BUILDING BLOCKS OF DATABASE MANAGEMENT SYSTEMS*,**

D. S. Batory

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

TR-87-23 February 1988

Abstract

We present a very simple formalism based on parameterized types and a rule-based algebra to survey and identify the storage structure and query processing algorithm building blocks of database management systems. We demonstrate building block reusability by showing how different combinations of a few blocks yield the structures and algorithms of three different systems, namely System R (centralized), R* (distributed), and GRACE (database machine). We believe that codifying knowledge of DBMS implementations is an important step toward a technology that assembles DBMSs rapidly and cheaply from libraries of prewritten components.

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

**Revision of "A Molecular Database System Technology," TR-87-23, June 1987.

1. Introduction

The algorithms and storage structures of database management systems are very complicated, and their interrelationships are not well-understood. Existing DBMSs reflect the current understanding; they are ad hoc and monolithic. DBMSs are expensive to build and exceedingly difficult to modify.

The difficulty of building new DBMSs and altering existing systems has intensified the pressing need to develop DBMSs for specialized applications (e.g., VLSI CAD, temporal databases, artificial intelligence), to reduce technology transfer times, and to provide a means for experimentally evaluating newly proposed algorithms and structures without having to build significant portions of a DBMS to do so. A common solution to these problems are next-generation database systems that can accommodate new features (algorithms, structures) easily. Realizing these systems is the primary goal of extensible DBMS research [IEE87a]. A byproduct of this research is a better understanding of the interrelationships of DBMS algorithms and storage structures.

Database system research has matured to the point where a building-blocks technology to DBMS construction is feasible. The idea is to assemble DBMSs quickly and cheaply through compositions of prewritten modules. *The* fundamental barrier in developing such a technology is knowing how to decompose DBMSs in a way in which identified pieces are *demonstratably* reusable. The solution requires a very general and uncomplicated framework in which known algorithms, known storage structures, and operational DBMSs can be cast and related.

In this paper, we present a simple formalism based on parameterized types and rule-based algebras to explain the relationships between storage structures and algorithms in DBMSs. We use this formalism as a platform for surveying and relating a wide spectrum of network database structures, data models, and query processing algorithms. The importance in doing so is an identification of the algorithmic and storage structure building-blocks of DBMSs. We demonstrate building-block reusability by showing how different combinations of a few blocks leads to the structures and query processing algorithms of three different systems, namely System R [Ast76], R* [Loh85], and GRACE [Kit83, Fus86]. Our formalism provides a valuable tool in specifying the design and implementation of DBMSs.

The technical basis for our work has appeared in a sequence of models of database implementation [Bat82,85,88]. We are currently implementing this formalism in the GENESIS extensible database management system. Concurrent and independent of our research, the importance of rule-based algebras in extensible DBMSs has been recognized by Graefe and DeWitt [Gra87], Freytag [Fre87], and Lohman [Loh87b]. While all rule-based algebras are basically similar, our work is distinguished in its exposition of the important relationship of parameterized types and rule-based algebras and the validation of our model against a spectrum of existing research and operational systems.

This paper is not a tutorial on specific algorithms or storage structures; it is a presentation of DBMS building-blocks and how they fit together. A familarity with basic database concepts is assumed [Teo82, Kor86].

1.1 Preliminaries

To give relational, network, and hierarchical DBMSs an equal footing, their databases can be described in terms of files and links. A link expresses a relationship between two files, where one file is the parent and the other is the child. A link is more general than a CODASYL set in that n:m relationships are permitted, and that a file can serve as both parent and child in the same link. Links are explicit in network and hierarchical representations of databases; they are implicit in relational representations.

Links between files are normally specified with join predicates that enable a DBMS to relate records of different files automatically as records are being inserted. Some links are not associated with join predicates, and relationships between individual records must be declared manually. (This distinction is analogous to automatic and manual sets in CODASYL).

SQL* is a query language that provides a common interface to relational, network, and hierarchical databases. One way SQL* differs from SQL [Cha76] is that link names can be used in place of join predicates. Consider the database containing files P and C (for person and car). Link D relates persons to their cars and has the predicate P.C#=C.C#. To print the name of a person and the type of his/her car given that the car color is green can be expressed in two ways:

```
SELECT   P.Name, C.Type
FROM     P,C
WHERE    P.C#=C.C# and C.Color='green'
```

(a)

```
SELECT   P.Name, C.Type
FROM     P,C
WHERE    D and C.Color='green'
```

(b)

Figure 1.1 SQL* SELECT Statements

If D did not have a join predicate, which could not occur in relational databases but might in network and hierarchical, (b) would be the only way to express the query.

Another way SQL* differs from SQL is that nested SELECTs are disallowed. This causes no loss of generality; Kim [Kim82] and Ganski and Wong [Gan87] have shown that nested SELECTs have equivalent expressions as one or more nonnested SELECT statements.

We develop our formalism based on this 'generic' network model, and use SQL* to illustrate its concepts.

1.2 Overview of Formalism

Two concepts are fundamental to a DBMS building-blocks technology: **standardized interfaces** and **layered DBMSs**. Standardized interfaces provide the plug-compatibility necessary for interchangeable building-blocks. As an example, one can build a monolithic file management system that provides a standard interface to all file structures. While the interchangeability of different structures is an important and recognized goal in DBMSs, there are lower-level primitives on which all file structures rely. The implementation of these primitives should not be duplicated. A better approach is to use a layered architecture, where each layer provides the primitives on which the next higher layer is defined. A building-blocks technology requires each of these layers to have a standardized interface.

Building blocks of DBMSs are **data types** (data + operations) and **algorithms** (implementations of operations). Each data type corresponds to a layer in the above discussions, and the interfaces to its operations are standardized. Algorithms realize the operation mappings of these types. Reusability is a consequence of this formalization: each data type and its algorithms are defined independently of any DBMS in which it will be used. For this reason, it can be combined with other data types in many DBMSs.

The fundamental data types of DBMSs fall into three classes: FILE, LINK, and MODEL. FILE is the class of data types that correspond to file implementations (e.g., B+ trees, inverted files, etc.). LINK is the class of data types that correspond to implementations of links (e.g., join algorithms, CODASYL sets). MODEL is the class of data models and their corresponding data languages.

Data types that are parameterized (e.g., STACK_OF[x]) can be composed with other types (e.g., INT) to form more complicated types (e.g., STACK_OF[INT]). Many of the types within the FILE, LINK, and MODEL classes are parameterized, and combinations of them can be identified with the architectures of recognized DBMSs. Section 2 explains this relationship in detail.

Associated with each class is a small set of generic operations that all members support. It is always possible for every member of the FILE class, for example, to perform record retrieval and record insertions. By cataloging algorithms that implement the generic operations of each of type in the FILE, LINK, and MODEL classes, it is possible to codify DBMS implementation knowledge as **rewrite rules** and to express and manipulate DBMS designs as **equations**. A rule-based algebra that accomplishes is outlined in Section 3. Catalogs of query processing algorithms in MODEL, LINK, and FILE classes are presented in Sections 4-6.

Computations in centralized DBMSs are unified with the distributed and parallel computations of database machines and distributed DBMSs in Section 7. The unification is captured in the algebra by special rewrite rules and execution site assignments for algorithms.

We illustrate the utility of our model in Section 8 where specifications of DBMSs are given. In Section 9, we review research relevant to our formalism.

2. Database Systems and Parameterized Types

Generic or **parameterized types** were introduced many years ago as a way to simplify software development and to promote software reusability [Lis77, Gog84]. A classical example of a parameterized type is STACK_OF[x]. One can define and implement stacks and stack operations independently of the objects that are placed on a stack. Parameterizing the STACK_OF type and the **module/layer** that implements the STACK_OF algorithms attains this independence. So if MATRIX is a type, STACK_OF[MATRIX] defines a type that is a stack of matrices. The implementation of the composite type is a composition of the STACK_OF and MATRIX modules. Because both types and their modules were developed independently, they are building blocks of other systems.

Implementations of database management systems can also be viewed as a composition of types. Consider the types BPLUS, ISAM, and HEAP. Instances of these types are specific B+ tree, isam, and heap file structures.

Let FILE denote the class of all file implementations, of which BPLUS, ISAM, and HEAP are members. To promote uniformity and module interchangability, all members of the FILE class support *exactly* the same interface. This is possible, as one can always retrieve records, insert records, delete records, etc. from a file regardless of how it's records are stored. By imposing a standardized interface, a program that references a file F whose type is BPLUS will still work if F's implementation is changed to HEAP. [1] (Providing, of course, that a type conversion - e.g., file unload and reload - takes place).

FILE types can have parameters. Let INDEX[df:FILE, xf:FILE] be the type that specifies a parameterized implementation of an inverted file. The notation df:FILE means that parameter df can be assigned one of a set of types belonging to the FILE class. When a file is of type INDEX, the INDEX module/layer maps the file (henceforth called an **abstract file**) to a data file and zero or more index files (henceforth called **concrete files**). The key idea behind the parameterization is that the data and operation mappings of INDEX *do not rely* on the implementations of the concrete data file and concrete index files. For this reason, the file types of the data and index files are parameters to INDEX.

A common implementation of indexed files has data files implemented as heaps and index files as B+ trees. This corresponds to the type expression INDEX[HEAP,BPLUS]. Assigning different file structures to the data file and index files yields different inverted file implementations.

Each FILE type encapsulates the data and operation mappings from an abstract file to one or more concrete files. As a few examples, ENCODE[ef:FILE] maps an unencoded/uncompressed file to an encoded/compressed file whose implementation is ef; XPOSE[sf:FILE] maps an untransposed file to a transposed file, where sf is the subfile implementation; and HPART[pf:FILE] horizontally partitions an abstract file, where pf is the implementation of a concrete file partition. The class of FILE implementations is quite large, having hundreds of members. A comprehensive list of known file structures and file mappings is given in [Bat84-85].

FILE is a class of building blocks. LINK is another. Let LINK be the class of all link implementations, whose members include relational implementations, such as ALG (join algorithms) and LINDEX[li:FILE] (link indices [Val87], [Hae78]), and CODASYL set implementations, such as PARRAY (pointer array) and RLIST (ring list). (Note that the LINDEX generates files, called **link indices**, as a result of its mapping of a link. The implementation of these files is specified by parameter li).

It is often the case that links are generated in the abstract-to-concrete mappings of files. The inverted files of INDEX, for example, use links to connect index files with data files. A more general parameterization of INDEX would be INDEX[df:FILE, xf:FILE, k:LINK], where k would specify the implementation of index-to-data file links. Common implementations for k are pointer arrays and linear lists [Teo82].

A third class of building blocks is MODEL. A network interface to databases is archaic by today's standards. DBMSs therefore provide data models and data languages as their user interfaces. A MODEL type encapsulates the data and operation mappings that occur between a DBMS's user interface and its representation of databases as networks. QUEL[f:FILE, n:LINK] is a member of the MODEL class. It represents the

---

[1] The imposition of a standardized interface is evident conceptually, but it is VERY rare to find DBMSs implemented in this way. The author is not aware of an exception where the interface to conceptual files matches that of internal files.

data model and data language of INGRES, and is parameterized by f and n, which are respectively the implementations of the underlying files and links of INGRES databases [Sto76]. Other members of MODEL are SQL[f:FILE, n:LINK], the data model and data language of System R [Ast76], DBTG[f:FILE, n:LINK], the primitive data model and data language of CODASYL [Kor86], and NETWORK[f:FILE, n:LINK], a programming language interface to FILE and LINK types.

Three comments. First, just as STACK_OF and MATRIX can be recognized as building blocks of other systems, so too can the instances of FILE, LINK, and MODEL classes be recognized as building blocks of database management systems. INGRES, for example, corresponds to the composition:

QUEL[INDEX[dfile, ifile, PARRAY], ALG]
where dfile = { HEAP, ISAM, HASH, COMP_ISAM, COMP_HASH }
and ifile = { HEAP, ISAM, HASH, COMP_ISAM, COMP_HASH }

That is, INGRES presents a QUEL front-end, maps relations to inverted files and implements relation-to-relation links by join algorithms. The data file and index files of inverted files can be stored in one of five structures: heaps, isam, hash, compressed isam, and compressed hash. (The latter two apply compression techniques as part of their file structure algorithms. The actual file structure that is used for a given data file or index file is specified by the physical database directive MODIFY-TO [Sto76]). Index-to-data links are implemented as pointer arrays.

As another example, RAPID, a statistical DBMS built by Statistics Canada in the mid-70's to process the Canadian census, has no data model front-end (i.e., RAPID has only a programming language interface), encodes conceptual files, uses join algorithms to link conceptual files together, creates indices over selected encoded fields, stores index files in B+ trees, transposes the data file, and stores subfiles in heaps. Pointer arrays connect index files to data files [Tur78]. This corresponds to the type expression:

NETWORK[ENCODE[INDEX[XPOSE[HEAP], BPLUS, PARRAY]], ALG]

Taking other combinations of FILE, MODEL, and LINK modules yields other database systems. Additional examples are given in [Bat84-85,87a,88].

Second, note the extensibility of describing DBMS implementations as type expressions. New file implementations, link implementations, and data model/data languages are constantly being invented. They are accommodated easily as new FILE, LINK, and MODEL types.

Third, the ANSI/SPARC notions of a conceptual level and an internal level are present in our composite types [Kor86]. For every type expression, the most abstract files are **conceptual** and the most concrete files are **internal**. In the case of INGRES, relations correspond to the conceptual files (i.e., the abstract files of QUEL) and the files stored in HEAP, ISAM, HASH, COMP_ISAM, and COMP_HASH structures are internal. For RAPID, unencoded files are conceptual and the files stored in HEAP and BPLUS structures are internal.

3. A Rule-Based Algebra

We lay the groundwork for our rule-based algebra in this section. We present its general framework, a means by which algebraic expressions can be simplified, and rewrite rules which can be used to generate new algorithms.

3.1 The Algebraic Framework

Algorithms process **streams** of records. Records can be ordered within streams, and records can be duplicated. Computations on streams are accomplished by **functions**, where a function represents either an **algorithm** or an **operation**. The distinction is that algorithms implement operations.

Files are distinguished from streams in that they are stored (rather than computed) sequences of records. Files and streams may be interchangeable as arguments to some functions. To promote uniformity, we will henceforth refer to them as **stored files** and **stream files**.

In general, there are many algorithms for a given operation. For example, the sort operation can be implemented by a bubble sort algorithm, a quicksort algorithm, a radix sort algorithm, etc. We express the relationship between an operation O and the class of algorithms $A_1, A_2, \cdots$ that implement O as algebraic identities or **catalogs** of the form:

```
O   =>   A1          ; algorithm #1
         A2          ; algorithm #2
         ...
```

Catalogs promote the extensibility and interchangeability of algorithms. If a new algorithm A_n for operation O is invented, we simply add it to the catalog for O. A_n can then be used in the design and specification of new DBMSs. This is algorithm **extensibility**. If algorithm A_i does not exhibit the desired performance characteristics, it could be swapped with another implementation of O which might yield better performance. This is algorithm **interchangeability**.

Algorithms are either atomic or nonatomic. The most primitive algorithms, those whose decompositions are not considered interesting, are **atomic**. Compositions of atomic algorithms yield more complicated or **nonatomic** algorithms. Determining which algorithms are atomic is subjective; convenience is the best guide.

