or edges in the graph (in particular, other arcs and edges connecting F1 and F2).

There are two generalizations of (G2) that require discussion. First, most join-filter (e.g., semijoin) algorithms assume that the join predicate J is over a single join attribute. In the case of multiple attributes, one can perform a JF over the compound attribute (as we have done), or it can be over a subset of the attributes [Ape82]. Generalizing (G2) in this manner is simple.

Second, as noted earlier, query graphs which contain cycles among J edges may require a large number of semijoins before all files are fully reduced. This result suggests that another rewrite (G2'), which does **not** remove the JF arc thereby allowing multiple (G2') rewrites to occur, would be needed. We adopt the stand of LaFortune and Wong [LaF86] that arbitrarily long semijoin chains are impractical, and that the need for a (G2') rewrite is unnecessary.

A third rewrite (G3) joins two files. Called **join edge removal,** it eliminates a J edge which connects files F1 and F2 and removes any JF arcs that may connect F1 and F2. In their place, the nodes of F1 and F2 are fused and the stream file that results from the join of F1 and F2 becomes the node's label. All other arcs and edges that were connected to F1 or F2 via other nodes are now reconnected to the fused node. Figure 4.4 shows the impact of a (G3) rewrite.

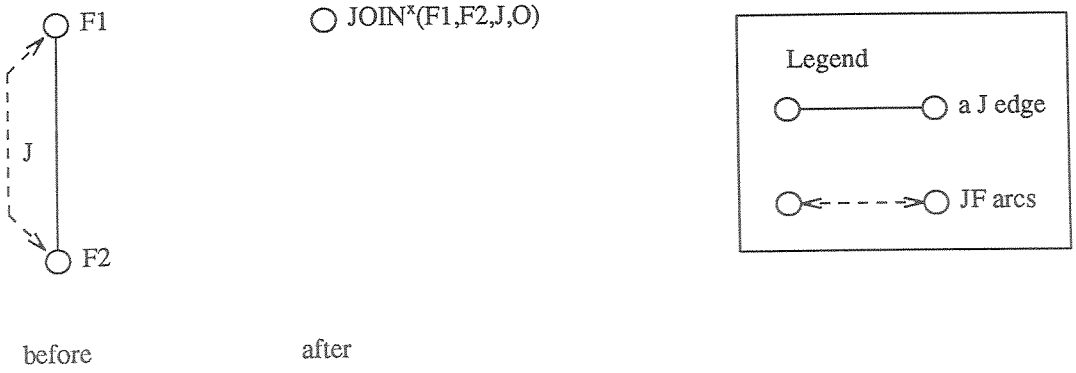

Figure 4.4 Join Edge Removal Rewrite (G3)

Figure 4.7 Final Pass Rewrite (G5)

4.3 Example

The output of query optimization algorithms are expressions that can be generated by applying rewrites (G1)-(G5) to a query graph. [3] As an example, consider Figure 4.8a which is the query graph that we encountered earlier. Suppose each file is at a different site (e.g., A is at site a, B is at site b, and so on), JF operations are performed at the reducing file site, and JOIN and PROD operations are performed at site r where the query originated.

Suppose the graph of Figure 4.8a was input to a REDUCING_PHASE algorithm which eliminated the JF arcs A→B, B→A, and E→C in this order. Figure 4.8b shows the graph after the removal of these three arcs. Note that the labels A1, B1, and E1 are expressions and represent stream files; labels C and D are names of stored files.

Suppose the graph of Figure 4.8b is then input to a JOINING_PHASE algorithm which eliminates the J edges E1–C, A1–B1, and C–B1 in this order, followed by a cross product and final pass rewrite. Figure 4.8c shows an intermediate stage of the reduction after J edges E1–C and A1–B1 have been removed. The expression that results from the complete sequence of rewrites is:

---

[3] Again we emphasize that lower-level details of query optimization, such as selecting specific indices to process queries, pushing selections before joins, etc. are implicitly captured in the implementations of the RET, JOIN, JF, PROD, and END operations. We do not introduce these details as they are unnecessary to demonstrate results on reusability for joining phase and reducing phase algorithms.

Common subexpressions are generated frequently in query optimization. Consider Figure 4.9a which shows a simple query graph with two files. Suppose the (G2) rewrite removed the JF arcs A→B and B→A in this order, yielding the successive graphs of Figures 4.9b-c. When the reduced files are joined, the following expression results:

JOIN(JF(A,B,J,O1), JF(B, JF(A,B,J,O1), J,O2), J, O3)

Note that JF(A,B,J,O1) appears twice in the expression body.

