The One-to-One Match Operator of the Volcano Query Processing System

Tom Keller, Goetz Graefe

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 89-009

June, 1989

The One-to-One Match Operator of the Volcano Query Processing System

Tom Keller, Goetz Graefe

Oregon Graduate Center
Beaverton, Oregon 97006-1999

Abstract

Much of the current research on relational database systems focuses on increasing the functionality and flexibility of query processing. Query processing in relational databases is based on relational operators. There are a number of theoretical operations, however, that are typically not included in commercial systems. For example, most commercial systems provide the natural join operation but not all provide relational operators such as semi-join and outer join.

We present an operator that exploits the similarity between binary relational operators. Binary operators are those that produce a single output relation from two input relations. The *one-to-one match* operator has the capability of computing a class of binary relational operations.

The hash-based implementation of *one-to-one match*, currently used in the Volcano query processing system, is described in this paper. This implementation can compute the following binary relational operations: natural join, semi-join, outer join, anti-join, union, intersection, difference, anti-difference and Cartesian product. Furthermore, it is able to perform aggregate functions and duplicate elimination in conjunction with these binary operators. The implementation is based on the classical hash join algorithm. By enhancing the simple hash join algorithm, the functionality of the algorithm is increased without severely affecting execution time. In addition, user supplied arguments determine the operation *one-to-one match* performs, making it very flexible.

1. Introduction

Among the many proposed data models, the relational model, first proposed in [1], is most widely used in current commercial database systems. The firm mathematical theory behind the relational model [2,3,4] and efficient implementations have contributed to its success [5,6]. Much of the current research on relational database systems focuses on increasing the functionality, flexibility and speed of query processing. Each year, new query processing algorithms and improvements of old query processing algorithms are proposed.

Query processing in relational databases utilizes relational operators. The five fundamental relational algebra operators are *select*, *project*, *Cartesian product*, *union* and *set-difference*.

Of particular interest are binary relational operators; those operators that output a single relation given two relations as input. One of the most heavily used binary relational operators is the *join*. Other binary operators include *semi-join*, *outer join*, *union*, *intersection* and *difference*.

The binary relational operations mentioned above have a common basis of execution. This means that it is possible to do all of the operations using a single module. Volcano's *one-to-one match* is an operation capable of performing a class of binary relational operators including the operations listed above.

Two implementations of one-to-one match are used in the Volcano query evaluation system. One implementation is based on sort-merge [7]. The sort-based algorithm has all the abilities of the hash-based implementation except that aggregate functions and duplicate elimination are done when the inputs are sorted, not in one-to-one match. The hash-based one-to-one match operator is the focus of this paper.

This paper is organized as follows. Section 2 briefly surveys other join, aggregation and duplicate elimination algorithms with special attention on those that are hash-based. A brief overview of the Volcano query evaluation system is given in Section 3. In Section 4 the background of the one-to-one match operator is presented and the hash-based implementation is explained in detail. Some experimental performance results are presented in Section 5. Section 6 contains directions for future work.

2. Related Work

2.1. Relational Join

Join is a binary relational operator used for combining two relations. It is one of the most time-consuming relational operators and also one of the most frequently used. A great amount of research effort has been given to the development of join algorithms and improvements of existing algorithms.

Enrollment

	Name	Course Number
E1	Adam	1
E2	Adam	2
E3	Betty	1
E4	Carol	2
E5	Denny	3
E6	Earl	4
E7	Frank	5

Table 1. Enrollment Relation.

Course

	Course Number	Name
C1	1	Data Structures
C2	2	Algorithms
C3	3	Architecture
C4	4	Database

Table 2. Course Relation.

Tables 1 and 2 represent two example relations used throughout the paper. Consider the *Enrollment* relation (Table 1). For each course that a student takes there is one tuple in the *Enrollment* relation. Assume that each student is uniquely identified by name so that the first two tuples in the table represent courses taken by the same person (i.e. Adam is taking course number 1 and 2). Table 2 contains only those course numbers and names of Computer Science courses. The labels consisting of a letters followed by a number are unique tuple identifiers. For example, E1 is interpreted as tuple one of the *Enrollment* relation.

Student-Course

	Enrollment.Name	Course Number	Course.Name
SC1	Adam	1	Data Structures
SC2	Adam	2	Algorithms
SC3	Betty	1	Data Structures
SC4	Carol	2	Algorithms
SC5	Denney	3	Architecture
SC6	Earl	4	Database

Table 3. Student-Course Relation.

The natural join of these two relations is shown in Table 3. Tuples from each relation that have equal values in common fields, *Course* Number in the example, are "joined" to produce an output tuple. The common field or attribute is called the *join attribute.* For instance, tuple E1 from the *Enrollment* relation and tuple C1 from the *Course* relation match and produce the output tuple SC1.

2.2. Non-Hash-Based Join Algorithms

The most natural, and most naive, way to compute the join is called *nested loops join* or *simple iteration.* For each tuple in one of the relations, say the *Enrollment* relation, the entire other relation, *Course,* is scanned sequentially for matching tuples. Each tuple in one relation is compared with all tuples in the other relation. The algorithmic complexity of *nested loops join* is the product of the relation sizes.

A simple optimization can be made to the *nested loops join* algorithm. Instead of processing each relation a tuple at a time, they are processed a block at a time. This algorithm is called *block nested loops* join. Now the *Course* relation is scanned once for each block of tuples in *Enrollment* instead of each tuple in *Enrollment.* The number of times *Course* must be scanned is reduced by a factor equal to the number of *Enrollment* tuples that fit in one block.

If there is an index on the join attribute, Course Number in our example, the scan of the second relation is reduced to an index lookup. This modification to *nested loops join* is called *index nested loops.* It has the advantage of reducing the time needed to search for matching tuples from a file scan to an index lookup.

Another group of join algorithms are those which are merge-based and are consequently named *merge join.* The first step is to sort both relations on the join attribute if they are not already sorted. Once sorted on the join attribute, joining the relations can be accomplished by well known merge techniques [8]. Blasgen and Eswaran provide a detailed description of the *sort-merge join* algorithm [7].

As in the case for ***nested loops join***, ***sort-merge join*** can benefit from the use of an index on the join attribute. Consider the case where *Enrollment* is sorted by join attribute but *Course* is not. Also, there is an index on Course Number (the join attribute) for the *Course* relation. Because the *Course* index contains references to the tuples of *Course* sorted on Course Number, the index can serve in place of the sorted *Course* relation. However, because *Course* is not physically sorted by Course Number, more storage accesses will be needed than if it were sorted.

2.3. Hash-Based Join Algorithms

First, we describe ***classic hashing*** as described in [9]. A main memory hash table is built with tuples from the smaller relation, the ***build input***, hashed on the join attribute. As the second relation, the ***probe input***, is scanned, the hash value of each tuple is used to probe the hash table for matches.

To see one advantage hash-based join algorithms offer over sort-based algorithms, consider joining N relations. Specifically, the N relations are to be joined in the order shown in Figure 1. If a sort-based algorithm is used, the relations R and S are sorted and joined. The next join operation, RS joined with T, must wait until the join of R and S is complete. This is necessary

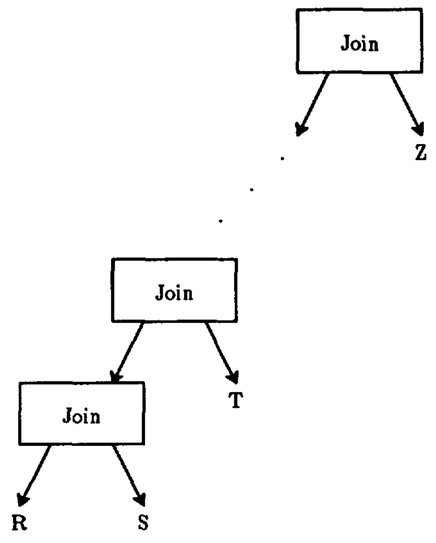

Figure 1: Join of N Relations

because RS must be sorted before it can be joined with T and the entire result is needed for sorting. With sort-based join algorithms, no more than one join operation can execute at any given time. However, hash-based algorithms allow more dataflow. As soon as a tuple of RS is produced by the first join it can be consumed by the next join. With hash-based join algorithms possibly all join operation can be going on simultaneously if the output of each join is the *probe input* to another join. That is, if the tuple being produced by each join are being used to probe an existing hash table, not being used to build it, hash-based joins have greater parallel execution potential. In the example, if RS were being used to fill the hash table of the next join then the next join would have to wait until the join of R and S was complete. Only after the entire hash table has been filled with tuples from RS can the table be probed with tuples of T to produce results.