Algorithms are either robust or nonrobust. A **robust** algorithm processes all instances of its operation; **nonrobust** algorithms do not. All popular sorting algorithms are robust. However, one can imagine a sorting algorithm that takes advantage of a preordering of its input elements. Because it works only under special conditions, it is nonrobust. In general, robust algorithms can always be used as operation implementations, but using a nonrobust algorithm is legal only if restrictions are met. We encounter nonrobust algorithms in Sections 4, 5, and 6.

For the above framework to be useful, there must be agreement on what operations to catalog. Describing DBMSs as compositions of generic types provides the answer. The interface standardization of the FILE and LINK classes identifies a small set of operations. One can retrieve, insert, delete, etc. records from files, and one can traverse (i.e., compute joins), connect, and disconnect records from links. The MODEL class, in contrast, does not have a standardized interface; each MODEL type supports its own (but fixed) set of operations. Another source of operations are those that are performed on record streams (e.g., sort, filter, merge, split, etc.). The number of operations on streams is unrestricted. The file, link, and stream operations that we use in this paper are listed with their definitions in Appendix 1. We show in [Bat88] how new operations on FILEs and LINKs, beyond those in the fixed set, can be admitted without sacraficing the simplicity of this framework.

The size of individual catalogs is potentially very large. We appeal to the **transformational approach** [Par83, Fre86] which enables a large class of algorithms to be expressed as a smaller class plus some rewrite rules. The idea is to enumerate identities which can transform one algorithm into another. Algorithms listed in the smaller class are **basic**, and those that can be derived via identities are **variants**. Generally, basic algorithms are more 'simple' than their variants.

Given the existence of catalogs (which themselves are rewrite rules), the types of additional rewrites needed is limited. Those that we will use are very general; they express relationships among *operations* and do not reference specific algorithms. (Again, relationships among algorithms are expressed by catalogs). Let $E_0, E_1, \cdots$ be equivalent expressions formed by the composition of operations, where E_0 is the 'simplest'. We express transformational rewrite rules in our algebra as:

E_0 <-> E_1 ; rewrite rule #1

E_2 ; rewrite rule #2

...

Note that <-> implies bidirectionality (i.e., E_1 can be rewritten as E_0 and vice versa).

As an example, let RET(F,Q,O) be the operation that retrieves records from file F in O order that satisfy predicate Q. Let SORT(S,O) be the operation that sorts records in stream S in O order, and let FILTER(S,Q) eliminate records from stream S that do not satisfy query Q. The following rewrites capture relationships among the RET, SORT, and FILTER operations:

RET(F, Q and P, O) <-> SORT(RET(F, Q and P, O1), O) (R1)

FILTER(RET(F, Q, O), P) (R2)

(R1) states that retrieving records from file F in O1 order and then resorting into O order is the same as retrieving records from F in O order. (R2) states that retrieving records and filtering is equivalent to retrieving records with a more restricted predicate. Additional rewrites are given Sections 3.3, 5.4, 6.2, and 7.

3.2 File Characteristics

Stored and stream files have characteristics that are essential in manipulating and simplifying nonatomic algorithms. These characteristics are ordering, record membership, and query fragment.

Let S be a stream file. ORDER(S) is the order in which records of S are sequenced. A random order is indicated by ORDER(S)=*. MEMBER(S) is the predicate that all records of S satisfy.

Let F be a stored file. ORDER(F) is the order in which records of F are stored. MEMBER(F) is defined as MEMBER(S). Unless F is a partition of some larger file, MEMBER(F)=true.

Stored files have an additional characteristic. Selection predicates of SQL* SELECT statements can be decomposed into a conjunction of join predicates and subpredicates over individual stored files. Let Q(F,R) be the subpredicate, or **query fragment**, of SELECT statement R over file F. Using the SELECT of Figure 1.1 (reproduced below) as an example, Q(C,R)=(C.Color='green') and Q(P,R)=true.

```
SELECT   P.Name, C.Type
FROM     P,C
WHERE    D and C.Color='green'
```

Ceri and Pelagatti were among the first to recognize the utility of assigning characteristics to files [Cer84]. They showed how distributed query processing algorithms could be simplified using record membership and query fragment characteristics. We will show how these same ideas have been used to simplify designs of database machines in Section 7. Rewrites that take advantage of file characteristics are listed in Appendix 2. Others are found in [Cer84, Fre86].

3.3 Conversion Rewrite Rules

Some functions have arguments that make no distinction between stored and stream files. When a file argument can be either, it is possible to replace a stored file with its equivalent stream expression (and vice versa) without altering the function's result. Rules that accomplish this exchange are **conversion rewrites**.

Let G be a function which accepts a stored or stream file as its argument. Let F be a stored file with query fragment Q (an abbreviation of Q(F,R)) and let S be a stream file. The basic conversion rewrites are:

G(F)	<->	G(RET(F,Q,O))	; stored-to-stream	(A1)
G(S)	<->	STORE_TMP(T,S,O); G(T)	; stream-to-stored	(A2)

STORE_TMP(T,S,O) stores stream S in temporary file T in O order. Note that the order O in which records are read from F or stored in T is an optimization variable. Also note in (A2) that the expression 'A;B' means execute A before B.

Combinations of (A1) and (A2) can be taken to produce other rewrites:

G(F)	<->	STORE_TMP(T, RET(F,Q,O1), O2); G(T)	; stored-to-stored	(A1*A2)
G(S)	<->	STORE_TMP(T,S,O1); G(RET(T,true,O2))	; stream-to-stream	(A2*A1)
		STORE_TMP(T, RET(F,Q,O1), O2); G(RET(T,true,O3))	; stored-to-stream2	(A1*A2*A1)

Further compositions of (A1) and (A2) yield redundancies, so the number of distinct conversion rewrites is limited.

3.4 Recap

In the following successive sections, we use this algebraic framework to catalog and unify query processing algorithms in MODEL, LINK, and FILE types. By doing so, we identify algorithmic building blocks of DBMSs.

4. MODEL Retrieval Algorithms

Let R be a retrieval statement in the data language of MODEL M. (R could be a SQL* SELECT, a non-nested SELECT statement in SQL[ ], a RETRIEVE statement in QUEL[ ], etc.). The most abstract description of query processing in database systems is captured by the following expression:

EVAL(Q_OPT(R))

Q_OPT is the **query optimization** operation which maps R to an executable expression. EVAL(E) executes expression E. Different DBMSs implement Q_OPT in different ways. To survey recognized implementations, we note that Q_OPT is a well-known, albeit intuitive, composition of three lower-level operations:

Q_OPT(R) => JOINING_PHASE(REDUCING_PHASE(Q_GRAPH(R)))

Q_GRAPH:R→G maps retrieval statements the data language of M to query graphs (see [Ber81a, Yu84a-b]). REDUCING_PHASE:G→G maps query graphs with unreduced files to graphs with reduced files, and JOINING_PHASE:G→E maps query graphs to executable expressions. All of these functions rely on a common definition of query graphs. (It is this standardization which enables their implementations to be plug-compatible). The details of this query graph definition are not essential to this paper, nor are descriptions of the internal mechanics of Q_GRAPH, REDUCING_PHASE, and JOINING_PHASE algorithms. Such details are found in [Bat87c].

The catalog of Q_GRAPH(R) has one algorithm for each MODEL type M:

```
Q_GRAPH( R )  =>  SQL_GRAPH( R )         ; if M is SQL[ ]
                  QUEL_GRAPH( R )        ; if M is QUEL[ ]
                  . . .
```

In the following sections, recognized REDUCING_PHASE and JOINING_PHASE algorithms are surveyed.

4.1 REDUCING_PHASE Algorithms

REDUCING_PHASE represents the class of algorithms that map query graphs with unreduced files to graphs with reduced files. A file is **reduced** if records (and fields) that are unneeded in processing a query have been eliminated. Reductions are accomplished by RET operations (e.g., selections and projections), JF operations, and JOIN operations.

The JOIN(F1,F2,J,O) operation forms the join of files F1 and F2 over predicate or link J and produces joined records in O order. Nested loops and hash joins are among its implementations. The **jfilter** operation, JF(F1,F2,J,O), eliminates records from file F1 that cannot participate in a join with F2. J is the joining predicate or link, and the selected F1 records are returned in O order. Semijoins and Bloom semijoins are among JF implementations [Mac86a-b]. (JF is a generalization of operations that are familiar to most readers; it is a basic operation in our algebra).

With this in mind, three subclasses of REDUCING_PHASE algorithms can be identified: those that deal with JF and RET operations only, those that also consider JOINs, and those that use rule-based reductions:

```
REDUCING_PHASE( G )  =>
        NO_JOINS( G )              ; JF and RET operations only
        WITH_JOINS( G )            ; JOIN, JF, and RET operations
        RULE_REDUCE( G, RS )       ; not yet investigated. RS is the rule set
```

NO_JOINS represents the most familiar class of REDUCING_PHASE algorithms, namely those that perform semijoin reductions. Many of these algorithms are not robust as they are applicable only to special classes of

query graphs, e.g., trees, chains, and simple cliques: [2,3]

```
NO_JOINS( G )  =>                                ; robust algorithms
                    G                            ; identity
                    HOME( G )                    ; transfer all files to query site
                    SDD1( G )                    ; SDD-1 algorithm [Ber81b]
                    AHY( G )                     ; Apers, Hevner, Yao [Ape83]

                                                 ; tree queries only
                    BC( TREE( G ) )              ; Bernstein and Chiu [Ber81a]
                    YOL( TREE( G ) )             ; Yu, Ozsoyoglu, Lam [Yu84a]

                                                 ; chain queries only
                    CBH( CHAIN( G ) )            ; Chiu, Bernstein, and Ho [Chi84]

                                                 ; simple clique only
                    HY( SIMPLE_CLIQUE( G ) )     ; Hevner and Yao [Hev79]
```

The first two NO_JOINS implementations, G and HOME(G), do not perform JF operations. G is the identity function. It is the most common REDUCING_PHASE implementation used in operational DBMSs. HOME(G) is the simple strategy of locally processing the files of a query (by RET operations), and transferring the results for subsequent joining to the site at which the query was issued.

G and HOME are among the few NO_JOINS implementations that are robust. SDD1 and AHY are two others. The remaining have limited applicability, as special difficulties arise when reducing query graphs with cycles [Ber81a, Yu84b]. The TREE, CHAIN, and SIMPLE_CLIQUE functions represent classes of algorithms that map query graphs (typically with cycles) to equivalent tree, chain, or simple-clique graphs. (Equivalence here means that the same output results by executing the queries represented by either graph). Although implementations of CHAIN and SIMPLE_CLIQUE are not known, implementations of TREE have been discussed:

```
TREE( G )  =>  KG( G )       ; Kambayashi and Yoshikawa [Kam83]
               KAM( G )      ; Kambayashi [Kam85]
               GS( G )       ; Goodman and Shmueli [Goo82]
```

Two papers have recently proposed that JOINs also be considered in the reduction phase. This leads to the second subclass of REDUCING_PHASE algorithms, WITH_JOINS:

---

[2] Algorithms to test whether or not query graphs are tree-equivalent are described in [Yu84b].

[3] A simple clique is a clique where the label on all join edges is the same.

```
WITH_JOINS( G )   =>   LW2( G )      ; Lafortune and Wong algorithm with semijoins [Laf86]
                       IS( G )       ; I-Strategy of Kang and Roussopoulos [Kan87]
```

Both of these algorithms are robust; IS is driven by heuristics and LW2 is enumerative.

A third subclass of REDUCING_PHASE is RULE_REDUCE, which represents the class of rule-based reduction algorithms. Presently there are no implementations of RULE_REDUCE, although we note that rule-based algorithms for the joining phase have recently been proposed and could be modified easily to become RULE_REDUCE implementations. [4] Further discussion is given in Section 4.4.

4.3 JOINING_PHASE Algorithms

JOINING_PHASE represents the class of algorithms that transform query graphs into executable expressions. The expressions that JOINING_PHASE algorithms output are compositions of RET, JF, JOIN, and PROD operations. The PROD(F1,F2,O) operation forms the cross product of files F1 and F2 and sorts the result in O order.

There are two subclasses of JOINING_PHASE algorithms: JNP and DECOMPOSE. JNP algorithms are in some sense primitive as they reduce a graph directly to a single node. On the other hand, DECOMPOSE algorithms identify subgraphs to be reduced by JOINING_PHASE algorithms. By shrinking subgraphs to single nodes, and iteratively applying DECOMPOSE algorithms on the shrunken graph, a query graph can be reduced. When a single node is reached, the label of the node is the expression (i.e., output) of the reduction. Using the underscore _ to denote a formal variable, we have:

```
JOINING_PHASE( G )   =>   JNP( G )                            ; primitive algorithms
                          DECOMPOSE( G, JOINING_PHASE(_) )    ; subgraph reduction algorithms


JNP( G )   =>                          ; robust algorithms
             SYS_R( G )                ; System R, Selinger et al. [Sel79]
             RSTAR( G )                ; R*, Selinger et al. [Sel80]
             RR( G )                   ; Rosenthal and Reiner [Ros82]
             LW1( G )                  ; Lafortune and Wong algorithm (without semijoins) [Laf86]
             FS( G )                   ; F-Strategy of Kang and Roussopoulos [Kan87]
             RULE_JOIN( G, RS )        ; rule-based optimization, RS is rule set

                                       ; tree queries only
             BBC( TREE( G ) )          ; Baldissera, Brachi, and Ceri [Bal79]
```

---

[4] RULE_REDUCE algorithms behave like NO_JOINS algorithms if the set of rules does not contain JOIN operators, else they behave like WITH_JOINS algorithms. Because of this inherent ambiguity, we chose to place RULE_REDUCE algorithms in a separate class.

RULE_JOIN(G,RS)	=>	CH(G, RS)	; Chu and Hurley [Chu82]
		EXODUS(G, RS)	; Graefe and DeWitt [Gra87]
		SO(G, RS)	; Shenoy and Ozsoyoglu [She87]
		STARBURST(G, RS)	; Lohman [Loh87b]

DECOMPOSE(G, JOINING_PHASE(_)) =>

	; **robust algorithms**
DECOMP(G, JOINING_PHASE(_))	; Wong and Youseffi [Won76]
SUBSTITUTION(G, JOINING_PHASE(_))	; Wong and Youseffi [Won76]
D_INGRES(G, JOINING_PHASE(_))	; Distributed INGRES: Epstein, et al. [Eps78]
ES(G, JOINING_PHASE(_))	; Epstein and Stonebraker [Eps80]

Wong and Youseffi [Won76] proposed the first two decomposition algorithms: SUBSTITUTION and DECOMP. SUBSTITUTION simplifies a query graph G by selecting the node containing the smallest file F, and performing tuple substitution for each tuple/record in F. A sequence of subgraphs is generated, one subgraph for each tuple in F. If the query is over a single file, SUBSTITUTION processes the query directly. [5] DECOMP decomposes a query graph G into irreducible components. The query processing algorithm of University INGRES is expressable as a pair of mutually recursive functions:

UINGRES(G) = DECOMP(G, SUB(_))

SUB(G) = SUBSTITUTION(G, UINGRES(_))

That is, query graphs are decomposed by DECOMP, and each component is simplified by SUB. The sequence of graphs that result from substitution are decomposed recursively by UINGRES. When graphs have been reduced to a single node, SUBSTITUTION evaluates the query directly.