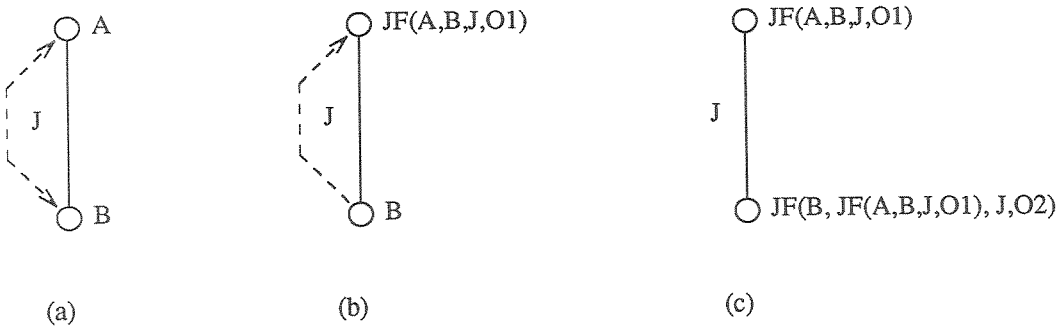

Figure 4.9 Common Subexpressions in Query Optimization

To account for temporary files and their use in common subexpression elimination, the **file spooling** rewrite is used:

$$S \rightarrow \text{STORE\_TMP}^{x}(T,S);\ T \qquad (2)$$

Eqn. (2) means stream S is stored in temporary file T at site x. All subsequent references to S are replaced with file T. The site of T can be the site at which S was generated, or it can be the wild card site. [4] Eqn. (2) can be applied to any node that has a stream file as its label.

To illustrate its use, if eqn. (2) was applied inbetween the JF arc removals of the example in Figure 4.9, the graphs of Figure 4.10 result. When the files are joined, the following expression is generated:

STORE_TMP(T, JF(A,B,J,O1)); JOIN(T, JF(B,T,J,O2), J,O3)

Note that JF(A,B,J,O1) is now evaluated once, even though its results are referenced twice in the JOIN expression.

---

[4] A notational detail which we will gloss over is the automatic generation of temporary file names. Each time we apply eqn. (2), we mean that a new temporary file is created. As readers will see in our description of the R_R* and BC algorithms, we will rely on higher-order functions. For example, a higher-order function F(E) that maps expression E to {STORE_TMP(T,E); T} suggests that no matter how many times we apply F, the same temporary file T will be used. Generating a new temporary file name each time F is applied is what we really want. To state this precisely introduces complications to our notation and in F's definition, both of which we prefer to avoid. We therefore rely on a notation which implicitly assumes that distinct instances of {STORE_TMP(T,E); T} will have distinct names for T.