Classic hashing fails when the main memory cannot hold the entire hash table. This situation is known as *hash table overflow*. Each of the following sections details a method of handling *hash table overflow*.

2.3.1. Simple Hash Join

Simple hash join [10,11] is identical to classic hash join until hash table overflow occurs. The simple hash join algorithm handles overflow in the simplest way possible (hence the name). When overflow occurs, a portion of the hash table is written to an overflow file on disk. In Figure 2 the hash table is being built with relation R when overflow occurs. Now, hashing tuples

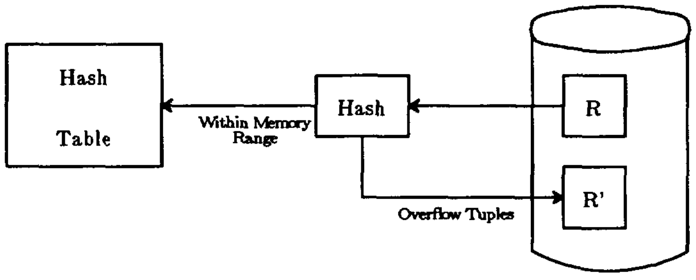

Figure 2: Hash Table Overflow While Building Hash Table

becomes more complicated. If a tuple hashes to the range of values contained in main memory it is inserted into the hash table. However, a tuple that hashes outside the range is appended to the overflow file R'.

While scanning the second relation a similar situation occurs. A tuple that hashes within the range of values contained in main memory is processed in the same way as classic hash join. Other tuples are written to another overflow file on disk, S', as shown in Figure 3. When the entire second relation has been scanned, processing does not halt. Instead, the two overflow files, R' and S', must now be joined. This continues until all tuples have been processed.

Although overflow is handled in a simple and natural way, simple hash join performs poorly when the smallest of the relations is many times larger than main memory.

2.3.2. Grace Algorithm

Simple hash join handles overflow by ***resolving*** it when it occurs, an optimistic approach. Another method of handling overflow is to ***avoid*** it, as the people working on the Grace database machine have. The Grace algorithm [12,13,14] uses ***overflow avoidance*** to prevent overflow before it occurs.

Instead of partitioning the relations when overflow occurs, the relations are partitioned prior to joining them. The first step of the Grace algorithm partitions both R and S into N disjoint sets as shown in Figure 4. By choosing N appropriately, every partition can be made small enough to fit in main memory, thus avoiding overflow. Partitioning is accomplished by hashing

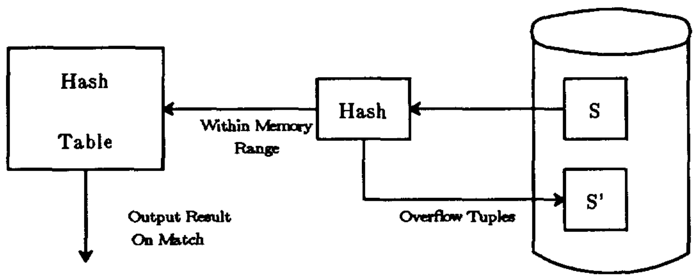

Figure 3: Hash Table Overflow While Probing Hash Table

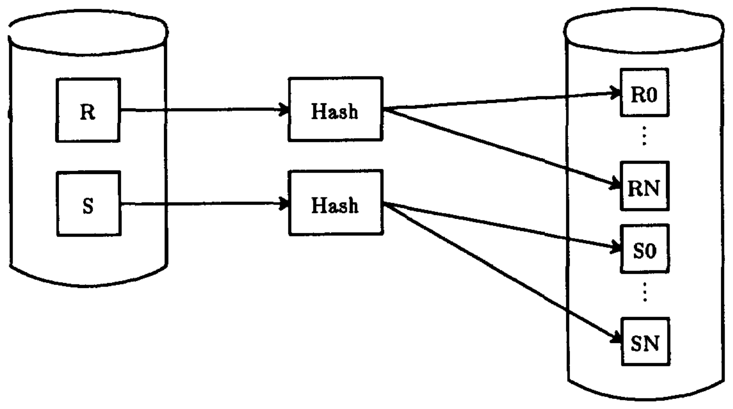

Figure 4: Grace Hash Join

on the join attributes of both relations. Once the relations have been partitioned, each partition pair can be joined.

To make certain that overflow does not occur, many small partitions may be created. When the partitions only take up a fraction of main memory, more than one partition may be joined at a time. The Grace algorithm requires two passes over the data. As a result, it only outperforms simple hash join when the smallest relation is many times the size of main memory. In this case simple hash join makes repeated passes over the same data because overflow occurs a number of times. Grace, however, only makes the two passes over the data.

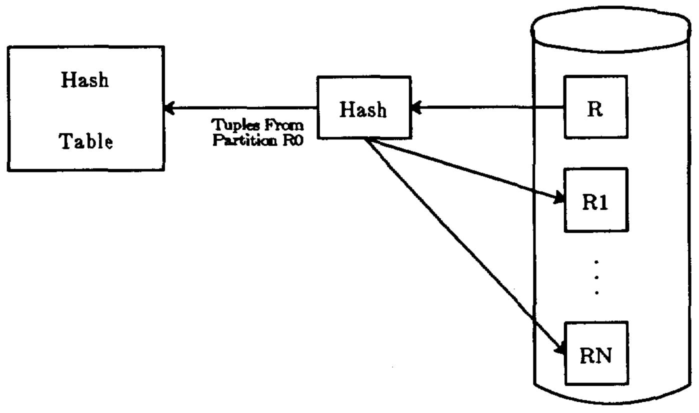

Figure 5: Hybrid Hash Join Build.

2.3.3. Hybrid Hash Join

Hybrid hash join [10] combines features of simple hash join and Grace hash join. The Grace algorithm first partitions the relations and then begins processing the partitions. Hybrid hash join processes one of the partitions while doing the partitioning. Thus, hybrid hash join begins by building the hash table from one partition of relation R while spooling the other partitions out to disk. When relation R has been partitioned, relation S is processed. The first partition of S can be immediately joined with the first partition of R while the remaining partitions of S are also spooled to disk. This process is shown in Figures 5 and 6.

For relations that are many times larger than main memory, hybrid hash join behaves similar to Grace hash join. When the relation size is slightly larger than main memory, the algorithm performs like simple hash join.

2.4. Join Algorithm Comparison

It is beyond the scope of this paper to compare and analyze the different join algorithms. The sections above allude to relative algorithm advantages. Detailed comparison of sort-merge, simple hash, Grace hash and hybrid hash join for a uniprocessor system is given in [9,10]. The same algorithms are analyzed for shared-memory multiprocessor systems in [11,15] and for shared-nothing multiprocessor systems in [16]. In [17], a number of sort-based and hash-based

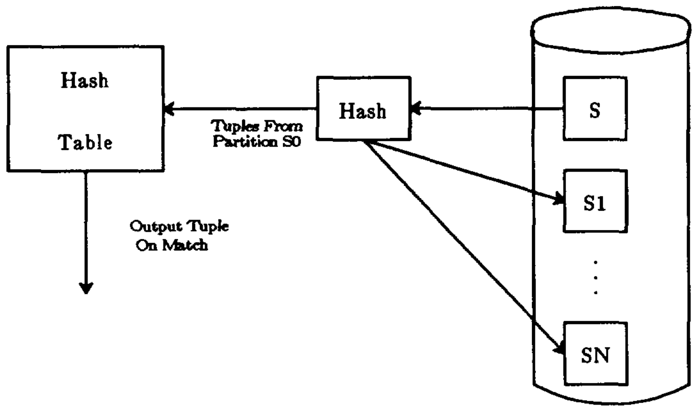

Figure 6: Hybrid Hash Join Probe

pipelined multiprocessor join algorithms are described and analyzed. A hash-based join algorithm for a specific multiprocessor architecture, a cube-connected parallel computer, is presented in [18].

2.5. Aggregate Functions and Duplicate Elimination

The one-to-one match operator has the ability to perform aggregate functions and duplicate elimination. Notice that duplicate elimination can be considered a special form of aggregate functions. An aggregate function is an operation that computes a piece of information about groups of tuples in a relation. A group is defined as a set of tuples with equal values for a specified attribute. Common aggregate operations are *count*, *max* and *sum*.

As an example, suppose that we need to know how many courses each student is enrolled in. This is the result of performing a *count* aggregate operation on the *Enrollment* relation (Table 1). In this example the tuples need to be grouped by the Name attribute. The number of tuples in each group is counted to produce the aggregation result. Table 4 shows the result of this aggregation. The attribute Name is the *grouping attribute* and the aggregate operation is *count*. Techniques for performing aggregation are discussed in greater detail in [19,4,20].