4.4 Recap and Observations

The result of mapping a retrieval statement R in the data language of MODEL M to a network database interface is an expression composed of RET, JF, JOIN, and PROD operations. *No reference is made in this expression to implementations of RET, JF, JOIN, and PROD operations; e.g., indices, join algorithms, etc.* As an example, the SQL* SELECT statement of Figure 1.1b (which we reproduce in Figure 4.1a) could be processed by many possible expressions. One such expression is given in Figure 4.1b:

```
SELECT  P.Name, C.Type
FROM    P,C
WHERE   D and C.Color='green'
```

(a)

JOIN(P, JF(C,P,D,*), D, *)

(b)

Figure 4.1 Example Output of Q_OPT Operation

---

[5] It is possible to evaluate complicated queries solely by calls to SUBSTITUTION using the following recursive function:

SB(G) = SUBSTITUTION(G, SB(_))

However, it is not possible to do the same with DECOMP.

The execution strategy of the expression in Figure 4.1b is to jfilter (e.g., semijoin) file C with file P before it is joined with P. The restriction predicate "C.Color='green'" is captured by the query fragment Q(C,R) of file C.

A Q_OPT expression can be executed once algorithms are chosen to implement RET, JF, JOIN, and PROD. We survey implementations of these operations in Sections 5 and 6. Before doing so, we make two observations.

Observation 1. The primary reason why rule-based algebras play a central role in extensible DBMSs is the ease with which new algorithms can be added to existing catalogs and to new DBMSs. Further, these algebras provide a very concise way to explain the different permutations of algorithms that one finds in comparing different DBMSs. The class of Q_OPT implementations, for example, is approximately the cross product of the classes that implement Q_GRAPH, REDUCING_PHASE, and JOINING_PHASE algorithms.

Not all combinations of algorithms yield robust Q_OPT algorithms. By pairing nonrobust algorithms, such as CBH and HY, with a robust counterpart, G, a new robust algorithm can be formed:

$$\text{NEW_ROBUST(G)} = \begin{cases} \text{HY(SIMPLE_CLIQUE(G))} & \text{; if G is a simple clique} \\ \text{BC(TREE(G))} & \text{; if G is not a simple clique but is a tree} \\ \text{G} & \text{; give up otherwise} \end{cases}$$

In this way, the number of different nonatomic algorithms that could be synthesized is enormous. [6]

Observation 2. Understanding rule-based query optimization is currently an important research topic. By grouping algorithms into catalogs, is it possible to use rule-based optimizers for both the REDUCING_PHASE and JOINING_PHASE of a query. As rule-optimizers run faster on smaller sets of rules, performance might be enhanced by optimizing on smaller rule sets, one set at a time, rather than using a single large set [Fre87, Bat87c, Loh87].

---

[6] We have taken liberties with functional syntax in this example. More accurately, a user would provide a function STRATEGY(E1(_), E2(_), E3(_), G) which would encode the conditions for executing E1(G), E2(G), or E3(G).

5. LINK Retrieval Algorithms

JOIN and JF are the basic retrieval operations on LINKs. In this section we survey their implementations for several LINK types: links with structures (PARRAY, RLIST), links without structures (ALG), and concrete links (CONL[ ]). Other LINK types are discussed in [Hae78, Bat85, Val87]. Rewrite rules are also presented that are specific to JOIN and JF operations. These rewrites have been implicitly used in designs of distributed DBMSs and database machines [Ker79, Val85, Ger86, Seg86].

5.1 The ALG Type

ALG (short for 'algorithm') is the LINK type that implements links solely by algorithms, and not by special storage structures. Common join and semijoin algorithms are realizations of this type.

It is well-known that many join algorithms do not treat F1 and F2 identically. For this reason, researchers have distinguished F1 and F2 by the terms 'inner' and 'outer'. Let JN(Fo,Fi,J,O) be the class of algorithms that realize the join of outer file Fo with inner file Fi. Thus, JOIN has two JN implementations, where the roles of inner and outer are swapped between F1 and F2:

JOIN(F1, F2, J, O)	=>	JN(F1, F2, J, O)	; F1 is outer file	(P1)
		JN(F2, F1, J, O)	; F2 is outer file	(P2)

Three subclasses of JN algorithms are recognizable: sort-merge, hashing, and nested-loop:

JN(Fo, Fi, J, O)	=>	SORT_MERGE_JN(Fo, Fi, J, O)	; sort merge joins	(J1)
		HASHING_JN(Fo, Fi, J, O)	; hashing joins	(J2)
		NESTED_LOOP_JN(Fo, Fi, J, O)	; nested loop joins	(J3)

JN algorithms have two requirements. First, their J parameter must be a predicate (not a link). Second, both Fo and Fi must be stream files. If they are not, they must be converted. The conversions are achieved, in part, by the following function. Let STR(F,s) be the stream of F records in s order:

$$\text{STR}(F,s) = \begin{cases} \text{RET}(F, Q(F,R), s) & \text{; if F is stored} \quad \text{(a)} \\ \text{SORT}(F, s) & \text{; if F is stream} \quad \text{(b)} \end{cases}$$

F is stored in case (a); records that satisfy Q(F,R) are retrieved from F in s order. F is a stream file in case (b); F is sorted in s order.

Let So(s)=STR(Fo,s) be the stream of Fo records in s order. Converting Fi to a stream is more complicated as the inner file of nested loop algorithms requires an extra parameter. Let Si(j, s) be the stream of Fi records in s order that have join value j. Parameter j can be assigned a data value or the wild card (*). In the latter case, Si(*, s) denotes the stream of all Fi records in s order. Let P(j, J) be the predicate which specifies that the join value of a record must equal j. Si(j, s) is given by:

$$\text{Si}(j,s) = \begin{cases} \text{STR}(Fi, s) & \text{; if } j{=}* \quad \text{(c)} \\ \text{FILTER}(\text{ STR}(Fi, s), P(j, J)\ ) & \text{; if } j{\neq}* \quad \text{(d)} \end{cases}$$

In case (d), stream Fi is filtered of all records that do not satisfy predicate P(j, J). By implication, Fi is recomputed for each value of j. Although this is among the simplest algorithms for (d), it certainly is not the most efficient. A more efficient algorithm might be to store Fi in a temporary file (thus computing Fi only once), and then retrieving temporary file records in s order for each value of j. Variants such as this are derivable from (d) using (A1), (A2), (R1), and (R2) rewrites.

In the following, sort-merge, hashing, and nested loop join algorithms are surveyed. We will assume J is a predicate and will use JFLD(J) to denote the join field of predicate J. Lastly, we'll catalog classes of JF

algorithms for the ALG type.

5.1.1 Sort-Merge Join Algorithms

Let MERGE_SCAN(Go, Gi, J) be the class of algorithms that perform merge-scan joins on streams Go and Gi using equijoin predicate J [Bla77]. Gi and Go must be in join field JFLD(J) order. The resulting stream of records is produced in JFLD(J) order.

SORT_MERGE_JN(Fo, Fi, J, O) => SORT(MERGE_SCAN(So(JFLD(J)), Si(*, JFLD(J)), J), O) (J1.1)

It is worth noting that when there are groups of records in Go and Gi that have the same join value, existing sort-merge joins require the rereading of Gi records [Loh87a]. In our model, Go and Gi cannot be reread. (To allow rereading would cause enormous complications in the definitions and implementations of every function in our algebra). Instead of rereading, functions (e.g., MERGE_SCAN) internally buffer records that might need to be reread, thereby keeping their external mechanics simple.

5.1.2 Hashing Join Algorithms

Let HASH_JN(Go, Gi, J) be the class of algorithms that perform hash joins on streams Go and Gi using equijoin predicate J. Go and Gi need not be in any specific order. The resulting stream of records is produced in (essentially) a random order.

HASHING_JN(Fo, Fi, J, O) => SORT(HASH_JN(So(*), Si(*, *), J), O) (J2.1)

HASH_JN(Go, Gi, J) =>	GRACE_HJ(Go, Gi, J)	; Kitsuregawa [Sha86, Kit83]	(HJ1)
	SIMPLE_HJ(Go, Gi, J)	; Shapiro [Sha86]	(HJ2)
	HYBRID_HJ(Go, Gi, J)	; Shapiro [Sha86]	(HJ3)
	CLASSICAL_HJ(Go, Gi, J)	; Shapiro [Sha86]	(HJ4)
	FRAG_HJ(Go, Gi, J)	; Sacco [Sac86]	(HJ5)

CLASSICAL_HJ, as described by Shapiro [Sha86], is not robust. It can only be used if Go can fit into main memory. See [Sha86, Ger86] for analyses and performance comparisons of several hash-join algorithms.

5.1.3 Nested Loop Join Algorithms

Let NESTED_LOOPS(Go, Gi(_), J) be the class of algorithms that perform a nested-loops join on streams Go and Gi(_) using equijoin predicate J. (The underscore _ denotes a formal parameter of Gi; Gi(x) is the stream of inner records that have x as their join value). The basic algorithm is to adjoin each record r in Go with each record in Gi(x), where x is the join value of record r. The resulting stream of records are produced in ORDER(Go).

NESTED_LOOP_JN(Fo, Fi, J, O) => SORT(NESTED_LOOPS(So(*), Si(_,*), J), O) (J3.1)

NESTED_LOOPS(Go, Gi(_), J) =>

BASIC_NL(Go, Gi(_), J)	; basic nested loops [Bla77]	(L1)
BLOCK_NL(Go, Gi(_), J)	; block nested loops	(L2)
HASH_BLOCK_NL(Go, Gi(_), J)	; hash-block nested loops [Ger86]	(L3)

BLOCK_NL differs from HASH_BLOCK_NL in that buffered records of Go are stored in a hash-based data structure rather than a heap. The change in data structure reduces main-memory search times.

5.1.4 JF Algorithms

JF(Fo, Fi, J, O) represents the class of algorithms that eliminate Fo records that do not participate in a join with Fi. There are two subclasses, semijoins and Bloom semijoins:

JF(Fo, Fi, J, O)	=>	SEMIJOIN(Fo, Fi, J, O)	; semijoin filtering	(J6)
		BLOOM(Fo, Fi, J, O)	; Bloom semijoin filtering	(J7)

Semijoins are the most widely-known JF implementations. They are implemented as special cases of standard join algorithms; i.e., there are semijoin algorithms based on sort-merge, hashing, and nested-loops. As these algorithms are fundamentally no different than their JN counterparts, we do not consider them further.

Bloom semijoins appear to be a practical alternative to semijoins. Rather than transmitting join values of the inner relation to the outer, a bit map is transmitted instead [Blo70]. The map is initialized by hashing each join value in Fi to one (or several) bits. During the filtering process, a record of Fo can be discarded if the corresponding bit(s) for its join value are not set. By its nature, a Bloom semijoin does not eliminate every record of Fo that does not participate in a join with Fi as would a semijoin. However, the number of 'false drop' records can be statistically controlled. Furthermore, as Bloom semijoins are easy to implement, they are being used in database machines [DeW86, Ger86] and are being considering for use in distributed DBMSs [Mac86a-b].

5.2 The Linkset Types

PARRAY (pointer arrays) and RLIST (ring lists) are among a large class of LINK types that rely on special storage structures called **linksets**. In order for JOIN(F1,F2,J,O) and JF(F1,F2,J,O) operations on linksets to make sense, the J parameter must always be a link name, and F1 or F2 or both must be stored files. (This contrasts with join algorithms where J must be a predicate and F1 and F2 are streams. The reason why F1 or F2 must be stored is to take advantage of linkset structures).

Let FOLLOW(S, J, A(_)) be the operation that takes each record r in stream S and concatenates it with each record that is connected to it via link J. The operation A(p) is used to access a record given its pointer p. FOLLOW has a catalog entry for each linkset type L [Bat82, Teo82]:

FOLLOW(S, J, A(_))	=>	PA_FOLLOW(S, J, A(_))	; if L is PARRAY
		RL_FOLLOW(S, J, A(_))	; if L is RLIST
		. . .	

Let ACC(F,Q,S) be the FILE operation which takes a stream of pointers S and follows each pointer to a record in file F. If the record satisfies predicate Q, it is output. The implementation of JOIN for linkset types is captured by following pointers from F1 to F2 and vice versa:

JOIN(F1, F2, J, O)	=>	SORT(FOLLOW(STR(F1), J, ACC(F2, Q(F2,R), _)), O)	; if F2 is stored
		SORT(FOLLOW(STR(F2), J, ACC(F1, Q(F1,R), _)), O)	; if F1 is stored

Linkset implementations of the JF operation are no different than that of JOIN operations.

5.3 The CONL[ ] Type

There are several join algorithms that exploit storage structures which are introduced by FILE types. The join indices algorithm of Blasgen and Eswaren is an example [Bla77]. Relations (i.e., abstract files) are implemented by inverted files. Traversing a link (i.e., join) connecting two relations is realized by a merge-scan or nested loop join of secondary index files over the join keys of both relations. (A more detailed explanation of the algorithm is given in Section 6.2).

Rather than defining a special LINK type for each FILE type to account for these algorithms, we use a single LINK type, called CONL[n:LINK] (for concrete link). CONL[ ] states that the implementation of an abstract link exploits the FILE type of its connecting abstract files. Thus, if a DBMS supporting MODEL M implements conceptual files by inverted files and conceptual links by the above join indices algorithm, its type expression would be:

M[INDEX[df,xf,k], CONL[ALG]]

Specific values for df, xf, and k are not needed for this example.

The CONL type maps JOIN and JF operations directly to CJOIN and CJF (concrete JOIN and concrete JF) operations, whose implementations are FILE type dependent:

JOIN(F1, F2, J, O) => CJOIN(F1, F2, J, O) ; concrete (lower layer) join algorithms

JF(F1, F2, J, O) => CJF(F1, F2, J, O) ; concrete (lower layer) jfilter algorithms

Catalogs of CJOIN operations are presented when we examine the operation mappings of FILE types (Section 6). Catalogs of CJF operations are no different than those for CJOINs, and are not considered further.

5.4 JOIN and JF Rewrite Rules

Let G(Fo, Fi, J, O) be a JOIN or JF operation. Three rewrite rules specific to JOIN and JF are:

G(Fo, Fi, J, O) <-> G(JF(Fo, Fi, J, O1), Fi, J, O) (A4)

G(Fo, JF(Fi, Fo, J, O2), J, O) (A5)

G(JF(Fo, Fi, J, O1), JF(Fi, Fo, J, O2), J, O) (A6)

(A4) means replace parameter Fo in a JOIN or JF operation with JF(Fo, Fi, J, O1). Similarly, (A5) replaces parameter Fi with JF(Fi, Fo, J, O2). (A6) completes the triad by replacing both Fo and Fi. Note that O1 and O2 are unspecified orders in (A4)-(A6); orders other than random (*) can be assigned for purposes of optimization.

The utility of these rewrites is to eliminate records that do not participate in the join or jfilter of Fo and Fi. Generally used with JOIN operations [Ker79, DeW86, Val85, Seg86], the rewrites occasionally find use with JF operations. As an example, the **remote semijoin** of Segev [Seg86] is an (A5) rewrite of the SEMIJOIN implementation of JF:

SEMIJOIN(Fo, Fi, J, O) <-> SEMIJOIN(Fo, SEMIJOIN(Fi, Fo, J, O), J, O)

That is, instead of performing a semijoin of Fo on the join values of Fi directly, an alternative is to reduce the number of join values of Fi by semijoining Fi with Fo first. In the context of distributed DBMSs (the subject of Section 7), Segev has found situations where remote semijoins are more efficient than conventional semijoins.