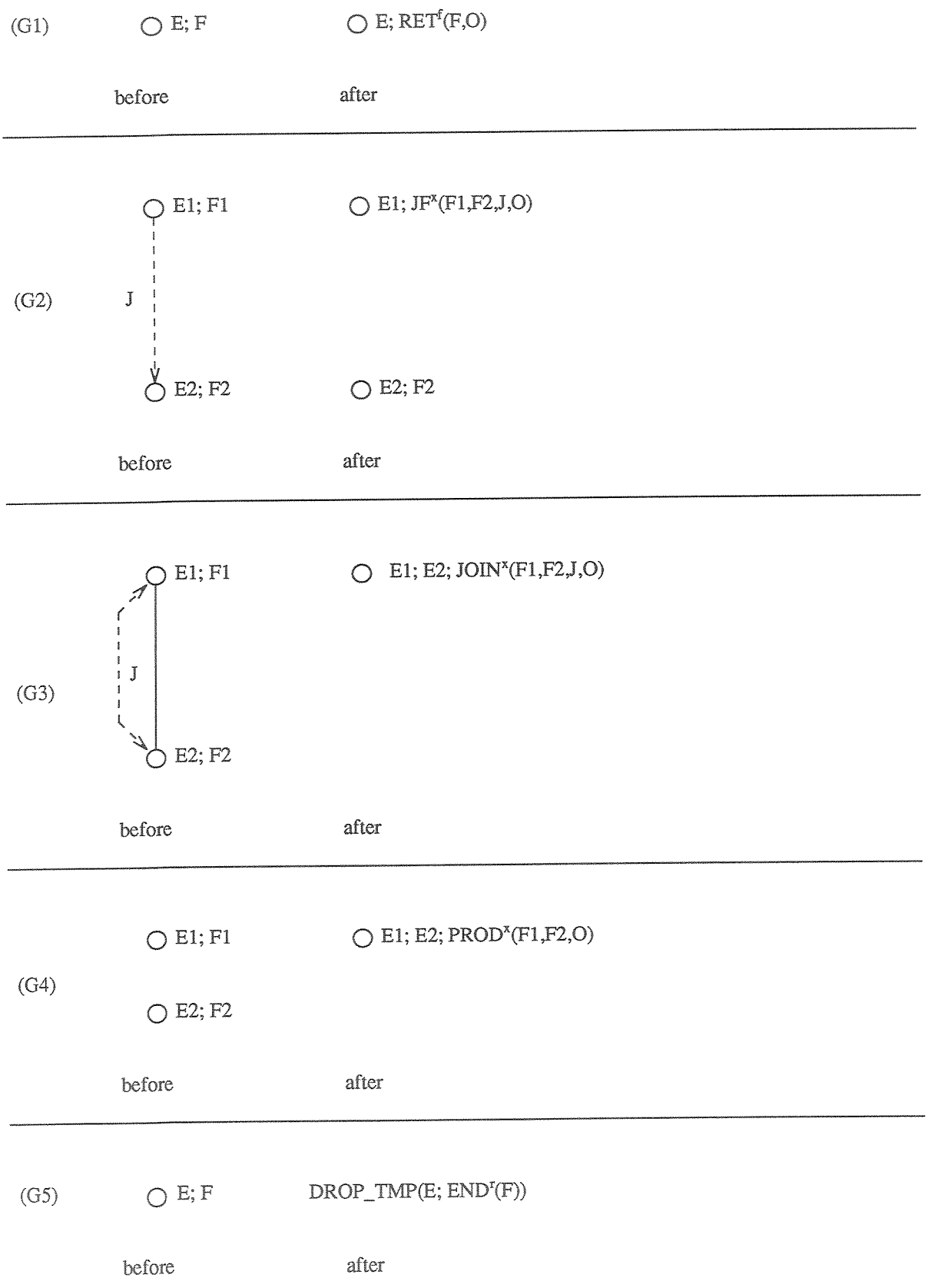

Figure 4.11 Generalized Rule Set

BC(G, RET-^{x}(_F, _O), JF-^{x}(_F1, _F2, _J,_O)) {

1) **(Local Processing).** Locally process all files in graph G by a (G1) rewrite. Select a node to be the root of G.

2) **(Leaves-to-Root).** Let i=0, N_0 be the leaves of G, and JA_0 be the set of JF arcs that terminate at the nodes of N_0. Repeat Step 2a until JA_i is empty, for some i.

   2a) **(Remove JF Arcs).** Let N_{i+1} be the set of nodes that are the origins of the arcs in JA_i. Use the (G2) rewrite to eliminate all JF arcs in JA_i. Increment i. Let JA_i be the set of JF arcs that terminate at the nodes of N_i.

3) **(Root-to-Leaves).** Let j=0, N_0 be the singleton set containing the root of G, and JA_0 be the set of JF arcs that originate from the root. Repeat Step 3a until JA_j is empty, for some j.

   3a) **(Remove JF Arcs).** Let N_{j+1} be the set of nodes that are the terminals of the arcs in JA_j. Use the (G2) rewrite to eliminate all JF arcs in JA_j. Increment j. Let JA_j be the set of JF arcs that originate at the nodes of N_j.

}.

Figure 5.1 The Bernstein and Chiu REDUCING_PHASE Algorithm

except that the ugly details and all the work of enumeration are not made explicit. [7]

The R_R* algorithm begins by choosing a node called the **sink**. Join edges are removed only if they are adjacent to the sink. When no join edges are present, cross products are considered. When the query graph has been reduced to a single node, the final pass rewrite is applied. The resulting expression is an access plan for the query graph's SELECT statement. [8]

R_R*(G, RET-^{f}(_F, _O), JOIN-^{x}(_F1, _F2, _J,_O), PROD-^{x}(_F1, _F2, _O), END-^{r}(_F)) {

1) **(Select Sink).** Nondeterministically select a node F in G and a retrieval order O. Locally process F by a (G1) rewrite. The resulting node is the **sink** F_0. Let i=0.

2) **(Main Loop).** While the query graph contains more than one node, repeat Steps 2a-b.

   2a) **(Edge Removal).** While there are J edges connected to F_i, nondeterministically select a J edge connected to F_i, a join order O_i, and a join site x_i. Eliminate the J edge by a (G3) rewrite. The resulting node is the sink F_{i+1}. Increment i.