3. The Volcano Query Evaluation System

In this section we present aspects of the Volcano query processing system used in the remainder of this paper. For a complete description of Volcano, see [21].

Course Load

	Name	Number of Courses
CL1	Adam	2
CL2	Betty	1
CL3	Carol	1
CL4	Denny	1
CL5	Earl	1
CL6	Frank	1

Table 4. Aggregate Function — Number of Courses Taken By Each Student

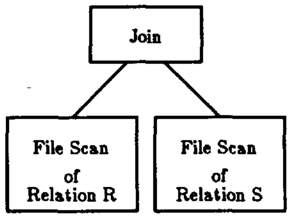

Figure 7: Join of Relations R and S

A query in Volcano is expressed as an algebra expression or tree. Each node of the tree represents an operation. For example, a join of two relations, R and S, is expressed by the tree shown in Figure 7. A *File Scan* node reads tuples of a relation by scanning the file they are stored in. Tuples read by the *File Scans* are consumed by the *Join* operation. The *Join* operation uses these tuples to compute the join of the two relations.

Each operator used in forming query trees is implemented as an *iterator*. That is, all operators are implemented by three procedures: *open*, *next*, and *close*. In addition to requiring all operations to be implemented as iterators, all operations are required to have a uniform argument list. The standardized interface and argument list mean that an operator does not need to know what operator produces its input. We call this concept *anonymous input* or *streams*. Returning to the example, *Join* has two input streams, both of them happen to be file scans. However, with the standardized interface and argument list, *Join* only sees its inputs as iterators. So, the inputs to *Join* can be any valid Volcano operator without affecting the *Join* algorithm.

As a consequence of the iterator paradigm, each operator must maintain its state between function calls. This is analogous to the need for local state in coroutines [22]. In the example, each *next* call to *File Scan* returns a single tuple of the relation. When a call to *next* returns the tuple, the iterator state must be saved so that the following call to *next* will return the next tuple from the file being scanned. This is accomplished in Volcano by the use of *state records*. In addition to storing iterator state between calls to *next*, the state records in Volcano contain pointers to the iterator(s) used for input to the operator.

Volcano incorporates all of the query information into Query Evaluation Plans (QEPs). Figure 8 shows a more detailed view of the join example. It is now apparent how the operations are linked together. Each QEP contains pointers to the functions implementing the operator, namely *open*, *next* and *close*. QEPs also contain another pointer to the state record of the operator. The state record contains, among other things, *input* pointers to QEPs. In Figure 8, the join operation has two inputs, *input1* and *input2*, that are themselves operators.

One-to-one match is a single module in the Volcano query evaluation system. It can compute a number of binary relational algebra operators, e.g., join, semi-join, outer join, union, intersection, difference, aggregate functions, and projection (duplicate elimination). One operation not directly computable by the one-to-one match operator is *relational division*. Relational division belongs to another class of binary relational operators and is handled by Volcano's *one-to-many match* operator. Two modules are used to carry out *one-to-many match*. The first module uses a standard algorithm, called *naive division*, to compute division. Division is also

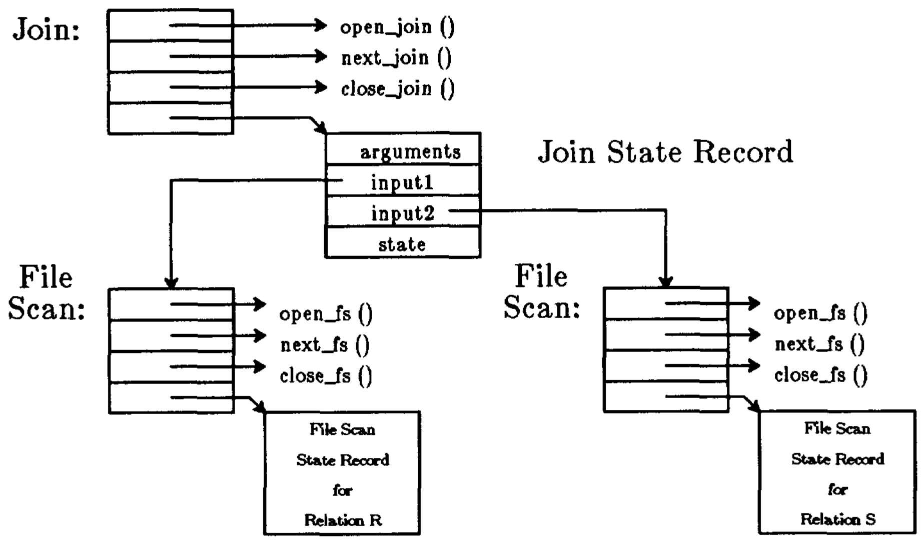

Figure 8: Query Evaluation Plan
Implementing Join of Relations R and S

provided by a new algorithm called ***hash-division*** which is described and compared to conventional division algorithms in [23]. It turns out, however, that multiple one-to-one match operators can be combined to compute a relational division using aggregate functions [23].

Another module essential to any database system is a sort module. Volcano's sorting algorithms are presented in [24]. This module is used, for example, to complement the sort-based one-to-one match operator, which is based on merge join.

To simplify the content of this paper, we assume that one-to-one match and Volcano are confined to a uniprocessor system. However, parallelism is a primary feature of Volcano and is encapsulated by Volcano's *exchange* operator [25].

4. One-to-One Match

4.1. Binary Operators Suited for One-to-One Match

This section focuses on one-to-one match independent of the algorithm implementing it. As mentioned previously, one-to-one match can compute natural join, semi-join, outer join, union, intersection, difference, anti-difference and Cartesian product. Each of these binary operators are based on a single principle. In all of these operations, a tuple is included in the operation's result depending on the outcome of one comparison with another tuple. To illustrate this principle, we focus on a sample of these binary operators.

4.1.1. Natural Join

Recall the two relations, *Enrollment* and *Course*, in Tables 1 and 2. The natural join of these relations is shown in Table 3. The tuples of *Enrollment* and *Course* are compared. Tuples that *match* (i.e. have the same join attribute value) are combined into a single result tuple. To be more concrete, the common join attribute of *Enrollment* and *Course* is Course Number. Because the Course Number of tuple E1 in *Enrollment* and C1 in *Course* is the same, E1 and C1 *match*. E1 and C1 are combined to produce tuple SC1 in *Student-Course*.

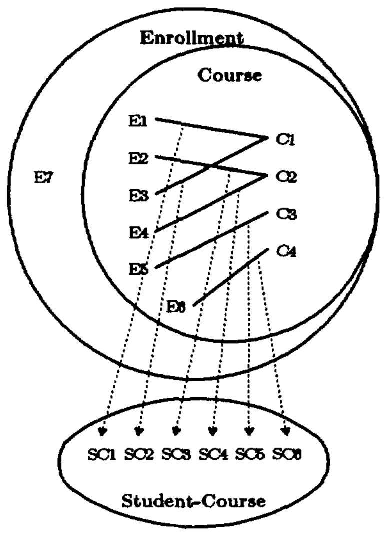

Figure 9: Natural Join of Enrollment and Course

Figure 9 shows a Venn diagram of the two relations. The intersecting region of the two relations consists of tuples from either relation that have a matching tuple in the other relation. So, E1 is in the intersecting region because it matches C1. Similarly, C1 matches both E1 and E3 so is also in the intersecting region. As the figure shows, each pair of matching tuples in the intersecting region are composed to produce a tuple in the result.

4.1.2. Semi-Join

The advantages of the semi-join operator were considered mainly for distributed databases [26]. An important consideration in distributed databases is the amount of data that must be sent over the network. Assume that the *Enrollment* relation is stored at Node 1 and the *Course* relation is stored at Node 2. Furthermore, the natural join of *Enrollment* and *Course* is needed at Node 2. Without using a semi-join, the strategy is to send the entire *Enrollment* relation to Node 2 and compute the natural join there.

If the *Enrollment* relation contains a large percentage of tuples that don't participate in the join, like tuple E7, a lot of unnecessary tuples will be transmitted. An alternate strategy is to project *Course* onto the join attribute as shown in Table 5. This relation is then shipped to

	Course Number
CN1	1
CN2	2
CN3	3
CN4	4

Table 5. Course Projected Onto Course Number

Node 1. Using the projected values, only the tuples of *Enrollment* that will participate in the join are chosen to send to Node 2. Semi-join is the operation that does this selection. Table 6 shows the semi-join of Table 1 and Table 5. The result of the semi-join is then sent to Node 2 where the natural join is computed.

Tuples are matched in semi-join the same way they are matched in natural join, when join attribute values are equal. The difference between natural and semi-join is the way that matching tuples produce a result. In natural join, the two matching tuples are composed into a single tuple. In semi-join, however, the result is the tuple from the first relation, the *Enrollment* tuple in the example. This is shown in Figure 10. Note the similarity of this figure and the previous figure.