6. FILE Retrieval Algorithms

We examine five representative FILE types: heap and B+ trees (HEAP, BPLUS), indexing (INDEX[ ]), augmenting record type identifiers (AUG[ ]), and horizontal partitioning (HPART[ ]). The first three are discussed in this section; the remaining are in Appendix 3.

We will consider the mappings of two file retrieval operations: RET and ACC. (Recall that ACC is the operation which accesses a record given its pointer). We will also examine the mappings of the CJOIN operation, an operation that arises when the implementation of links are forced to lower layers (see Section 5.3).

In our discussions, we will use AF to denote an abstract file and F as its dominant concrete file, i.e., the primary file to which AF is mapped [Bat85].

6.1 The BPLUS and HEAP Types

BPLUS and HEAP are among a large subclass of FILE types that map abstract files to simple file structures. Each structure/type has its own retrieval and access algorithms:

RET(AF, Q, O) => SORT(BPLUS_RET(F, Q), O) ; BPLUS catalogs (I1)

ACC(AF, Q, S) => BPLUS_ACC(F, Q, S) (I2)

RET(AF, Q, O) => SORT(HEAP_RET(F, Q), O) ; HEAP catalogs (I3)

ACC(AF, Q, S) => HEAP_ACC(F, Q, S) (I4)

Equations (I1) and (I3) capture the property that records retrieved from file structures are returned in the order in which they are stored. (B+ trees store records in primary key order; heaps store records in random order). If a different ordering is needed, a sort is performed. Algorithms for these and other file structures are given in [Teo82].

CJOIN operations map directly to JOIN operations:

CJOIN(AFo, AFi, J, O) => JOIN(Fo, Fi, J, O) ; JOIN has no CONL implementation (I5)

File structures reside at the lowest level of a DBMS. Therefore, JOIN operations at this level must be realized by sort-merge, hashing, nested-loops, or linkset algorithms, and cannot be pushed to lower layers (i.e., the CONL link implementation cannot be used).

6.2 The INDEX[ ] Type

INDEX[df:FILE, xf:FILE, k:LINK] maps an abstract file AF to an inverted file, consisting of a data file F and n index files $I_1 \cdots I_n$. Let L_j be the link that connects I_j to F. The implementation of the data file, index files, and links are the parameters df, xf, and k.

Pointers to abstract records are indistinguishable from pointers to F records. Thus, accessing AF records maps to accessing F records:

ACC(AF, Q, S) => ACC(F, Q, S) (X1)

There are many inverted file retrieval algorithms; three of which are classical. The first scans the data file. It always works, but is slow. The second, popularized by System R [Ast76], uses exactly one index file. The file selected could be a restriction index or an index on the sort column. (The former is used to eliminate records, while the latter eliminates a sort. Part of the algorithm is determining which of several index files to use). The third uses many index files and processes queries by taking the union and intersection of inverted lists.

To represent the third algorithm, let LISTPROC(Q, $\xi_{j \in XQ}$ G_j) be the function that takes a collection of

inverted lists and forms their union and intersection. Q is the query, XQ is the set of subscripts that identify the index files to be accessed, and G_j is the function that retrieves the index record(s) from index file I_j. Let $\xi_{j \in XQ} G_j$ denote a list of G functions, and let IQ_j be the predicate that selects index records from I_j. We have:

RET(AF, Q, O) => RET(F, Q, O) ; scan data file (X2)

JOIN(F, I_t, L_t, O) ; use one index file (X3)

; t is selected by query optimizer

JOIN(F, LISTPROC(Q, $\xi_{t \in XQ}$ RET(I_t, IQ_t, *)), L_j, O) (X4)

; use several index files

; XQ and j are selected by query optimizer

In (X3) and (X4), indices are used to process queries. The selected indices are encoded in the values assigned to the t subscripts, which are determined by a query optimizer. We outline how query optimizers work in the context of our algebra in Section 8. Note that the implementation of the JOIN operations in (X3) and (X4) are specified by parameter k of INDEX[ ].

There are many algorithms for joining two inverted files (i.e., CJOIN mappings), most of which can be mechanically generated from two simple rewrites. Both rewrites involve the use of join indices, i.e., index files on the join column. One rule states that a join of two files Fo and Fi can be replaced by two joins: the first joins Fo with the join index file of Fi, and the second joins the latter result with Fi. The second rewrite rule does the same with the roles of Fo and Fi reversed.

Let $Io_1, Io_2, \cdots$ be the index files of Fo and let $Ii_1, Ii_2, \cdots$ be the index files of Fi. Let $\hat{o}$ and $\hat{i}$ be the subscripts of the join index files of Fo and Fi (if they exist). The rewrite rules described above are:

JOIN(Fo, Fi, J, O) <-> JOIN(JOIN(Fo, $Ii_{\hat{i}}$, J, Oi), Fi, $Li_{\hat{i}}$, O) (R3)

JOIN(JOIN($Io_{\hat{o}}$, Fi, J, Oo), Fo, $Lo_{\hat{o}}$, O) (R4)

Note that the applicability of either rule depends on the existence of the specified join index.

Different algorithms arise for CJOIN depending on the stored or stream nature of AFo and AFi. There is one algorithm for each of the four possibilities:

CJOIN(AFo, AFi, AJ, O) => JOIN(AFo, AFi, J, O) ; AFo and AFi are stream (X5)

JOIN(AFo, Fi, J, O) ; AFo is stream, AFi is stored (X6)

JOIN(Fo, AFi, J, O) ; AFo is stored, AFi is stream (X7)

JOIN(Fo, Fi, J, O) ; AFo and AFi are stored (X8)

Recall the join indices algorithm of Blasgen and Eswaren [Bla77]. This algorithm realizes the join of two inverted files by joining the join indices of both files, following pointers to both data files, and applying restriction predicates. The join indices algorithm (X8') is a variant of (X8), and can be generated by applying (R3) and (R4) to (X8). (The order in which rules are applied is one of the variations of the algorithm; the one given below uses the order (R3) followed by (R4)).

CJOIN(AFo, AFi, AJ, O) => JOIN(JOIN(JOIN($Io_{\hat{o}}$, $Ii_{\hat{i}}$, J, Oo), Fo, $Lo_{\hat{o}}$, Oi), Fi, $L_{\hat{i}}$, O) (X8')

Other CJOIN algorithms for the INDEX type are given by Rosenthal and Reiner [Ros82].

7. Distributed and Parallel Computation

A program need not be executed on a single processor. Its computations can be distributed over several processors whose locations may be at remote sites. Functional expressions are ideal for expressing distributed computations. Each function F of an expression E can be executed at a different processor. Its input is from the processor that executed the function which immediately preceeds F in E, and its output is directed to the processor that executes the function immediately following F in E.

We use superscripts F^f to designate that function F is executed by processor f. Thus, the expression $A^a(B^b(C^b))$ states that functions B and C are executed by processor b and function A is executed by processor a. As the physical locations and interconnections between processors are not specified, expressions can denote distributed computations in a wide area network, a local area network, a loosely coupled system, or a tightly coupled system.

Most of the expressions we have encountered so far can be realized by a linear pipeline of processors. Distributing computations in this manner does not necessarily increase parallelism or efficiency. There appears to be a small number of rewrite rules whose sole purpose is to increase the parallelism of a computation, and to increase computation efficiency as a by-product. In the following sections, we identify three rules which have been used in the design of database machines. We begin with a brief discussion of stream multiplexors and demultiplexors.

7.1 Stream Multiplexors and Demultiplexors

SPLIT is the class of algorithms that partition a stream A into n substreams $X_1 \cdots X_n$. ASSEMBLE is the class of algorithms that do the inverse; they take n substreams and produce a single, ordered stream.

The mechanisms for splitting streams are fundamentally no different than those used for horizontal partitioning. Load balancing (or round robin), hashing, and range key splitting are the common methods [DeW86, Ger86, Kit83, Fus86]. Let $\xi_{i=1}^{n} X_i$ denote the list of streams $X_1 \cdots X_n$ and let the bar | delimit parameters that are lists. SPLIT has at least the following implementations:

$$\text{SPLIT}(A \mid \xi_{j=1}^{n} X_j) \Rightarrow \text{LSPLIT}(A \mid \xi_{j=1}^{n} X_j) \quad \text{; load balancing} \qquad \text{(M1)}$$

$$\text{HSPLIT}(A \mid \xi_{j=1}^{n} X_j) \quad \text{; hashing} \qquad \text{(M2)}$$

$$\text{RSPLIT}(A \mid \xi_{j=1}^{n} X_j) \quad \text{; range split} \qquad \text{(M3)}$$

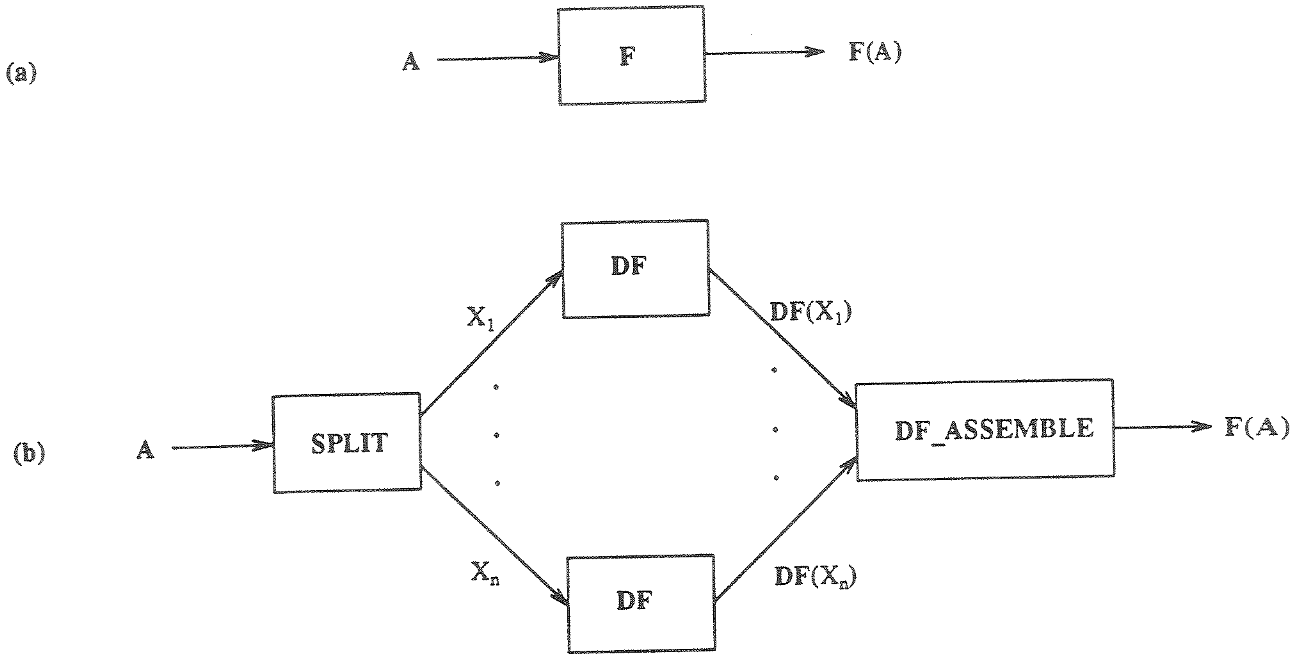
There are two general ways to ASSEMBLE streams. One merges streams $X_1 \cdots X_n$ which are already in O order into a single stream that is in O order. The second combines streams $X_1 \cdots X_n$ in any order, followed by a sort.

$$\text{ASSEMBLE}(\xi_{j=1}^{n} X_j \mid O) \Rightarrow \text{MERGE}(\xi_{j=1}^{n} X_j \mid O) \quad \text{; merge} \qquad \text{(M4)}$$

$$\text{SORT}(\text{COMBINE}(\xi_{j=1}^{n} X_j), O) \quad \text{; combine} \qquad \text{(M5)}$$

7.2 Distribution Rewrite Rule

Figure 7.1a shows a very common situation: a stream A is consumed by function F to produce stream F(A). Figure 7.1b shows how this computation can be distributed: A is split into n substreams $X_1 \cdots X_n$, function DF ("Distributed F") maps each substream, and the results are assembled by function DF_ASSEMBLE to produce F(A). We call this the **distributed rewrite (DR)** of F(A):



(c)

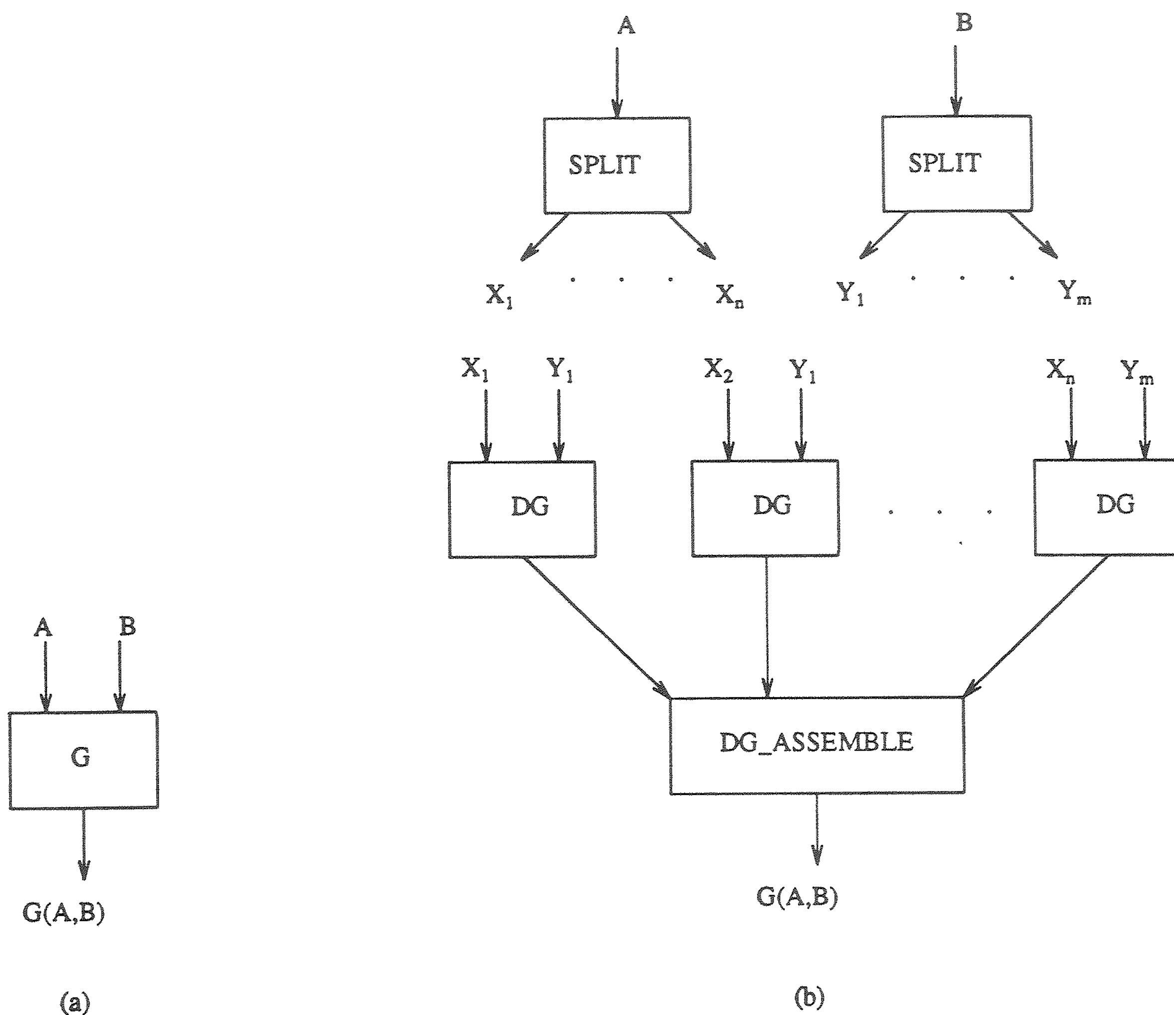
F	DF	DF_ASSEMBLE
SORT	SORT	MERGE
COUNT	COUNT	SUM
AVE	SC	SC_ASSEMBLE

Figure 7.1 The Distributed Rewrite Rule

$$F(A) \quad \text{<->} \quad SPLIT(\, A \mid \xi_{j=1}^{n} X_j \,);$$
$$DF_ASSEMBLE(\, \xi_{j=1}^{n} DF(X_j) \mid O \,) \qquad ; O = ORDER(F(A)) \qquad \text{(DR1)}$$

DF and DF_ASSEMBLE are functions that satisfy the DR for F. A table of DR-related functions given in Figure 7.1c. As an example, if F is AVE (average), the DF function is SC which reduces a stream of numbers to an ordered pair (sum of numbers in stream, count of numbers in stream). SC_ASSEMBLE combines ordered pairs to produce the average of the original stream. A more general definition of DR1 is given by Ceri and Pelagatti [Cer84].