   2b) **(Cross Product).** While there are no J edges connected to F_i and the query graph contains more than one node, nondeterministically select a node in the graph (different from F_i), an order O_i, and a cross product site x_i. Combine the selected node with F_i by a (G4) rewrite. The resulting node is the sink F_{i+1}. Increment i.

3) **(Final Step).** Graph G has been reduced to a single node F_i. Apply (G5) to F_i.

}.

Figure 5.2 The R_R* JOINING_PHASE Algorithm

---

[6] R_R* incorporates the heuristic to postpone cross products to the latest possible time. Although there is a heuristic ele-

6. Conclusions

Software reusability is a central concept to a building-blocks technology for DBMS construction. To realize such a technology forces a fundamental rethinking of the way DBMSs are currently built. Prerequisites for reusability (and plug-compatibility), namely standardized interfaces and DBMS-independent statements of algorithms, are quite foreign to current systems. This does not mean that future DBMSs won't embody these ideas.

The key to software or algorithm reusability is to find the appropriate abstractions of algorithms. In this paper, we have shown, in principle, how the reusability of an important class of query optimization algorithms can be achieved. The methods that we have implicitly used in this paper to identify algorithm abstractions are explicitly defined, and limitations discussed, in [Bat88]. We have used these ideas to show how interfaces to query optimization algorithms could be standardized.

The second requirement of finding a means to describe algorithms in a DBMS implementation-independent way involved the identification of five query graph rewrite rules that are implicitly used in query graph optimization algorithms. Each rule modifies a query graph and introduces a distinct operation on one or more conceptual files (e.g., retrieve and join). We showed how DBMS implementation-independent descriptions of query optimization algorithms could be produced by expressing these algorithms in terms of these rules. Our definitions confine DBMS-specific details to the implementation of conceptual operations, thus enforcing a layered software structure. Query processing algorithms could then be adapted to different DBMSs simply by changing the implementations of conceptual operations. Algorithm reusability was thus demonstrated.

There are other query optimization algorithms which we have not considered in this paper. These are algorithms that are embedded into RET, JOIN, JF, etc. operation implementations, and involve selecting indices to use in processing queries, choosing the fastest join method, etc. Demonstrating the reusability of these algorithms requires different techniques and concepts. In companion papers [Bat87a-88], we develop the ideas of reusability and a building-blocks technology for DBMSs further by explaining how implementations of conceptual operations for widely disparate DBMSs can be synthesized from a pool of primitive algorithms, and how performance modules for customized DBMSs can be synthesized from primitive modules.

We believe that our results in this paper on algorithm reusability are practical contributions to the realization of extensible database system technologies.

Acknowledgements. I thank Jim Barnett and the referees for their helpful comments on an earlier draft of this paper.

[Ric87] J.E. Richardson and M.J. Carey, 'Programming Constructs for Database System Implementation in EXODUS', **ACM SIGMOD 1987,** 196-207.

[Sel79] P.G. Selinger, et al., 'Access Path Selection in a Relational Database Management System', ACM **SIGMOD 1979,** 23-34.

[Sel80] P.A. Selinger and M. Adiba, 'Access Path Selection in Distributed Database Management Systems', RJ2882, IBM San Jose, 1980.

[Smi75] J.M. Smith and P.Y. Chang, 'Optimizing the Performance of a Relational Algebra Database Interface', **Comm. ACM,** 18,10 (Oct. 1975), 568-579.

[Sto86a] M. Stonebraker and L. Rowe, 'The Design of POSTGRES', **ACM SIGMOD 1986,** 340-355.

[Sto86b] M. Stonebraker, 'Object Management in POSTGRES Using Procedures', **Proc. Workshop on Object-Oriented Database Systems,** (Sept. 1986), 66-72.

[Tur79] D.A. Turner, 'A New Implementation Technique for Applicative Languages', *Software Practice and Experience* 9 (1979), 31-49.

[Ull80] J.D. Ullman, **Principles of Database Systems,** Computer Science Press, 1980.

[Won76] E. Wong and K. Youseffi, 'Decomposition - A Strategy for Query Processing', **ACM Trans. Database Syst.,** 1,3 (Sept. 1976), 233-241.

[Yu84a] C.T. Yu, Z.M. Ozsoyoglu, and K. Lam, 'Optimization of Tree Queries', **Jour. Comp. and Syst. Sci.** 29,3 (Dec. 1984), 409-445.

[Yu84b] C.T. Yu and C.C. Chang, 'Distributed Query Processing', **Computing Surveys,** (Dec. 1984), 399-433.