4.1.3. Outer Join

Outer join [27], like semi-join, is a slight modification of natural join. Like natural join, pairs of matching tuples are composed to get a tuple in the result. However, tuples from the

Enrollment'

	Name	Course Number
E1	Adam	1
E2	Adam	2
E3	Betty	1
E4	Carol	2
E5	Denny	3
E6	Earl	4

Table 6. Semi-Join of Enrollment and Course Number.

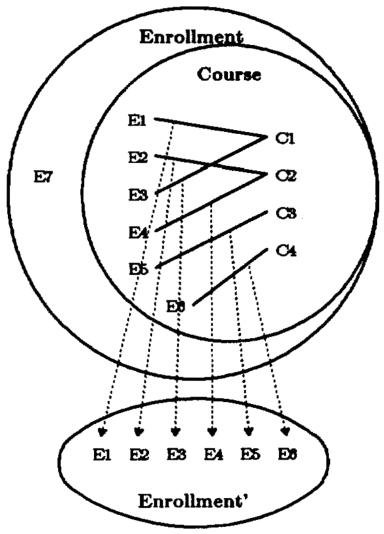

Figure 10: Semi-Join of Enrollment and Course Number

first relation that do not have a match in the second relation are not discarded in outer join. Instead, unmatched tuples are concatenated with an all NULL tuple. The outer join of *Enrollment* and *Course* is shown in Table 7. As Table 7 shows, tuple E7 of the *Enrollment* relation is unmatched but produces tuple SC7 in the result. This tuple has a NULL value for the Course Name attribute because E7 had no matching tuple in *Course*. Figure 11 shows the semi-join operation in a Venn diagram.

Outer join preserves all tuples of the first relation. This is useful in situations where all of the information from the first relation is needed to complete a query. For example, suppose

Student-Course'

	Enrollment.Name	Course Number	Course.Name
SC1	Adam	1	Data Structures
SC2	Adam	2	Algorithms
SC3	Betty	1	Data Structures
SC4	Carol	2	Algorithms
SC5	Denney	3	Architecture
SC6	Earl	4	Database
SC7	Frank	5	NULL

Table 7. Outer Join of Enrollment and Course.

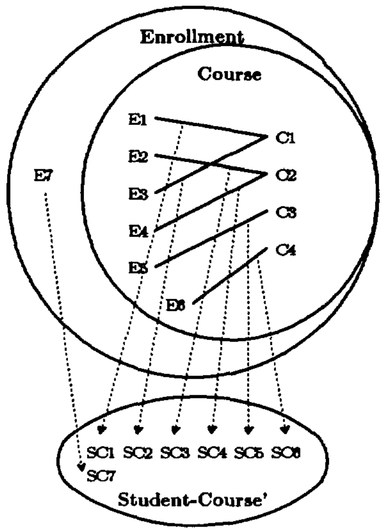

Figure 11: Outer Join of Enrollment and Course

that we need to get the number of valid courses taken by each student from the *Student-Course* relation in Table 3. Table 1 shows that Frank is enrolled for course number 5 which is invalid. However, the information that Frank is enrolled in zero valid courses cannot be determined from Table 3. The fact that Frank is a student not enrolled in any valid courses can be determined from the outer join shown in Figure 7.

4.1.4. Set Operations

Intersection will be used as an example of a set operation. Set operations are only meaningful for relations that are ***union compatible***. Union compatibility is the constraint that both relations have tuples with exactly the same attributes. Table 8 introduces a new relation, *Part-Time Enrollment*, that is union compatible with the *Enrollment* relation. The intersection

Part-Time Enrollment

	Name	Course Number
P1	Adam	1
P2	Adam	3
P3	Carol	2
P4	Gary	1

Table 8. Part-Time Enrollment Relation.

Common Enrollment

	Name	Course Number
E1=P1	Adam	1
E3=P3	Carol	2

Table 9. Common Enrollment Relation.

of these relations is given in Table 9.

As in natural join and semi-join, the result of intersection comes from the tuples that match. Unlike natural join, the result tuple is not a composition of the matching pair of tuples. In this respect, intersection is more like semi-join. The primary difference between intersection and semi-join is that matching tuples are exactly the same tuple. That is, the tuples have the same values for all attributes. So, in the example, tuple E1 and tuple P1 are the same tuple.

The Venn diagram for intersection is shown in Figure 12. Note that because matching tuples are the same tuple, relationally, it does not make sense to say that the result tuple comes from one relation or the other. In fact, the result can be said to come from both relations. It is only at the algorithm level that a distinction must be made.

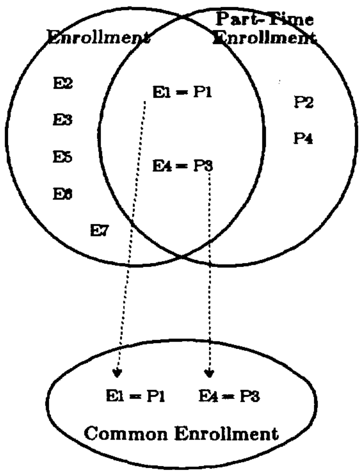

Figure 12: Intersection of Enrollment and Part-Time Enrollment

4.1.5. General Classification of Binary Operators

To capture the essence of one-to-one match we move away from operation consideration and view binary operations from the relation viewpoint. The general case of a binary operation on two relations, R and S, is shown in Figure 13. As in the examples, the tuples of both relations are separated into four groups. The groups are determined by matches between tuples of R and tuples of S. Note that it is not necessary at this point to define what *matches* means. The fours groups are:

(1) Tuples of R with no matching tuple in S
(2) Tuples of S with no matching tuple in R
(3) Tuples of R with one or more matching tuples in S and
(4) Tuples of S with one or more matching tuples in R

Given these four groups, each binary operation can be classified by the source of result tuples. Table 10 classifies join, semi-join, outer join, union, intersection and difference. Recall from the previous section that for intersection it makes no sense to distinguish tuples in Group 3 and Group 4. Also, marks in both Group 3 and Group 4 for non-set operations, join and outer join in Table 10, signifies that pairs of tuples that match are composed to produce a result.

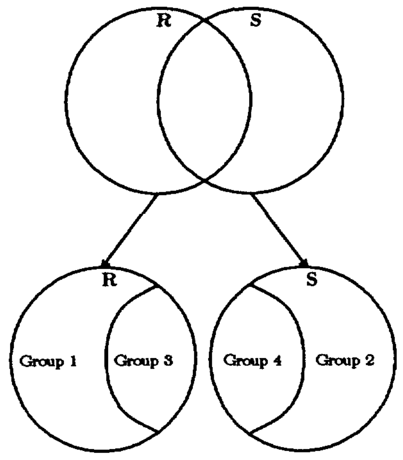

Figure 13: Classification of Tuples

Operation	GROUP 1	GROUP 2	GROUP 3	GROUP 4
Join			X	X
Outer Join	X		X	X
Semi-Join			X	
Intersection			X	X
Union	X	X	X	X
Difference	X			

Table 10. Classification of Operations by Source of Results.

From this classification it is apparent that there are an unlimited number of operations that can be performed. In this section there have been no restrictions placed on the definition of *match*, the way in which matching tuples are composed or how a tuple can be modified before being included in the result. Recently, a new operation has been proposed in [28] as a means of efficiently evaluating queries with universal quantifiers. The operator is called *complement join*[1]. Results of this operation come from Group 1 only. Looking back at Table 10 it looks similar to a set difference. In fact, it is identical to set difference except that the relations are not union compatible. This fact was also noted in [28].

4.1.6. Aggregate Functions and Duplicate Elimination

Aggregate functions and duplicate elimination are not binary operators. As such they cannot be described in terms of two relations and a nice Venn diagram. In spite of this, there is some similarity of aggregation and duplicate elimination to the operators examined above. Concentrating on aggregation for the moment, recall that it is a mathematical operation performed on groups of tuples within a relation. Once the groups of tuples have been determined, aggregation can be performed. This reduces aggregation to the finding of these groups. Grouping is based on equality of an attribute, the grouping attribute. The group that a tuple belongs to is found by the comparison of the tuple to a tuple representing the group. In other words, by comparing a single tuple to another tuple it is possible to determine the groups. This is the

[1] This operator bears a striking resemblance to an operator the author recently heard referred to as *natural-anti-semi-join.*

principle on which the one-to-one match operator is based.

4.2. Advantages of One-to-one Match

The progression of the examples above is meant not only to show the underlying principle of the operations, but also to show how a small change of one operator produces a new operator. Indeed, the similarity, we argue, points to the fact that all of these operations are but facets of a single operation.