(DR1) is applicable to functions that operate on a single input stream. (DR2) is a generalization which handles functions, such as JOIN and MERGE, that operate on multiple streams. Figure 7.2a shows a function G that processes streams A and B. Figure 7.2b shows how G can be distributed. A is split into n substreams $X_1 \cdots X_n$ and B is split into m substreams $Y_1 \cdots Y_m$. Each pair of substreams (one from A and one from B) is processed by DG ("Distributed G"). DG_ASSEMBLE combines n*m substreams to yield G(A,B). We have:



(c)

G	DG	DG_ASSEMBLE
JOIN	JOIN	MERGE
MERGE	MERGE	MERGE

Figure 7.2 Generalized Distributed Rewrite Rule

$$G(A,B) \quad <-> \quad SPLIT(\ A \mid \xi_{j=1}^{n} X_j\ );$$

$$SPLIT(\ B \mid \xi_{i=1}^{m} Y_i\ );$$

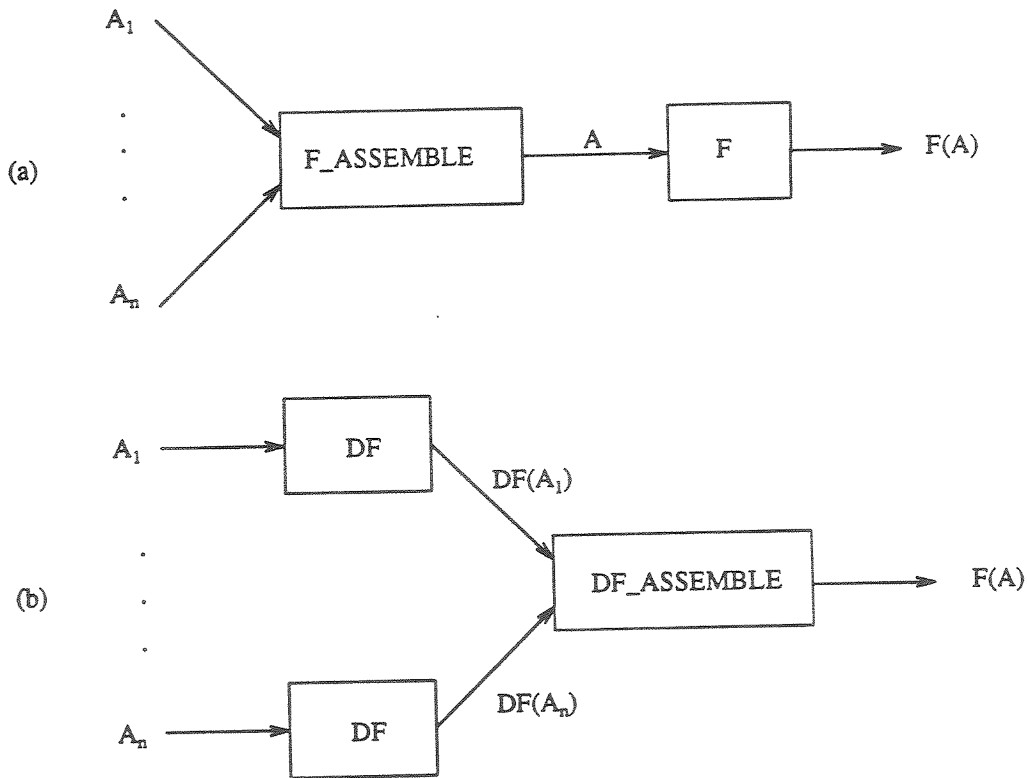
$$DG_ASSEMBLE(\ \xi_{j=1}^{n} \xi_{i=1}^{m} DG(\ X_j, Y_i\ ) \mid O\ ) \qquad ; O = ORDER(G(A,B)) \qquad \textbf{(DR2)}$$

DG and DG_ASSEMBLE are functions that satisfy the DR for G. Figure 7.2c is a table of DR-related functions that consume two or more streams.

It is possible to simplify (DR2) significantly by choosing the appropriate SPLIT functions. Consider the JOIN operation. In the GAMMA and GRACE database machines, both streams A and B are hash split on the join field using the same hashing function. Taking the join of X_i with Y_j is guarenteed to produce the null stream if $i \neq j$. Instead of evaluating n^2 joins, only n joins are needed. This simplification can be verified using file characteristics.

7.3 Assembly Rewrite Rule

Figure 7.3a shows a common situation involving assemble functions: a function F processes a single stream A which is an assembly of substreams $A_1 \cdots A_n$. Figure 7.3b shows how this computation can be distributed: function DF is applied to each substream, and the results are assembled by DF_ASSEMBLE to produce F(A). We call this the **assembly rewrite (AR).**



(c)

F	F_ASSEMBLE	DF	DF_ASSEMBLE
SORT	MERGE	SORT	MERGE
SPLIT	ASSEMBLE	SPLIT	ASSEMBLE

Figure 7.3 Assembly Rewrite Rule

$$F(\text{F_ASSEMBLE}(\xi_{j=1}^{n} A_j \mid O)) \quad <-> \qquad \text{(AR1)}$$

$$\text{DF_ASSEMBLE}(\xi_{j=1}^{n} \text{DF}(A_j) \mid O1) \qquad ; O1 = \text{ORDER}(F(A))$$

DF and DF_ASSEMBLE are functions that satisfy the AR for the function pair (F, F_ASSEMBLE). A table of AR-related functions is given in Figure 7.3c.

An interesting example of (AR1), which is used in the GRACE and GAMMA database machines, involves the (SPLIT, ASSEMBLE) pair. Figure 7.4a shows how substreams $A_1 \cdots A_n$ are assembled into a single stream only to be split immediately into m substreams $X_1 \cdots X_m$. The (AR1) rewrite transforms Figure 7.4a into 6.4b, where each of the A_j substreams is split immediately into m substreams $Y_{j,1} \cdots Y_{j,m}$, and the assembly of all substreams $Y_{1,k} \cdots Y_{n,k}$ for a fixed k yields X_k. This rewrite is:

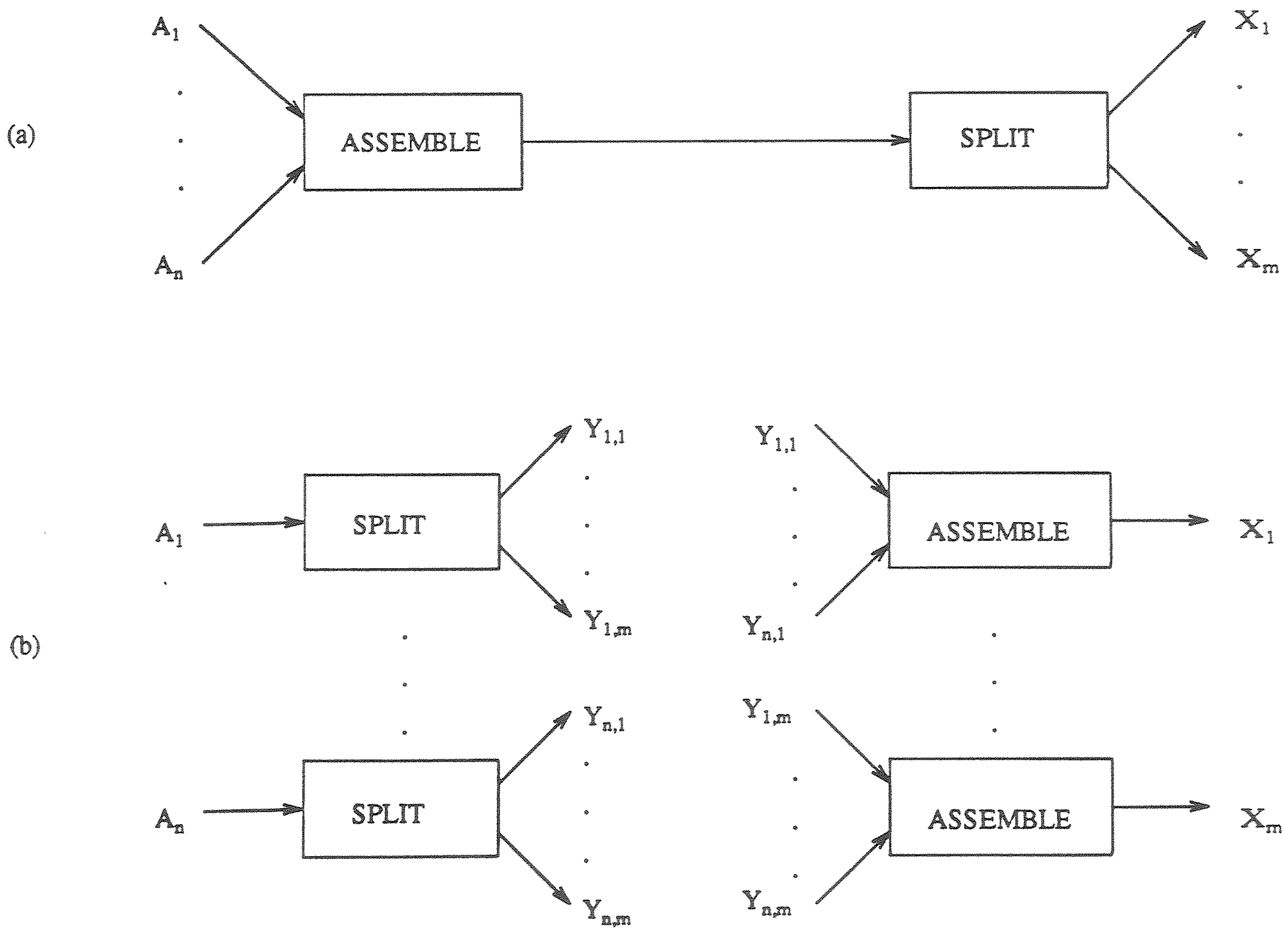


Figure 7.4 The SPLIT-ASSEMBLE Pair

$$\text{SPLIT}(\ \text{ASSEMBLE}(\ \xi_{j=1}^{n}\ A_j \mid O\ ) \mid \xi_{k=1}^{m}\ X_k\ ) \quad \text{<->}$$

$$\xi_{j=1}^{n}\ \text{SPLIT}(\ A_j \mid \xi_{k=1}^{m}\ Y_{j,k}\ );$$

$$\xi_{k=1}^{m}\ \text{NAME}(\ \text{ASSEMBLE}(\ \xi_{j=1}^{n}\ Y_{j,k} \mid O\ ), X_k\ ) \qquad \text{(AR1.1)}$$

Note that the use of NAME(A, N) in (AR1.1) is merely a syntactic convenience. It simply gives stream A the label N and performs no actual computation.

Figure 7.5 shows a generalization of (AR1) that applies to functions that operate on two or more input streams:

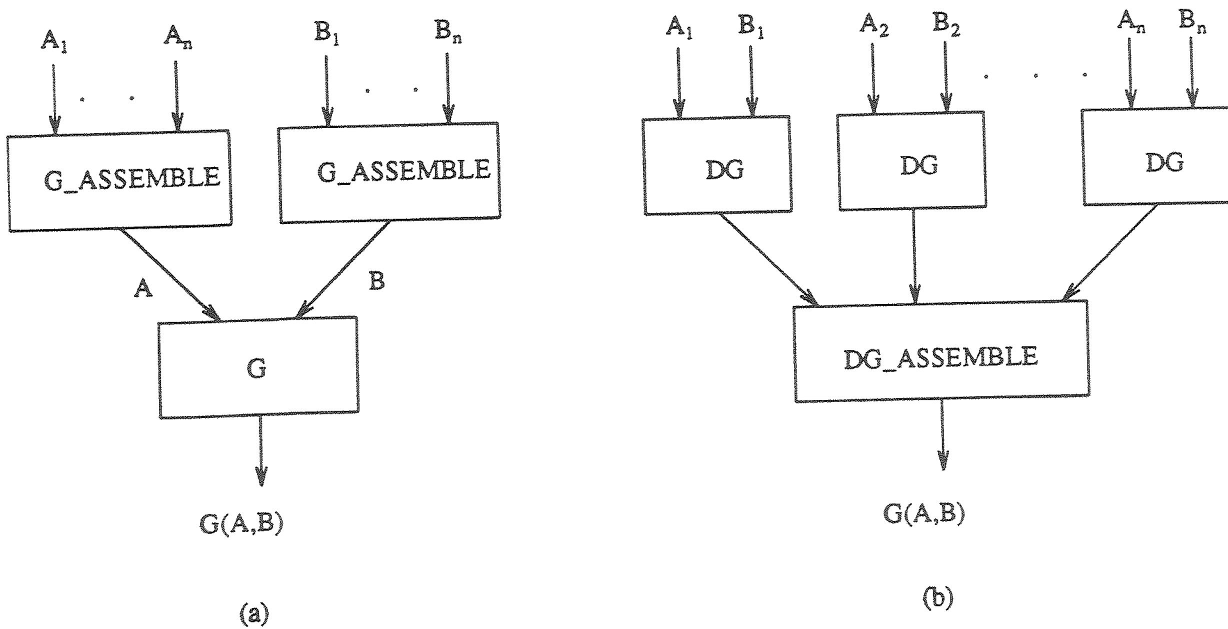


Figure 7.5 Generalized Assembly Rewrite

$$\text{G}(\ \text{ASSEMBLE}(\ \xi_{j=1}^{n}\ A_j \mid O\ ), \text{ASSEMBLE}(\ \xi_{k=1}^{m}\ B_k \mid O\ )) \quad \text{<->}$$

$$\text{DG_ASSEMBLE}(\ \xi_{j=1}^{n}\ \xi_{k=1}^{m}\ \text{DG}(\ A_j, B_k\ ) \mid O1\ ) \quad ; O1 = \text{ORDER}(\text{G}(A,B)) \qquad \text{(AR2)}$$

DG and DG_ASSEMBLE are functions that satisfy AR for the pair (G, G_ASSEMBLE). As an example, if the (G, G_ASSEMBLE) pair is (JOIN, MERGE), the (DG, DG_ASSEMBLE) functions are (JOIN, MERGE). That is, merging the fragments of relations and then joining is the same as joining all pairs of fragments and merging their results.