Consider the advantages of adopting the one-to-one match operator instead of a number of modules able to perform individual operations. For example, suppose a database system has one module for computing join and another module for computing semi-join. Furthermore, assume both of the algorithms are hash-based. Replacing the two modules with the one-to-one match operator increases the functionality of the database system. That is, instead of being limited to join and semi-join, the database system can now perform outer join, intersection, difference, union, etc. In fact, it now has the capability of performing a class of binary relational operations.

The functionality of data items that one-to-one match operates on is encapsulated in *support functions*. Support functions are supplied by the query implementor (which is the query optimizer in a complete database system) and used by the one-to-one match operator in computing the query. No data specific code is contained in the implementation of the one-to-one match operator. The encapsulation of functionality on data items increases the extensibility of the one-to-one match operator. For instance, extending a database system to allow complex objects or new abstract data types does not affect the one-to-one match implementation. All knowledge of the new data type or structure is encapsulated in the support functions. Examples and explanation of support functions will be given during the description of the hash-based one-to-one match implementation in the next section.

There is a second way of looking at the interaction of the one-to-one match operator and its support functions. The support functions can be viewed as arguments to an *algorithm shell*.

The shell represents a class of operations. A particular relational operation is created by *mapping* a set of support functions using this algorithm shell. This interaction of Volcano operators and support functions is used in all iterators, but it is best demonstrated by the one-to-one match operator and the wide variety of operations it implements.

Because of the similarity between join and semi-join, the two algorithms look very much alike. The changes made in one module to optimize and tune the operation are also applicable to the other module. Also, a bug found in one module is likely to exist in the other module as well. In addition, porting the modules to another operating system or new hardware requires changes to both modules. All three of these changes or modifications; optimization/tuning, maintenance, and porting to a new platform; occur in one place with the one-to-one match operator. The only change in performance in going from the two modules to the one-to-one match module is the time spent, by one-to-one match, determining which operation to perform. Time spent determining which operation to perform is overshadowed by computing the operation and does impact performance.

More detailed examples of advantages of one-to-one match follow. Using the definition of semi-join as an example, it is relatively straightforward to modify a given natural join algorithm to make it compute semi-join. The same is true of the other operators. Now consider a database system that includes a classic hash join algorithm. Suppose also that small modifications have been done to the algorithm to obtain modules for computing semi-join, outer join and intersection. However, it is determined that the size of the relations in the system have grown too large to fit into main memory. Each module, four of them, must be modified to handle overflow. Adding overflow handling to the one-to-one match operator involves modifying a single section of code.

Suppose instead that each operation is still implemented within its own module but now each module shares code common to all operations. Because the common code is shared, adding overflow handling to the classic hash join algorithm must only be done in one place. Consider adding the union operation to this database system. A new module is created using the shared code used by the other modules. For each new operator to be implemented, a new module must be created. However, one-to-one match would only require the change of a few arguments without any code writing or modification. That is, one-to-one match is already capable of computing the operation, it is just a matter of using appropriate arguments.

4.3. Hash-Based One-to-One Match

We turn now to the hash-based implementation of one-to-one match. This implementation is based on *classic hash join*. Classic hash join is separated into two phases. The first phase, called the *build phase*, constructs a memory resident hash table with tuples from one relation. The relation used to fill the hash table is referred to as the *build relation* or *build input*. During the second phase, called the *probe phase*, tuples from the other relation are used to probe the hash table. This relation is called the *probe relation* or *probe input*. Our algorithm extends the classic hash join algorithm by adding a third phase, called the *flush phase*.

Only the logical phases of our algorithm have been mentioned. However, the algorithm is invoked as an iterator so the three phases are embedded in *open*, *next* and *close* calls. A call to *open* begins and completes the *build phase*. Both the *probe* and *flush* phases are accomplished through repeated calls to *next*. Before the first call to *next* neither the *probe* or *flush* phase has begun. At the point when all of the results have been returned, both the *probe* and *flush* phases have been completed.

4.3.1. Natural Join

At this point a detailed example would best serve to describe the details of the algorithm. We begin by describing how one-to-one match implements natural join. Recall that the first phase, the build phase, constructs a hash table with the build relation. The hash table consists of N bucket pointers. Each bucket contains an arbitrary number of tuples in the form of a linked list. This is illustrated in Figure 14. Each tuple of the build relation is inserted into this hash table. The bucket the tuple is inserted into is determined by the hash value of the tuple.

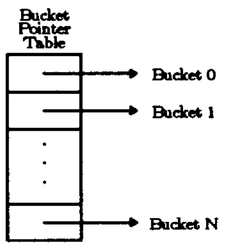

Figure 14: Hash Table

When the final build input tuple is inserted into the hash table the build phase ends. All of the build phase occurs during the *open* call.

During the probe phase, the hash table is probed for tuples matching those in the probe input. Probing occurs during a call to *next*. The first call to *next* retrieves the first tuple of the probe input. This tuple's hash value determines the hash table bucket where matching tuples can be found. If a match is found, the matching tuples are combined and this new tuple is returned. Note that a single probe input tuple may have more than one matching tuple in the hash table and cannot be discarded after finding a single match. Once all of the matching tuples have been found the current probe input tuple is discarded and the next probe input tuple is retrieved. The probe phase ends when the last probe input tuple is discarded.

Before returning an end-of-stream from the last *next* call, after the probe phase has ended, the memory consumed by the hash table is freed. Releasing the hash table memory occurs during the flush phase. More importantly, tuples in the hash table that were fixed in the buffer are unfixed. All of this happens during the last call to *next* and not during the call to *close*.

There are a number of support functions and arguments that are needed to perform the algorithm described above. The hash table size, N, is needed. Also, two hash functions are required. One hash function operates in tuples from the build input and one operates on tuples from the probe input, called *build hash* and *probe hash* respectively. Probing requires a function that compares a build input and a probe input tuple. This function, *compare*, returns either TRUE or FALSE. Finally, when a pair of matching tuples is found they must be combined by another function. The combining function, *compose*, fills a result tuple given the matching build and probe tuples.

4.3.2. Semi-Join

The semi-join implementation differs from the natural join implementation only in the way that matching tuples are used to produce a result. When computing natural join, a matching build and probe tuple are combined into a result tuple. Combining matching tuples in natural join produces a new tuple. In semi-join, however, matching tuples do not produce a new tuple. That is, no output tuples are created and no result file must be created and filled. Instead, the result a matching build and probe tuple is the probe input tuple. When performing semi-join, the one-to-one match operator behaves as a filter on the probe input. That is, one-to-one match only passes on those tuples of the probe input that would have participated in the join. Without the need to combine a build input and probe input tuple there is no no need for the *compose* support function.

4.3.3. Outer Join

To compute outer join, the entire probe phase must behave differently than the probe phase described for natural join. Matching tuples are still combined to make a result tuple but probe input tuples without a match are handled differently. Unmatched probe input tuples were discarded in natural join. In outer join, these unmatched tuples are combined with an all NULL tuple and returned as a result. Because these unmatched tuples produce a new result tuple, another support function is required to compute outer join. This function, *probe compose*, combines an unmatched probe input tuple with an all NULL tuple.

4.3.4. Aggregation

The one-to-one match operator is capable of performing an aggregate function on the build input. As in natural join, the hash table is constructed from the build input during the build phase. In order to describe aggregation the previous aggregation example will be referenced. After a tuple is retrieved from the build input it is hashed to a bucket. In natural join the bucket a tuple belongs in was based on the value of its join attribute. But, to perform aggregation, the bucket is based on the value of the grouping attribute. Therefore, tuples with the same grouping attribute value, Name in the example, get hashed to the same bucket.

Instead of inserting this tuple into the bucket, as natural join does, the bucket must be searched for a matching tuple. Again, tuples with the same grouping attribute value are matching tuples. When no match is found, the tuple is inserted and aggregation for that group of tuples is initialized. For example, tuple E1 in the *Enrollment* relation gets hashed to bucket K. There is no matching tuple in the bucket (i.e. no previous tuple belonging to the same group has been found) so it is inserted into the bucket. As it is inserted into the bucket, the count, representing the number of courses Adam is enrolled in, is initialized to one. If a matching tuple is found, the tuple being inserted is aggregated with the tuple already in the hash table. Now tuple E2 of *Enrollment* is hashed to the same bucket as E1, namely K. However, E2 matches tuple E1 because both names are Adam. Instead of inserting E2 into the bucket, the number of courses Adam is enrolled in is incremented by one.

When the entire build input has been inserted into the hash table, aggregation is finished. Note that this is all accomplished during the *open* call. Now the hash table contains the results of the aggregation. There is no probe input and no probing relation so the probe phase accomplishes nothing during aggregation. Instead, the flush phase begins on the first *next* call. Each call to *next* returns one of the results contained in the hash table. When computing aggregation, the flush phase is responsible for more than just freeing the hash table memory. In this case it flushes results from the hash table while freeing hash table memory.

As with natural join, aggregate functions require a number of support functions. While a tuple is being inserted it must be compared to tuples already in the hash table. A function, called *build compare,* compares a tuple being inserted into the hash table with tuples already in the hash table. If no match is found, aggregation is initialized by invoking an *initialize* function on the tuple after inserting it into the hash table. Tuples that match must be aggregated with an ***aggregate*** function. Unlike natural join, aggregation does not require a probe input, a ***probe hash*** function, a ***compare*** function or a ***compose*** function.

If an aggregate function or duplicate elimination is required on the build input to one-to-one match, the output records typically have a different format than the input records. Therefore, a new file is created for such output records, typically on a virtual device. Virtual devices are a construct that allows allocating and manipulating temporary space in memory in the same way as disk-resident files, but virtual devices never require I/O.

4.3.5. Aggregation with Natural Join

Instead of just producing the result of the aggregation as described in the previous section, suppose the results are to be used in a subsequent join. For example, now that the number of courses taken by each student has been determined we may want to know each student's year in school. Table 11 shows the *Class* relation and Table 12 shows the result of joining Table 4 and Table 11.

Class

	Name	Class
CS1	Adam	Freshman
CS2	Betty	Freshman
CS3	Carol	Sophomore
CS4	Denny	Sophomore
CS5	Earl	Junior
CS6	Frank	Senior

Table 11. Class Relation

	Name	Number of Courses	Class
CLC1	Adam	2	Freshman
CLC2	Betty	1	Freshman
CLC3	Carol	1	Sophomore
CLC4	Denny	1	Sophomore
CLC5	Earl	1	Junior
CLC6	Frank	1	Senior

Table 12. Aggregation Followed by Join

Aggregation is performed first, just as described in the previous section. However, the probe input is not empty now. On the first call to *next*, the first tuple of *Class* is retrieved and the probe phase begins. From this point on the algorithm performs exactly as described for natural join.

The previous explanation may be used to show the advantage of including aggregation and duplicate elimination in the one-to-one match operator. One-to-one match was born from the observation that aggregation of a relation is often followed by a join. More importantly, the grouping attribute of the aggregation is also the join attribute of the join. Consider what would happen if the aggregate and join operations were separated. First, a hash table is used for performing the aggregation of the *Enrollment* relation. When the aggregation is complete, each aggregate tuple is removed from the hash table and sent to the join operation. The join operation takes each tuple resulting from the aggregation and inserts it into a new hash table. In fact, because the join attribute is the same as the grouping attribute, the new hash table will be equivalent to the previous hash table. By combining the aggregate and join operations, the hash table is only constructed a single time.

4.3.6. Intersection

As already observed, intersection is the equivalent of a natural join of union compatible relations. However, the implementation of intersection resembles the implementation of semi-join. Recall that semi-join differed from natural join because it did not create new result tuples, instead, the result of a matching build and probe tuple was the probe tuple. Because the matching build and probe tuple are exactly the same in intersection, it does not matter which tuple is returned as the result. In order to prevent having to check for intersection, the implementation of semi-join is used to implement intersection.

4.3.7. Difference

The implementation of difference varies from that of the previously described implementations during the probe and flush phases. When *next* is first called the probe phase begins. A tuple from the probe input is used to probe the hash table. Instead of producing a result when a matching tuple is found, the build input tuple, in the hash table, is marked. Marked tuples in the hash table are those that also exist in the probe input. The probing phase finishes during the first call to *next* because probing does not produce any output tuples.

After the probe phase completes, the hash table still contains all of the tuples of the build input. However, some of the tuples are marked and some are unmarked. The result of the difference are those tuples in the hash table that are unmarked (i.e. no matching tuple exists in the probe input). During the flush phase, unmarked tuples are returned while the hash table is being emptied.

Difference still requires *build hash, probe hash* and *compare* support functions but does not need *compose*. Notice that the support functions needed for difference are exactly the same as those needed for semi-join. From the standpoint of support functions the two algorithms are indistinguishable. For this reason it is necessary to include another argument to one-to-one match to differentiate between semi-join and difference.

4.3.8. Other Operators

As mentioned early in the paper, it is advantageous to build the hash table with smaller of the two relations. This may conflict with the way the implementation has been described so far. For example, suppose relation R has 10,000 tuples and relation S has only 100 tuples. The desired operation is the semi-join of R and S (i.e. those tuples of relation S that participate in the join of R and S). semi-join, as described above, requires that the hash table be built with relation R and probed with relation S.

We observed that with the capability of selecting tuples in the hash table and the addition of the flush phase it should be possible to perform the semi-join the other way around (i.e. build with relation S and probe with relation R). In fact, one-to-one match has the capability of always building the hash table with the smaller relation regardless of operator requirements. First, the hash table is built with tuples from relation S. The probe phase behaves similar to the difference implementation described above. That is, the hash table is probed with tuples from relation R but instead of producing results while probing, tuples in the hash table are marked. After the probe phase has completed the flush phase empties the hash table returning those tuples that have been marked. Recall that marked tuples are those with one or more matching tuples in the probe input.

4.3.9. Operator Summary

The combination of marking tuples in the hash table and using the flush phase to remove result tuples from the hash table makes it possible to return tuples from any group shown in Figure 13. Furthermore, by associating a function with each of the groups, tuples from each domain can be transformed before being returned (i.e. combining a tuple from Group 1 with an all NULL tuple when computing outer join).

One-to-one match determines which operation to perform from the user supplied support functions. Table 13 lists the operations that have been discussed and the support functions that are needed for these operations. Note again that some operations, such as semi-join and difference, require another argument to be distinguishable. Also, operations called "Reverse" are

Operator	Build Compare	Initialize/ Aggregate	Compose	Probe Compose	Build Compose	Op
Join	no	no	yes	no	no	none
Aggregate	yes	yes	no	no	no	none
Join and Aggregate	yes	yes	yes	no	no	none
Outer Join	no	no	yes	yes	no	none
Semi-Join	no	no	no	no	no	none
Reverse Outer Join	no	no	yes	no	yes	none
Reverse Semi-Join	no	no	no	no	no	*semi*
Union	no	no	no	no	no	*union*
Difference	no	no	no	no	no	*difference*
Intersection	no	no	no	no	no	none

Table 13. Argument Determination of Operation.

those described in the section on other operators.

4.4. Hash Table Overflow

While hash tables in main memory are usually quite fast, a severe problem occurs if the hash table does not fit in main memory. This situation is called ***hash table overflow.*** There are two ways to deal with hash table overflow. First, if a query optimizer is used and can anticipate overflow, overflow can be avoided, typically by partitioning the input(s). Such ***overflow avoidance*** techniques are the basis for the hash join algorithm used in the Grace database machine [14]. Second, overflow files can be used to resolve the problem after it occurs. Several ***overflow resolution*** schemes have been designed and compared [10,11,29,16]. At the current time, we are studying how best to implement hash table overflow avoidance and resolution for the rather complex one-to-one match operator in Volcano.

Almost all techniques to deal with hash table overflow use several temporary files called ***overflow files.*** The number of bucket files can be quite large, and is limited only by the buffer memory needed to hold clusters being filled with records. Depending on the scheme, the records of selected or all hash buckets of the hash table to be built are written into bucket files. Using several bucket files allows partitioning, the main reason for and advantage of hash based algorithms.

Gerber also considered two schemes that do no use partitioned overflow files [11,29]. Both schemes use multiple passes and assume that in each pass, the hash buckets to be kept in memory can be determined a priori. First, re-reading the input multiple times and extracting the currently needed records in each pass clearly works if the build input is a stored relation; if the build input is produced by another operation, it is written when it is first received, and can then be scanned repeatedly. Both variants require the same number of I/O operations. Second, it might be tempting to write a new file each time some records are extracted, thus limiting the number of record to be read in subsequent passes. It turns out, however, that the additional write operations exactly counter-balance the savings in read operations. Therefore, all three schemes and variants discussed in this paragraph have the same I/O complexity. If the total build input size is a multiple of the memory size, say N times the memory size, the build input must be read N times. Thus, the I/O complexity of all these algorithms is N^2, rather undesirable for large inputs and not competitive with sort based algorithms for very large files.

More efficient schemes fan the overflow into multiple files simultaneously, thus providing for reading selected records efficiently. Typically, records are assigned to these *partition files* using the same or a modified hash function that is used for the in-memory hash table. In fact, in an overflow resolution scheme, it might be advisable to decide dynamically whether or not to use the same hash function, depending on how uniformly it distributed input records over the hash buckets.