7.4 Split Rewrite Rule

SPLIT splits a stream A into substreams $X_1 \cdots X_n$, where all records of A are assigned to exactly one substream. Suppose record r is assigned to X_i if r satisfies predicate P_i, where P_i is the **split predicate** for X_i.[7] Consider Figure 7.6a. Let F(Q) be a function which produces A, where all records in A satisfy predicate Q. Figure 7.6b shows how query modification can be used to distribute this computation. Instead of evaluating F(Q) once, n different computation instances are spawned, where the ith computation evaluates F(Q and P_i), which produces X_i:

$$\text{SPLIT}(\, F(Q) \mid \xi_{i=1}^{n} X_i \,) \quad \text{<-->} \quad \xi_{i=1}^{n} \text{NAME}(\, F(Q \text{ and } P_i\,), X_i\,) \qquad \text{(SR)}$$

We call this the **split rewrite (SR)**. The (SR) is used in the SABRE database machine [Che86].

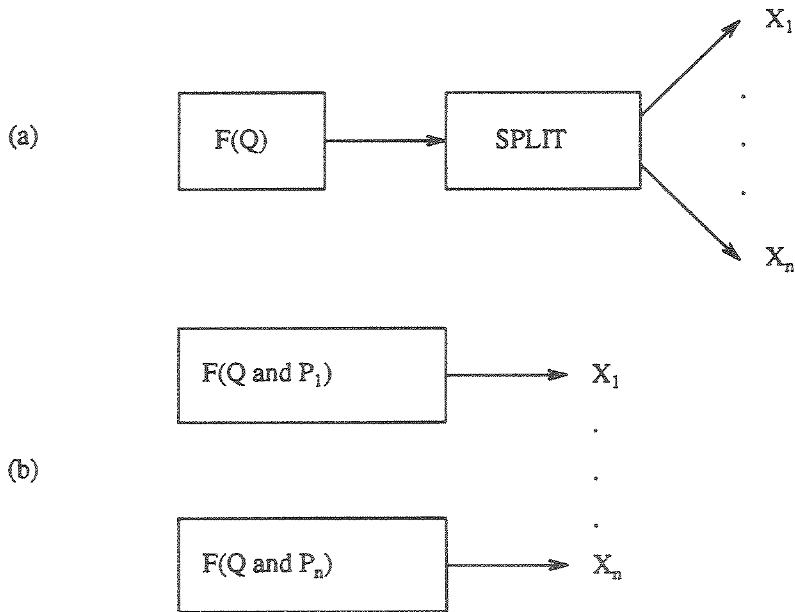


Figure 7.6 Split Rewrite Rule

[7] In the case that LSPLIT (load-balancing) is used, P_i=true for all i.

8. Examples: a Centralized DBMS, a Distributed DBMS, and a Database Machine

The catalogs and rewrite rules that we have presented can be used to express the design and algorithms of prototype as well as operational DBMSs. In this section, we show how the query processing algorithms of three rather different DBMSs can be built from a common pool of components: the MODEL types SQL[ ] and NETWORK[ ], the FILE types AUG[ ], HPART[ ], HEAP, and BPLUS, and the link types ALG, PARRAY, and CONL[ ]. We will use the notation O_X to denote the implementation of operation O in system X.

8.1 System R: a Centralized DBMS

System R (now SQL/DS) is a relational DBMS that was developed at IBM San Jose in the late 1970s [Ast76]. It is a centralized and software-based system that runs on a single processor. Its query processing algorithm, by Selinger et al. [Sel79-80], is universally recognized as a seminal research contribution.

System R consists of two subsystems: the **Relational Data System (RDS)** and the **Relational Storage System (RSS)**. The RDS presents the SQL front-end of System R; it maps relations to RSS, a sophisticated file management system. The boundaries of the RDS and RSS do not match precisely with the functional interfaces that our algebra requires. However, an approximate match has RDS providing the query processing functions at the conceptual level with RSS providing the functions of all lower levels. RDS has the type expression SQL[RSS_TYPE, ALG], where RSS_TYPE will be defined shortly.

RDS is responsible for transforming SQL queries into executable expressions. Query optimization is done exclusively in the joining phase by the SYS_R algorithm; no optimization occurs in the reducing phase (i.e., REDUCING_PHASE(G)=G). Thus, in RDS:

$$\text{Q_OPT}_{RDS}(\text{R}) \quad \Rightarrow \quad \text{SYS_R}(\text{SQL_GRAPH}(\text{R})) \qquad \text{(E0)}$$

A characteristic of the SYS_R algorithm is that it generates joins where the outer file is always a stream and the inner file is always stored. Joins are realized in RDS by either sort-merge or basic nested loop algorithms, which are instances of (J1) and (J3):

$$\text{JOIN}_{RDS}(\text{F1, F2, J, O}) \quad \Rightarrow \quad \text{SORT(MERGE_SCAN(SORT(F1, JFLD(J)), RET(F2, Q2, JFLD(J)), J), O)} \qquad \text{(E1)}$$

$$\text{SORT(BASIC_NL(F1, RET(F2, Q2 and P(_,J), *), J), O)} \qquad \text{(E2)}$$

When a JOIN_{RDS} is processed, the cheaper of (E1) and (E2) is executed. This decision is made either at run-time for ad-hoc queries, or compile time for repetitive queries [Cha81]. Note that the only operation in (E1) and (E2) that requires the support of RSS is RET.

RSS maps relations to inverted files. Data file records are augmented with their relation identifiers, and are then stored in heaps. Index files are stored in B+ trees, and index-to-data file links are realized by pointer arrays. This implementation is defined by the type expression RSS_TYPE = INDEX[AUG[HEAP], BPLUS, PARRAY].

To understand file retrievals in RSS, first consider the INDEX[ ] algorithms. Index files in RSS are assigned different labels: restriction, ordering, and clustering. The restriction and ordering labels are query dependent, while clustering is not. A clustering index provides a fast access path to all data records and is an alternative to data file scans. Typically, a clustering index is also an index on a primary key.

Four different retrieval algorithms are supported in the indexing layer: **dbscan**, which scans the data file, **clustered index scan**, which uses the cluster index to scan the data file, **sorted scan**, which uses a restriction index to process a query followed by a sort, and **unclustered index scan**, which is used when an ordered result is needed and an index on the ordering column is present. dbscan is identical to (X2), while the others are instances of (X3). Pointer array algorithms are used to perform index file - data file joins. In the algorithms below, F denotes the data file, I_j an index file, $\hat{r}$ denotes the subscript of a restriction index, $\hat{o}$ the ordering index, and $\hat{c}$ the clustering index:

$\underset{RSS}{RET}(AF, Q, O)$ =>

$RET(F, Q, O)$	; dbscan	(E3')
$SORT(\ PA_FOLLOW(\ RET(I_{\hat{c}}, QI_{\hat{c}}, *), L_{\hat{c}}, ACC(F, Q, _)\ ), O)$	; clustered index scan	(E4')
$SORT(\ PA_FOLLOW(\ RET(I_{\hat{s}}, QI_{\hat{s}}, *), L_{\hat{s}}, ACC(F, Q, _)\ ), O)$	; sorted scan	(E5')
$PA_FOLLOW(\ RET(I_{\hat{o}}, QI_{\hat{o}}, *), L_{\hat{o}}, ACC(F, Q, _)\ )$	; ordering index scan	(E6')

It is important to note that the output of dbscan and clustered index scan algorithms in RSS is not sorted, unlike sorted scan and ordering index scan. In contrast, our algebra requires sorts to be present in (E3') and (E4') to conform with the higher-level specifications of $\underset{RSS}{RET}$. In this sense, our descriptions of $\underset{RSS}{RET}$ are approximate. [8]

Immediately below the indexing layer is the augment RTI layer, which augments RTIs (relation identifier) to data records. (G1) and (G2) from Appendix 3 define the mapping of RET and ACC operations.

At the internal level, data files are stored in heap file structures called **segments**. Segments can contain the records of a single data file or multiple data files. (Assigning data files to segments is the responsibility of the DBA). Index files are implemented as B+ trees. (I1)-(I4) are the mappings of the retrieval and access algorithms for B+ trees and heaps. Let H be the heap file (segment) in which the records of data file F are stored, and let IH_j be the internal file to which index file I_j is mapped. Substituting augment RTI and internal algorithms into (E3')-(E6') and simplifying yields the RSS retrieval algorithms:

$\underset{RSS}{RET}(AF, Q, O)$ =>

$SORT(\ HEAP_RET(H, Q \text{ and } RTIQ(AF)), O)$	; dbscan	(E3)
$SORT(\ PA_FOLLOW(\ BPLUS_RET(HI_{\hat{c}}, QI_{\hat{c}}), L_{\hat{c}}, HEAP_ACC(H, Q, _)\ ), O)$	; clustered index scan	(E4)
$SORT(\ PA_FOLLOW(\ BPLUS_RET(HI_{\hat{s}}, QI_{\hat{s}}), L_{\hat{s}}, HEAP_ACC(H, Q, _)\ ), O)$	; sorted scan	(E5)
$PA_FOLLOW(\ BPLUS_RET(HI_{\hat{o}}, QI_{\hat{o}}), L_{\hat{o}}, HEAP_ACC(H, Q, _)\ )$	; ordering index scan	(E6)

As in the case of $\underset{RDS}{JOIN}$, the cheapest algorithm among (E3)-(E6) is executed when a $\underset{RSS}{RET}$ is processed. Composing (E1) and (E2) with (E3)-(E6) yields all (eight) algorithms that are used to accomplish the join of conceptual files/relations.

Further discussion on query optimization is given in [Loh87b, Bat87b]. Another model of System R, different from ours, has been presented by Freytag [Fre87].

8.2 R*: A Distributed DBMS

R* is an experimental distributed DBMS that is an enhancement of System R. The RDS of System R underwent modification to enable relations to be located at different sites. (We will call this modified subsystem RDS*). The RSS of System R remained unchanged. Thus, the type expressions defining R* and System R are identical. Further, algorithms (E3)-(E6) are the same for both DBMSs.

---

[8] It is this type of mismatch that is common in existing DBMSs; different algorithms with incompatible or ad-hoc specifications 1) cause query optimization to be more difficult (as the conditions to use certain algorithms are more complicated), 2) make extensibility more difficult, and 3) obscures the recognition that a building-blocks approach can be taken to construct DBMSs.

Query processing in R* is very similar to that in System R. The RSTAR algorithm is used instead of SYS_R:

$$Q_OPT_{RDS^*}(R) \Rightarrow RSTAR(SQL_GRAPH(R)) \quad (E0)$$

The main distinction between SYS_R and RSTAR is the selection at which sites (i.e., processors) different functions of an access path are to be executed. We rewrite (E1) and (E2) below using processor superscripts, and give a variant of (E2) which stores the stream of inner file records in a temporary file and joins the outer file with the temporary file. (Readers can recognize this as an (A2*A1) rewrite of (E2). The rationale for this variant and its use is given in [Loh85, Mac86a-b]). We use x to denote the join site, and o and i are the outer file and inner file sites:

$JOIN^x_{RDS^*}(Fo^o, Fi^i, J, O) \Rightarrow$

$$SORT^x(MERGE_SCAN^x(SORT^x(Fo, JFLD(J)), RET^i(Fi, Qi, JFLD(J)), J), O) \quad (E7)$$

$$SORT^x(BASIC_NL^x(Fo, RET^i(Fi, Qi \text{ and } P(_J), *), J), O) \quad (E8)$$

$$STORE_TMP^x(T, RET^i(Fi, Qi, JFLD(J)), *);$$
$$SORT^x(MERGE_SCAN^x(SORT^x(Fo, JFLD(J)), RET^x(T, null, JFLD(J)), J), O) \quad (E9)$$

Four cases are considered in RDS*. Case 1 deals with Fo and Fi at the same site. Setting i=o=x, algorithms (E7) and (E8) reduce to the System R algorithms, (E1) and (E2). Algorithm (E9) is not considered in Case 1.

Cases 2-4 assume Fo and Fi are at different sites. Case 2 ships the outer relation Fo to the inner. Setting x=i in (E8) and (E9) yields another two algorithms.. (E9) is not considered in Case 2.

Case 3 converts the inner file to a stream and ships it to the site of the outer file. Two different shipping strategies are distinguished: **fetch-inner-as-needed** and **ship-inner-in-whole-and-store**. The algorithms that use the former strategy are (E7) and (E8) with x=o. The algorithm that uses the latter strategy is (E9) with x=o.

Case 4 processes joins at a designated site that may be distinct from o or i. Called the **wild card** site, it remains unspecified until run-time. The algorithms for Case 4 are (E7)-(E9) with x='wild_card'.

There is a total of ten algorithms which RDS* considers in optimizing the join of two files, eight more than that considered in System R. Again, these extra algorithms are simple variants - mostly choosing the processor/site superscripts - of the System R algorithms.

8.3 GRACE: A Database Machine

GRACE is a parallel relational database machine being developed at the University of Tokyo. Descriptions of the join algorithm for conceptual files/relations have been featured in [Kit83, Fus86]. We show below how this algorithm is composed from atomic algorithms.

GRACE stores a relation by augmenting the relation id (RTI) to each of its tuples and by horizontally partitioning the augmented relation (presumably using a load-balancing method) over several disks. Each disk has a filtering and projection unit, so all operations on internal files are done in hardware.

A join of two conceptual files/relations is accomplished in two phases. The first is the **staging phase**. Filtered tuples from *both* relations are read in parallel from each disk. They are then hash-split on the join key and stored in temporary files, where each temporary file contains the records from all disks that have the same hash address. What is unusual about this algorithm is that each temporary file contains tuples from *both* relations; no attempt is made to store tuples of outer and inner relations in separate files. Once the temporary files have been loaded, the staging phase is complete.

Next is the **processing phase**, where a GRACE_JN is performed on each temporary file. (GRACE_JN works by separating outer records from inner records and forming their join. Further details are given in Appendix 3). GRACE_JN is executed on a single processor. However, there can be several processors

executing this algorithm on different temporary files simultaneously. As there are more temporary files than available processors, files are processed in increasing order of their size. When a processor is finished with one file, it begins processing the next available temporary file. Each invocation of GRACE_JN produces a stream of joined records. This stream is either hash-split (as in the staging phase) for subsequent joins, or it is merged with other join-streams to form the final result.

The architecture of GRACE consists of three FILE types composed in the following order: augment RTI, horizontal partitioning, and heap. There is no data model layer, and conceptual JOIN operations are mapped through the augment layer. This corresponds to the type expression NETWORK[AUG[HPART[HEAP]], CONL[ALG]]. Furthermore, conversion rewrites are used to create and read temporary files. The distributed rewrite and assembly rewrite rules spawn copies of GRACE_JN to introduce parallelism. If Fo and Fi are stream files that are to be joined on predicate J, where the resulting joined records are output in random order, the GRACE join implementation is:

$$\underset{GRACE}{JOIN}(Fo, Fi, J, *) \Rightarrow \xi_r \, HSPLIT^r(Fo_r \mid \xi_{k=1}^{n} X_{r,k}); \quad (GR1)$$

$$\xi_s \, HSPLIT^s(Fi_s \mid \xi_{k=1}^{n} Y_{r,k}); \quad (GR2)$$

$$\xi_{k=1}^{n} \, STORE_TMP^k(T_k, COMBINE^k(\xi_r X_{r,k}, \xi_s Y_{s,k}), *); \quad (GR3)$$

$$COMBINE(\xi_{k=1}^{n} GRACE_JN^{\sigma(k)}(HEAP_RET^k(T_k, null), J)) \quad (GR4)$$

A derivation and detailed explanation of the above expressions are given in Appendix 4. Very briefly, (GR1)-(GR3) correspond to the staging phase where relation fragments Fo_r on disk r and Fi_s on disk s (for all r and s) are hash split into temporary files $T_1 \cdots T_n$, one temporary file for each hash value. (GR4) corresponds to the processing phase, where processor $\sigma(k)$ is assigned to process temporary file T_k. The method of processing involves reading the temporary file and processing its contents by the GRACE_JN algorithm. σ is the function that schedules the processing of temporary files in order of their size.

8.4 More Observations

Observation 3. Software engineering, unlike any other engineering discipline, provides little in the way of tools to minimize the reinvention of technology. Research in software engineering currently stresses the development of programming environments, where editors, languages, debuggers, etc. make programming easier. Little emphasis is placed on software reusability. It is our belief that no matter how good programming environments become, significant increases in software productivity will be achieved only when well-understood technology doesn't have to be reinvented.

In this and in our earlier papers, we have shown that there is a considerable overlap of algorithms and structures among different DBMSs. As we have stated in the past, by developing libraries of these atoms, and providing the means by which to specify atomic compositions, customized database systems can be developed very quickly and cheaply. The success of a building-blocks technology for DBMSs rests on its simplicity and its exploitation of software reusability. In this way, we see the role of rule-based algebras complimenting existing research in software engineering.

Observation 4. Designing a DBMS is a very difficult art. As a research community, we understand how individual atomic algorithms work, and we are good at designing such algorithms. The algebra in this paper shows that compositions of atoms define DBMSs. Yet our understanding is incomplete. It is not obvious why certain layers were used in a DBMS, or why they are used in a particular order, when other layers and orderings could have been used. Nor is it obvious why certain algorithms were used within a layer as opposed to others. Furthermore, it is not clear when rewrite rules can be applied to yield faster algorithms (e.g., (E9) in Section 7.2), and what are good ways to schedule the parallel execution of functions. Although the algebra provides a tool to specify DBMS designs and to understand the building blocks of DBMSs, how to use this tool effectively (and hence how to best design customized DBMSs) still remains a major open problem.

9. Related Research

As a precursor to extensible database research, Yao unified a number of query processing algorithms in a constructive way using primitives, but did not relate these primitives to implementations of DBMSs [Yao79]. Rule-based algebras retain the spirit of Yao's work, and provides a formalism to go beyond it in many ways.

More recently, a number of independently conceived and developed proposals relating rule-based algebras to extensible DBMSs have appeared. The work of Graefe and DeWitt [Gra87] consider the problems of building efficient rule-based optimizers. The works of Freytag [Fre87] and, more recently, Lohman [Loh87b] are much closer to the framework that we have presented in this paper. Both of these works show how query optimization can be divided into phases, where each phase has its own rule set. Lohman goes further by explaining how query optimizers work in a rule-based setting that is quite similar to ours. Also, he makes the important connection of atomic algorithms as DBMS building blocks. The phases of optimization in both of these papers correspond to the mappings of retrieval operations through our layers (parameterized types). Our formalism shows how these works and their exposition of query optimization algorithms can be understood in the broader context of DBMS storage structures.

10. Conclusions

Parameterized types provide a simple way to describe DBMSs in a layered manner. Rule-based algebras express the abstract-to-concrete mappings of operations (algorithms) of these types. We have used this framework and have developed a notation to catalog and relate a large spectrum of query processing algorithms and storage structures. We have shown that important design concepts of centralized, distributed, and machine database architectures have algebraic representations, and that different DBMSs could be composed from a common pool of components.

Expressing DBMS implementations algebraically is an important step forward in specifying, understanding, and communicating their design. However, there are many aspects of DBMS implementation that we have not addressed in this paper. Modification operations (e.g., record insertion), recovery, concurrency control, new data types and operators, and performance models still need to be integrated into this framework. Preliminary results are already available [Bat82,86a-b,87b,88]. Further, the elementary basis of our formalism, namely data types and algorithms, holds promise in codifying knowledge in other areas of computer science. A prime candidate are data structures and their algorithms, the main memory counterparts to DBMSs.

We believe that codifying knowledge of DBMS implementations is an important step toward a technology that assembles DBMSs rapidly and cheaply from libraries of prewritten components. It is this framework that we are implementing in the GENESIS extensible DBMS project.

Acknowledgements. I gratefully acknowledge the help of Guy Lohman (IBM Almaden) and Mike Stonebraker (UC Berkeley) for information about R* and INGRES. I also thank Peter Dadam (IBM Heidelberg) for his thoughts on the presentation of this material.

References

[Ape83] P.M.G. Apers, A.R. Hevner, and S.B. Yao, 'Optimization Algorithms for Distributed Queries', **IEEE Trans. Software Engr.**, 9,1 (Jan. 1983), 57-68.

[Ast76] M.M. Astrahan, et al., 'System R: A Relational Approach to Database Management', **ACM Trans. Database Syst.**, 1,2 (June 1976), 97-137.

[Bal79] C. Baldissera, G. Brachi, and S. Ceri, 'A Query Processing Strategy for Distributed Databases', **Proc. EURO-IFIP Congress**, North Holland, 1979.

[Bat82] D.S. Batory and C.C. Gotlieb, 'A Unifying Model of Physical Databases', **ACM Trans. Database Syst.**, 7,4 (Dec. 1982), 509-539.

[Bat84] D.S. Batory, 'Conceptual-To-Internal Mappings in Commercial Database Systems', **ACM PODS 1984**, 70-78.

[Bat85] D.S. Batory, 'Modeling the Storage Architectures of Commercial Database Systems', **ACM Trans. Database Syst.**, 10,4 (Dec. 1985), 463-528.

[Bat86a] D.S. Batory, et al., 'GENESIS: An Extensible Database Management System', to appear in **IEEE Trans. Software Engr.**

[Bat86b] D.S. Batory and T.Y. Leung, 'Implementation Concepts for an Extensible Data Model and Data Language', TR-86-24, University of Texas at Austin, 1986.

[Bat87a] D.S. Batory, 'Principles of Database Management System Extensibility', in [IEE87a].

[Bat87b] D.S. Batory, 'Principles of Extensible Query Optimization', in [IEE87b].

[Bat87c] D.S. Batory, 'A Rule-Based View of Query Processing Algorithms', to appear in **Information Sciences.**

[Bat88] D.S. Batory, 'Concepts for a Database System Compiler', to appear in **ACM PODS 1988.**

[Ber81a] P.A. Bernstein and D.M. Chiu, 'Using Semi-Joins to Solve Relational Queries', **Jour. ACM**, 28,1 (Jan. 1981), 25-40.

[Ber81b] P.A. Bernstein, et al., 'Query Processing in a System for Distributed Databases (SDD-1)', **ACM Trans. Database Syst.**, 6,4 (Dec. 1981), 602-625.

[Bla77] M.W. Blasgen and K.P. Eswaren, 'On the Evaluation of Queries in a Relational Database System', **IBM Systems Jour.**, 16 (1976), 363-377.

[Blo70] B.H. Bloom, 'Space/Time Tradeoffs in Hash Coding with Allowable Errors', **Comm. ACM**, 13,7 (July 1970), 442-426.

[Cer84] S. Ceri and G. Pelagatti, **Distributed Databases: Principles and Systems**, McGraw-Hill, New York, 1984.

[Cha76] D.D. Chamberlin, et al., 'SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control', **IBM Jour. Res. and Dev.**, 20,6 (Nov. 1976), 560-575.

[Cha81] D.D. Chamberlin, et al., 'Support for Repetitive Transactions and Ad Hoc Queries in System R', **ACM Trans. Database Syst.**, 6,1 (March 1981), 70-94.

[Che86] J-P. Cheiney, et al., 'A Reliable Parallel Backend Using Multiattribute Clustering and Select-Join Operator', **VLDB 1986**, 220-227.

[Chi84] D.M. Chiu, P.A. Bernstein, and Y.C. Ho, 'Optimizing Chain Queries in a Distributed Databse System', **SIAM Jour. Computing**, 13,1 (Feb. 1984), 116-134.

[Chu82] W.W. Chu and P. Hurley, 'Optimal Query Processing for Distributed Database Systems', **IEEE Trans. Computers**, 31 (Sept. 1982), 835-850.

[DeW86] D.J. DeWitt, et al., 'GAMMA - A High Performance Dataflow Database Machine', **VLDB 1986**, 228-237.

[Eps78] R. Epstein, M. Stonebraker, and E. Wong, 'Distributed Query Processing in a Relational Database System', **ACM SIGMOD 1978**, 169-180.

[Eps80] R. Epstein and M. Stonebraker, 'Analysis of Distributed Database Processing Strategies', **VLDB**

1980, 92-101.

[Fre87] J.C. Freytag, 'A Rule-Based View of Query Optimization', **ACM SIGMOD 1987**, 173-180.

[Fus86] S. Fushimi, M. Kitsuregawa, and H. Tanaka, 'An Overview of the System Software of a Parallel Relational Database Machine GRACE', **VLDB 1986**, 209-219.

[Gan87] R.A. Ganski and H.K.T. Wong, 'Optimization of Nested SQL Queries Revisited', **ACM SIGMOD 1987**, 23-33.

[Ger86] R.H. Gerber, 'Dataflow Query Processing Using Multiprocessor Hash-Partitioned Algorithms', TR-672, Dept. Computer Sci., University of Wisconsin, 1986.

[Gog84] J. Goguen, 'Parameterized Programming', **IEEE Trans. Software Engr.**, SE-10,5 (September 1984), 528-543.

[Goo82] N. Goodman and O. Shmueli, 'The Tree Property is Fundamental for Query Processing', **ACM PODS 1982**, 40-48.

[Gra87] G. Graefe and D.J. DeWitt, 'The EXODUS Optimizer Generator', **ACM SIGMOD 1987**, 160-172.

[Hae78] T. Haerder, 'Implementing a Generalized Access Path Structure for A Relational Database System', **ACM Trans. Database Syst.**, 3,3 (September 1978), 285-298.

[Hev79] A.R. Hevner and S.B. Yao, 'Query Processing in Distributed Database Systems', **IEEE Trans. Software Engr.**, 5,3 (May 1979), 177-187.

[IEE87a] **IEEE Database Engineering,** Extensible Database Systems, M. Carey (ed), June 1987.

[IEE87b] **IEEE Database Engineering,** Query Optimization, G. Lohman (ed), November 1987.

[Kam83] Y. Kambayashi and M. Yoshikawa, 'Query Processing Utilizing Dependencies and Horizontal Decomposition', **ACM SIGMOD 1983**, 55-57.

[Kam85] Y. Kambayashi, 'Processing Cyclic Queries', in [Kim85], 62-80.

[Kan87] H. Kang and N. Roussopoulos, 'Combining Joins and Semijoins in Distributed Query Processing', **VLDB 1987.**

[Ker79] L. Kerschberg, P.D. Ting, and S.B. Yao, 'Query Optimization in Star Computer Networks', **ACM Trans. Database Syst.**, 7,4 (Dec. 1982), 678-711.

[Kim82] W. Kim, 'On Optimizing an SQL-like Nested Query', **ACM Trans. Database Syst.**, 7,3 (Sept. 1982), 443-469.

[Kim85] W. Kim, D.S. Reiner, and D.S. Batory, ed., **Query Processing in Database Systems**, Springer-Verlag, 1985.

[Kit83] M. Kitsuregawa, H. Tanaka, and T. Moto-oka, 'Application of Hash to Database Machine and its Architecture', **New Generation Computing**, 1,1 (1983), 63-74.

[Kor86] H.F. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, 1986.

[Laf86] S. LaFortune and E. Wong, 'A State Transition Model for Distributed Query Processing', **ACM Trans. Database Syst.**, 11,3 (Sept. 1986), 294-322.

[Lis77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, 'Abstraction Mechanisms in CLU', **Comm. ACM**, 20,8 (Aug. 1977), 564-576.

[Loh85] G.M. Lohman, et al., 'Query Processing in R*', in [Kim85], 31-47.

[Loh87a] G.M. Lohman, private correspondence.

[Loh87b] G.M. Lohman, 'Grammar-like Functional Rules for Representing Query Optimization Alternatives', RJ 5992, IBM Almaden Research Center, December 1987.

[Mac86a] L.F. Mackert and G.M. Lohman, 'R* Optimizer Validation and Performance Evaluation for Local Queries', **ACM SIGMOD 1986**, 84-95.

[Mac86b] L.F. Mackert and G.M. Lohman, 'R* Optimizer Validation and Performance Evaluation for Distributed Queries', **VLDB 1986**, 149-159.

[Par83] H. Partsch and R. Steinbruggen, 'Program Transformation Systems', **ACM Computing Surv.**,

15,3 (Sept. 1983), 199-236.

[Ros82] A. Rosenthal and D. Reiner, 'An Architecture for Query Optimization', **ACM SIGMOD 1982**, 246-255.

[Sac86] D. Sacco, 'Fragmentation: A Technique for Efficient Query Processing', **ACM Trans. Database Syst.**, 11,2 (June 1986), 113-133.

[Seg86] A. Segev, 'Optimization of Join Operations in Horizontally Partitioned Database Systems', **ACM Trans. Database Syst.**, (March 1986), 48-80.

[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.

[Sha86] L.D. Shapiro, 'Join Processing in Database Systems with Large Main Memories', **ACM Trans. Database Syst.**, (Sept. 1986), 265-293.

[She87] S.T. Shenoy and Z.M. Ozsoyoglu, 'A System for Semantic Query Optimization', **ACM SIGMOD 1987**, 181-195.

[Sto76] M. Stonebraker, E. Wong, P. Kreps, and G. Held, 'The Design and Implementation of INGRES', **ACM Trans. Database Syst.**, 1,3 (Sept. 1976), 189-222.

[Teo82] T.J. Teorey and J.P. Fry, *Design of Database Structures*, Prentice-Hall, 1982.

[Tur79] M.J. Turner, R. Hammond, and P. Cotton, 'A DBMS for Large Statistical Databases', **VLDB 1979**, 319-327.

[Val85] P. Valduriez and G. Gardarin, 'Join and Semijoin Algorithms for a Multiprocessor Database Machine', **ACM Trans. Database Syst.**, 9,1 (March 1984), 133-161.

[Val87] P. Valduriez, 'Join Indices', **ACM Trans. Database Syst.**, 12,2 (June 1987), 218-246.

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

[Won86] H.K.T. Wong, et al., 'Bit Transposition in Very Large Scientific and Statistical Databases', **Algorithmica**, (1986), 289-309.

[Yao79] S.B. Yao, 'Optimization of Query Evaluation Algorithms', **ACM Trans. Database Syst.**, 4,2 (June 1979), 133-155.

[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.