Multiple overflow files carry both advantages and disadvantages. The premier advantage is that in most cases, each record has to be written and read only once. The exception occurs if the number of files necessary to ensure that each file will fit into memory exceeds the number of output buffers available during the partitioning phase. In this case, *recursive* or *multi-level* partitioning is required, in which each partition is partitioned again until all partition files fit into memory. It is interesting to note that this is exactly the situation in which multiple merge levels are required for sorting a large file or relation. A careful comparative analysis shows that the I/O behaviors of partitioning and merging are the same, including the fractions of random and sequential I/O, except that the directions of the data streams are reversed.

A second advantage is that multiple overflow files allow *bucket tuning*, i.e., grouping overflow files such that each group will fit into memory. However, the advantage of bucket tuning is not entirely clear since it seems that the same performance could be achieved simply by processing one bucket at a time.

A chief disadvantage of multiple overflow files is that they have to be written using random I/O which is much more expensive on moving-head disks than sequential I/O. In fact, whether or not random I/O is necessary can turn the superior performance of hash-based algorithms such as hybrid hash join [10,9] into inferior performance when compared to carefully tuned sort-based algorithms, at least in certain parameter ranges [30].

It is interesting to note that if an aggregate function or duplicate elimination is performed on the build input, only the *output* of this operation must fit in main memory. In particular when memory is scarce, aggregating into a new, temporary file pays off since it avoids internal fragmentation in the buffer, i.e., the records are packed densely into clusters.

For Volcano's one-to-one match operator, we are considering three overflow resolution schemes. First, we consider a refinement of hybrid hash join, suitably modified for aggregation processing and tuned for fast reaction to overflow to allow continued dataflow from the build input operator. Second, since Volcano is meant to run on a shared-memory machine or a group of homogeneous shared-memory machines, we will explore *memory trading* between partitions. Third, we intend to experiment with *data compression.*

If an aggregate function or duplicate elimination is required on the build input of one-to-one match, the output records typically have a different format than the input records, and a new file is created for such output records. Instead of using a single such file, we create multiple files. These files will become overflow files if necessary, but only if necessary. The latter condition distinguishes our scheme from Grace's overflow avoidance scheme. If overflow occurs, one of these files is selected to be dumped to disk. Clusters (pages) of this file can be written very fast because records are already assembled into suitable pages, thus no copying occurs at this point. This distinguishes our algorithm from standard hybrid hash as implemented in GAMMA [16]. Thus, our algorithm combines hybrid hash's flexibility to handle overflow as it occurs with Grace's overflow avoidance technique. We believe that this combination will also result in the best dataflow behavior, i.e., one-to-one match's input operator will have to be stopped for the least amount of time while hash table overflow is being resolved.

If an operator is execute by multiple processes, i.e., the input is partitioned into multiple disjoint subsets, there is a good chance that the load is not entirely balanced, and only a subset of the partitions experience hash table overflow. If all these processes run on one shared-memory machine, ***memory trading*** can be used to avoid overflow alltogether. Of course, this will require at least some amount of synchronization between these processes, which may impede overall performance. However, since all processes work with one shared buffer, a counting semaphore is probably sufficient to control total memory demand and determine when overflow resolution is required.

Finally, ***data compression*** can be used to reduce space requirements. As CPUs' performance improvements proceed at a faster rate than disk transfer rate improvements [31], compressing data in the hash table might be become a very attractive technique to resolve hash table overflow, in particular if the final hash table size (without compression) is only slightly larger or a small multiple of total memory size. Recent compression techniques, originally developed chiefly for networking applications, can yield significant compression rates, particularly for text data.

5. Performance

From the beginning, we focussed our attention on the performance of the Volcano system, in particular the tradeoff of generality and extensibility vs. performance. This section presents preliminary performance results of the one-to-one match operator.

5.1. Benchmarks

The relations used for measuring performance are those used in the Wisconsin Benchmark [32]. Each tuple is 208 bytes in length and is composed of 13 4-byte integer fields and 3 52-character (byte) strings. An example tuple is shown in Table 14. The attribute names signify the values the attribute can have. ***Unique1*** and ***unique2*** have unique integers ranging from zero

unique1	unique2	two	four	..	ten thousand	..	stringu1	stringu2	string4
6546	10	1	3	..	9999	..	Ax..A..xA	Gx..xZ	Ox..xO

Table 14. Example Wisconsin Benchmark Tuple

to the size of the relation (i.e. 0-99999 for a 10,000 tuple relation). Remaining integer fields have integers in the range signified by the attribute name. Thus, the *ten thousand* attribute contains integers in the range 0-9999. Both *stringu1* and *stringu2* are unique strings. Each string has three varying character positions with x's separating them. The final attribute, *string4*, is confined to four specific strings. In order to limit the results from these performance runs, we restricted the operations to join and aggregation.

5.2. Hardware

The queries were run on a dedicated Tektronix 4316 workstation (Motorola 68020 CPU) with 4 MBytes of RAM running under the UTek 3.1 operating system. Two Tektronix 4495 Mass Storage Units with an 86 MByte CDC WREN III hard disk drives were connected to the workstation through a SCSI port. The SCSI bus interface transfer rate is 1.25 MBytes per

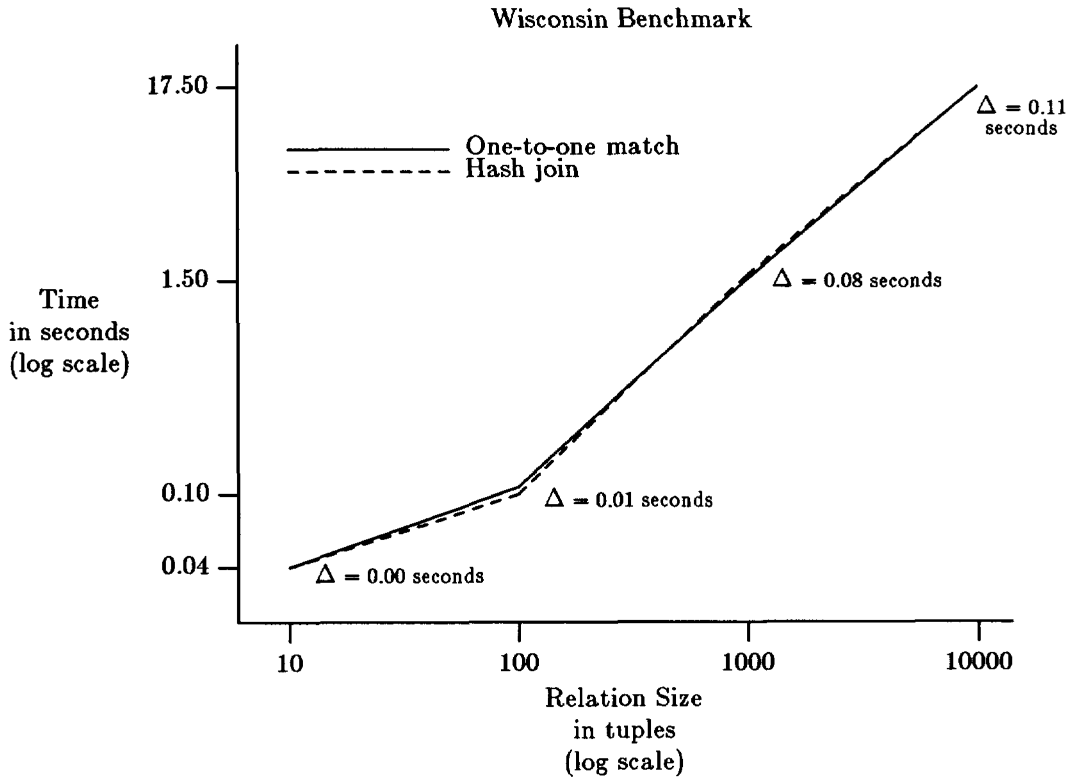

Figure 15: One-to-one match and Hash join Comparison

second but at the time of printing the performance specifications of the hard drive were unavailable. All relations, both input and output, were stored in operating system files.

5.3. One-to-One Match versus Classical Hash Join

To begin, one-to-one match was used to join relations of varying sizes. Relation sizes varied from 10 tuples to 10000 tuples with both inputs being the same size. Each join produced a result relation with the same number of tuples as the relations that were joined. For example, two 10,000 tuple relations are joined to produce a 10,000 tuple relation. Results are measured in elapsed time to complete the join, in seconds, and are shown in Figure 15. Also plotted in the figure is the performance of a hash-based join algorithm. The hash-based join algorithm, labeled hash join, is a classical hash join algorithm. Notice that using one-to-one match to compute join instead of a dedicated hash-based join algorithm does not reduce performance. Any time that the one-to-one match operator needs to determine which operation to perform is negligible compared to the time required to read and write the relations.