Appendix 1. Some Basic Operations

	Basic operations on stored files (retrieval only)
RET(F,Q,O)	Generate a stream of records in O order from stored file F that satisfy restriction predicate Q.
ACC(F,Q,S)	S is a stream of pointers. Follow each pointer of S to access a record from stored file F. Output the stream of records that satisfy predicate Q.
	Basic operations on links (retrieval only)
JOIN(F1,F2,J,O)	Generate the stream of records in O order that represent the join of files F1 and F2 via join predicate or link J. F1 and F2 can be stream or stored.
JF(F1,F2,J,O)	Eliminate records from F1 that cannot participate in a join with F2 via join predicate or link J. A stream of F1 records are output in O order. F1 and F2 can be stream or stored.
CJOIN(F1,F2,J,O)	Same as JOIN except that CJOIN is mapped between layers.
CJF(F1,F2,J,O)	Same as JF except that CJF is mapped between layers.
	Basic operations on stream files
SORT(S,O)	Produce the stream that is stream S sorted into O order.
STORE_TMP(T,S,O)	Store stream S in O order in temporary file T. STORE_TMP produces no output.
FILTER(S,Q)	Eliminate records of S that do not satisfy predicate Q.
SPLIT(S $\mid \xi_{i=1}^{n} X_i$)	Split S into n substreams $X_1 \cdots X_n$.
ASSEMBLE($\xi_{i=1}^{n} X_i \mid$ O)	Assemble n substreams $X_1 \cdots X_n$ into a single stream with records arranged in O order.

Appendix 2. Some File Characteristic Rewrites and Identities

Let ∅ be the null stream. Some rewrites using file characteristics are:

∅	<->	RET(F, Q, O)	; if (MEMBER(F) and Q) = false	(C1)
		JOIN(F1, F2, J, O)	; if (MEMBER(F1) and Q(F1,R) and	(C2)
			; MEMBER(F2) and Q(F2,R)) = false	
S	<->	SORT(S, *)		(C3)
		SORT(S, ORDER(S))		(C4)
		FILTER(S, MEMBER(S))		(C5)
		FILTER(S, null)		(C6)

Some MEMBER and ORDER identities:

Ω	ORDER(Ω)	MEMBER(Ω))
RET(F,Q,*)	ORDER(F)	Q
SORT(S,O)	O	MEMBER(S)
ACC(F,Q,S)	ORDER(S)	MEMBER(S) and Q
FILTER(S,Q)	ORDER(S)	MEMBER(S) and Q
JOIN(F1,F2,J,O)	O	MEMBER(F1) and MEMBER(F2) and J
JF(F1,F2,J,O)	O	MEMBER(F1)
MERGE_SCAN(Fo,Fi,J)	J	MEMBER(Fo) and MEMBER(Fi) and J
HASH_JN(Fo,Fi,J)	*	MEMBER(Fo) and MEMBER(Fi) and J
NESTED_LOOPS(Fo,Fi(_),J)	ORDER(Fo)	MEMBER(Fo) and MEMBER(Fi) and J
SEMIJOIN(Fo,Fi,J,O)	O	MEMBER(Fo) and J
BLOOM(Fo,Fi,J,O)	O	MEMBER(Fo)

Note that the MEMBER predicate for the BLOOM semijoin operation is approximate and not as restrictive as the MEMBER predicate for the SEMIJOIN operation.

Appendix 3. Augment RTI and Horizontal Partitioning FILE Types

A3.1 The Augment RTI Type

Let AUG[cf:FILE] be the FILE type that augments the **record type identifier** (RTI) of abstract file AF onto each record in AF. Let RT be the augmented field and let α, the RTI of AF, be its contents. Parameter cf is the implementation of the concrete file F that is produced. F has the following characteristics:

$$\begin{aligned} MEMBER(F) &= MEMBER(AF) \text{ and } (RT=\alpha) \\ Q(F,R) &= Q(AF,R) \text{ and } (RT=\alpha) \end{aligned}$$

With the exception of a special join algorithm, the RET, ACC, and CJOIN operations map directly to their concrete counterparts:

RET(AF, Q, O) => RET(F, Q and RT=α, O) (G1)

ACC(AF, Q, S) => ACC(F, Q and RT=α, S) (G2)

CJOIN(AFo, AFi, J, O) => JOIN(Fo, Fi, J, O) (G3)

SORT(GRACE_JN(STREAM(Fo, Fi), J), O) ; J is a predicate (G4)

where:

STREAM(A, B) => COMBINE(STR(A,*), STR(B,*))

and STR(F,s) was defined in Section 4.

GRACE_JN is a special join algorithm that was invented for the GRACE database machine [Kit83, Fus86]. It works in the following way. Assume the records of stream files Fo and Fi have been augmented with RTIs, and Fo and Fi are to be joined over predicate J. Fo and Fi are merged into a single stream and sorted on (join value, RTI) pairs. (This achieves the effect of sorting Fo and Fi simultaneously). The sorted stream is then partitioned into substreams that have the same join value. Each substream for a join value can be partitioned further into a pair of substreams: one that contains only Fo records and another only Fi records. Taking the cross product of each substream pair yields the join of Fo and Fi.

The GRACE_JN algorithm is not atomic but is a variant of the sort-merge join algorithm. To be consistent with our use of nonredundant catalogs and transformational rewrites, (G4) should be eliminated. However, we have retained them for exposition reasons as they are used in Section 8.3 and Appendix 4.

A3.2 The Horizontal Partitioning Type

Let HPART[fp:FILE] be the FILE type that horizontally partitions abstract file AF into n subfiles $F_1 \cdots F_n$. Parameter fp is the implementation of each of these subfiles.

Horizontal partitioning can occur in one of three ways: 1) **load balancing** - record insertions are directed to the subfile which has the fewest number of records. Load balancing attempts to equalize the number of records in each subfile. 2) **key ranges** - each subfile is identified with a disjoint range of keys. All records whose keys belong to a given range are stored in the same subfile. 3) **hashed** - each subfile is identified with a hash key or range of hash keys. All records that share the same hash key are stored in the same subfile.

Let P_i be the partitioning predicate for subfile F_i. (For load balanced files, P_i=true. That is, partitioning is not based on file contents). Horizontal partitioning assigns F_i the following membership characteristic:

$$MEMBER(F_i) = MEMBER(AF) \text{ and } P_i$$

It turns out that the abstract-to-concrete mappings of the RET, ACC, and CJOIN algorithms are independent of the partitioning method used. The basic idea is to distribute an operation over all subfiles, and to assemble the results from each subfile. Using the notation $\overset{n}{\underset{i=1}{\xi}} G_i$ to denote the list of functions $G_1 \cdots G_n$, and | to separate arguments that are lists of functions, we have:

$$RET(AF, Q, O) \Rightarrow ASSEMBLE(\overset{n}{\underset{r=1}{\xi}} RET(F_r, Q, O) \mid O) \qquad (H1)$$

$$ACC(AF, Q, S) \Rightarrow ASSEMBLE(\overset{n}{\underset{r=1}{\xi}} ACC(F_r, Q, S) \mid ORDER(S)) \qquad (H2)$$

Suppose AFo has n subfiles and AFi has m. Different algorithms arise for the CJOIN operation depending on the stored or stream nature of AFo and AFi. There is one specialization for each of the four possibilities:

CJOIN(AFo, AFi, AJ, O) =>

$$JOIN(AFo, AFi, AJ, O) \quad \text{; AFo and AFi are stream} \qquad (H3)$$

$$ASSEMBLE(\overset{m}{\underset{s=1}{\xi}} JOIN(AFo, Fi_s, J, O) \mid O) \quad \text{; AFo is stream, AFi is stored} \qquad (H4)$$

$$ASSEMBLE(\overset{n}{\underset{r=1}{\xi}} JOIN(Fo_r, AFi, J, O) \mid O) \quad \text{; AFo is stored, AFi is stream} \qquad (H5)$$

$$ASSEMBLE(\overset{n}{\underset{r=1}{\xi}} \overset{m}{\underset{s=1}{\xi}} JOIN(Fo_r, Fi_s, J, O) \mid O) \quad \text{; AFo and AFi are stored} \qquad (H6)$$

Ceri and Pelagatti [Cer84] used MEMBER predicates to show how equations (H1) and (H6) could be simplified by eliminating operations on subfiles that are guarenteed to produce a null stream as a result. Suppose two stored abstract files AFo and AFi are hash-partitioned using the same hashing function. Let $Fo_1 \cdots Fo_n$ and $Fi_1 \cdots Fi_n$ be their subfiles. A join of AFo and AFi over their partitioning keys reduces (H6) from n^2 joins to n joins (i.e., the join of Fo_r and Fi_r for r=1..n). Simplifications are possible for hashed and range key partitioning; no simplification is possible for load balanced partitionings.

We note that the use of file characteristics to simplify equations has practical importance. The GAMMA database machine query optimizer uses rudimentary file characteristics to eliminate unnecessary retrievals from subfiles [Dew86, Ger86].

Appendix 4. The RET and JOIN Algorithms of GRACE

The conceptual file retrieval operation in GRACE generates a stream of records from file F in a random order that satisfies predicate Q (i.e., RET(F,Q,*)). Its implementation follows from mappings through the composite type NETWORK[AUG[HPART[HEAP]], CONL[ALG]] that defines the GRACE architecture.

Step 1) Map RET through the AUG[ ] type. By (G1) we get:

$$RET_{GRACE}(F, Q, *) \quad => \quad RET(F, Q \text{ and } RT=\alpha, *)$$

Step 2) Map RET through the HPART[ ] type. Assume F maps to m partitions $F_1 \cdots F_m$, where each partition is stored on a separate disk. By (H1) and (M5) we get:

$$=> \quad COMBINE(\xi_{j=1}^{m} RET(F_j, Q \text{ and } RT=\alpha, *))$$

Step 3) Map RET through the HEAP type. Assume processor j can access the disk containing partition F_j. By (I1) and adding processor subscripts to the retrieval algorithms, we have:

$$=> \quad COMBINE(\xi_{j=1}^{m} HEAP_RET^{j}(F_j, Q \text{ and } RT=\alpha)) \qquad \text{(GR0)}$$

(GR0) is the algorithm that implements retrievals on conceptual files in GRACE.□

Now consider the conceptual join operation of GRACE. It joins stream files Fo (outer) and Fi (inner) on join predicate J, where joined records are not output in any order (i.e., JOIN(Fo,Fi,J,*)). Its implementation comes from the following mappings.

Step 1) Map JOIN to CJOIN via the CONL[ ] type:

$$JOIN_{GRACE}(Fo, Fi, J, *) \quad => \quad CJOIN(Fo, Fi, J, *)$$

Step 2). The CJOIN is mapped through the AUG[ ] type to be rewritten by (G4).

$$=> \quad GRACE_JN(COMBINE(Fo, Fi), J)$$

Step 3). The distributed rewrite (DR2) is applied to hash split streams Fo and Fi into n substreams each . Substream X_k of Fo and substream Y_k of Fi are joined by a GRACE_JN algorithm; a total of n joins are performed. The resulting streams are then combined. (Note that file characteristics were used to reduce the number of joins from n^2 to n):

$$=> \quad HSPLIT(Fo \mid \xi_{k=1}^{n} X_k);$$

$$HSPLIT(Fi \mid \xi_{k=1}^{n} Y_k);$$

$$COMBINE(\xi_{k=1}^{n} GRACE_JN(COMBINE(X_k, Y_k), J))$$

Step 4). The stream generated by the innermost COMBINE in Step 3 is spooled to a temporary file, and then reread. This is accomplished by the stream-to-stream (A2*A1) rewrite. Note that n temporary files $T_1 \cdots T_n$ are created:

$$=> \quad HSPLIT(Fo \mid \xi_{k=1}^{n} X_k);$$

$$HSPLIT(Fi \mid \xi_{k=1}^{n} Y_k);$$

$$\xi_{k=1}^{n} STORE_TMP(T_k, COMBINE(X_k, Y_k), *);$$

$$\text{COMBINE}(\overset{n}{\underset{k=1}{\xi}} \text{GRACE_JN}(\text{RET}(T_k, \text{null}, *), J))$$

Step 5) Without loss of generality, assume that streams Fo and Fi are actually combinations of substreams; the number of substreams can be different for Fo and Fi. Furthermore, we will assume that substream Fo_r (Fi_s) is produced by processor r (s):

$$\text{Fo} \quad \text{<-->} \quad \text{COMBINE}(\underset{r}{\xi} \text{Fo}_r) \qquad \text{(PS)}$$

$$\text{Fi} \quad \text{<-->} \quad \text{COMBINE}(\underset{s}{\xi} \text{Fi}_s)$$

Note the implementation of the conceptual RET operation has this form. Substream j is generated by $\text{HEAP_RET}^j(F_j, Q \text{ and } RT=\alpha))$. We'll explain the reason for this step later. We will refer to (PS) as the **partitioned stream** representation of Fo and Fi.

Step 6). Replace Fo and Fi with their (PS) representations. Then apply the assembly rewrite (AR1) to each HSPLIT function. We get:

$$\Rightarrow \quad \underset{r}{\xi} \text{HSPLIT}(\text{Fo}_r \mid \overset{n}{\underset{k=1}{\xi}} X_{r,k});$$

$$\underset{s}{\xi} \text{HSPLIT}(\text{Fi}_s \mid \overset{n}{\underset{k=1}{\xi}} Y_{s,k});$$

$$\overset{n}{\underset{k=1}{\xi}} \text{STORE_TMP}(T_k, \text{COMBINE}(\underset{r}{\xi} X_{r,k}, \underset{s}{\xi} Y_{s,k}), *);$$

$$\text{COMBINE}(\overset{n}{\underset{k=1}{\xi}} \text{GRACE_JN}(\text{RET}(T_k, \text{null}, *), J))$$

Step 7). Temporary files are implemented as heaps. Implementing heap retrieval as (I4), simplifying, and adding processor superscripts to all but the outermost COMBINE function, we get:

$$\Rightarrow \quad \underset{r}{\xi} \text{HSPLIT}^r(\text{Fo}_r \mid \overset{n}{\underset{k=1}{\xi}} X_{r,k}); \qquad \text{(GR1)}$$

$$\underset{s}{\xi} \text{HSPLIT}^s(\text{Fi}_s \mid \overset{n}{\underset{k=1}{\xi}} Y_{r,k}); \qquad \text{(GR2)}$$

$$\overset{n}{\underset{k=1}{\xi}} \text{STORE_TMP}^k(T_k, \text{COMBINE}^k(\underset{r}{\xi} X_{r,k}, \underset{s}{\xi} Y_{s,k}), *); \qquad \text{(GR3)}$$

$$\text{COMBINE}(\overset{n}{\underset{k=1}{\xi}} \text{GRACE_JN}^{\sigma(k)}(\text{HEAP_RET}^k(T_k, \text{null}), J)) \qquad \text{(GR4)}$$

σ is a scheduler function which assigns processor $\sigma(k)$ to process temporary file T_k. (GR1)-(GR3) corresponds to the **staging phase** of the GRACE join algorithm and (GR4) is the **processing phase.**□

Note that the conceptual JOIN operation is in partitioned stream form, i.e., its result is a combination of substreams. Thus, conceptual JOINs can consume the input of conceptual RET and other conceptual JOIN operations. (This was the reason for introducing (PS)). Thus, three conceptual files A, B, and C could be joined in GRACE by RETrieving each file and nesting JOINs within JOINs:

$$\underset{\text{GRACE}}{\text{JOIN}}(\underset{\text{GRACE}}{\text{JOIN}}(\underset{\text{GRACE}}{\text{RET}}(A, Q_A, *), \underset{\text{GRACE}}{\text{RET}}(B, Q_B, *), J_{AB}, *), \underset{\text{GRACE}}{\text{RET}}(C, Q_C, *), J_{BC}, *)$$