5.4. The Affect of Join Selectivity on One-to-One Match

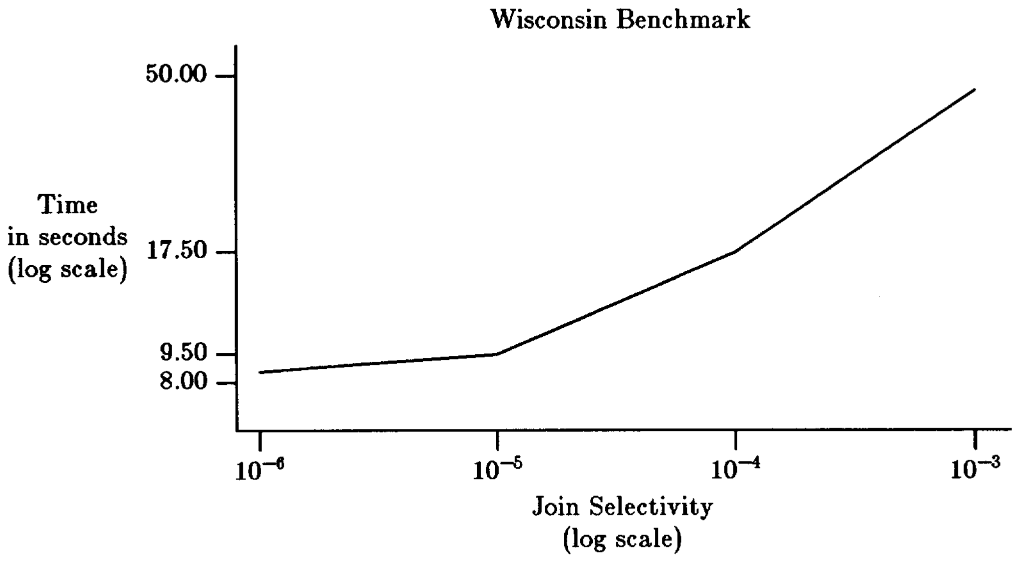

Figure 16: One-to-One Match With Variable Join Selectivities

To show the impact of result size on performance, we restricted the input relation sizes to 10,000 tuples and varied the join selectivity. The join selectivity is the ratio of the size of the result to the largest result size possible. For example, if relation R is of size M and relation S is of size N and the join of R and S results in a relation of size K then the join selectivity is $\frac{K}{MN}$. Therefore, a join selectivity of 10^{-3} when joining two 10,000 tuple relations will produce a 100,000 tuple result:

$$\frac{100{,}000}{10{,}000^2} = \frac{10^5}{10^8} = 10^{-3}.$$

Figure 16 shows the results. The first interesting feature of the graph is the drastic change in slope at a join selectivity of 10^{-5}. This is the point at which the size of the result relation begins to approach and then dominate the size of the input relations. For example, a join selectivity of 10^{-5} results in a relation containing 1,000 tuples. But note that the result tuples are almost twice as long as the input tuples. So, with a join selectivity of 10^{-5},

$$20{,}000 \times 208 \; bytes = 4{,}160{,}000 \; bytes$$

are read and

$$1{,}000 \times 412 \; bytes = 412{,}000 \; bytes$$

are written. As the size of the result gets larger, the cost of reading the inputs becomes less significant. Thus, the time it takes to complete the join becomes proportional to the size of the result. This can be seen in the graph as the curve becomes linear after a join selectivity of 10^{-5}.

5.5. One-to-One Match versus Aggregation+Join

To show the advantage of including aggregation in one-to-one match, we performed an aggregation followed by a join on the grouping attribute. In the first group of tests, the aggregation operation and join operation were computed separately. Remember that this requires the hash table to be built twice. The second group of runs were computed by the one-to-one match operator which combines aggregation and join into a single operation. The comparison

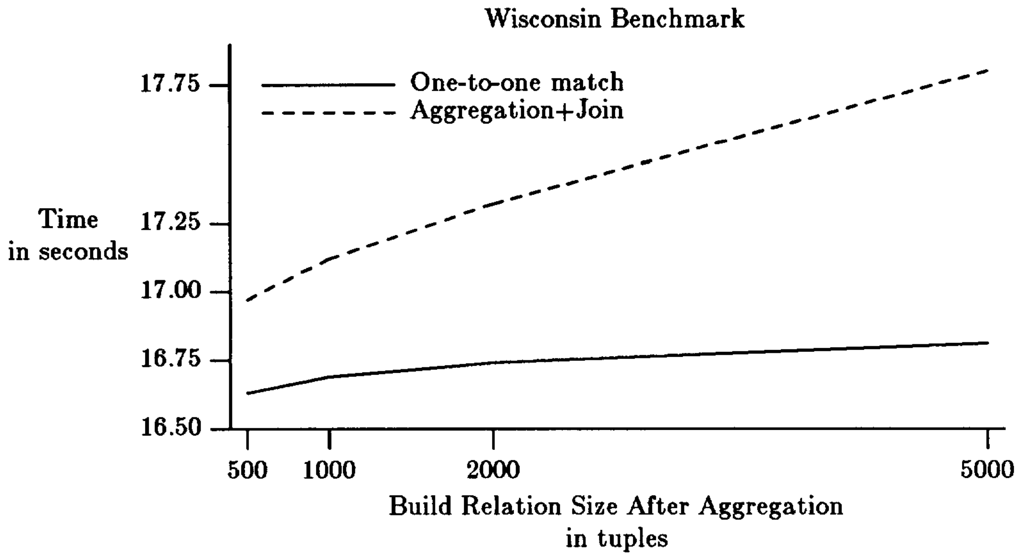

Figure 17: One-to-One Match and Aggregation+Join Comparison

of the two methods is shown in Figure 17. The most important feature of the graph is the difference between the two curves. Both methods require the same amount of time to compute the aggregation and to compute the join. However, the aggregation+join algorithm must empty the hash table of aggregation results, communicate those results to the join operation and build a new hash table with the aggregation results. As the size of the aggregation result increases so does the cost of destroying and rebuilding the hash table. In the graph this is seen as the increasing distance between the two curves as the aggregation result size increases.

6. Future Work

The one-to-one match operator is currently based on classic hash join. This means that no provision has been made for hash table overflow. It is not clear at this point if it is feasible to handle hash table overflow as described in Section 2 or whether another means of handling overflow will be required.

It was mentioned previously that the use of support functions to encapsulate functionality on data items leads to greater extensibility. The Volcano query evaluation system is being developed with this goal in mind. In fact, Volcano will be one of the components of the Revelation system currently being researched.

Currently, the hash-based implementation of one-to-one match contains specific code dealing with creation and destruction of the hash table as well as the insertion and removal of tuples from the hash table. By removing index dependent code, the implementation of one-to-one match may be further generalized. Manipulation of the index, a hash table in the hash-based implementation, could be made through index implementation independent calls (i.e. insert (item), remove (item), etc.). This generalization could allow replacing the index implementation without affecting the one-to-one match algorithm. For example, instead of using a hash table for finding matches, an in memory B-Tree could be used. The benefits of making the one-to-one match algorithm independent of indexing may be outweighed by the costs, particularly in a parallel environment.

7. Summary

We have presented the one-to-one match operator, an operator in the Volcano query evaluation system. It is capable of performing a class of binary relational operators including natural join, semi-join, outer join, anti join, union, intersection, difference, anti-difference and Cartesian product. All of these operations are based on a single principle. The results of the operation are determined by comparing a single tuple of one relation to a single tuple of the other relation. In addition, aggregation and duplicate elimination can also be performed independent of or in conjunction with the operations listed above.

We have discussed the hash-based implementation of one-to-one match. Because a single module is used to perform a number of binary relational operators, as opposed to using a separate module for each operator, the code is easier to maintain, debug, optimize, tune, and port to new hardware and software platforms. The operation that one-to-one match performs is determined by a number support functions, supplied by the query implementor or the query optimizer. By allowing the support functions to be determined outside of the algorithm the flexibility of one-to-one match is increased. In addition, support functions encapsulate functionality on data items. All these advantages are obtained while maintaining the performance achieved in a module-per-operator database system.

8. Acknowledgments

We thank Len Shapiro for asking the questions requiring the most thought. We also thank him for providing a basis for describing one-to-one match.

References

1. E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* **13**(6) pp. 377-387 (June 1970).
2. E.F. Codd, "Relational Completeness of Database Sublanguages," pp. 65-98 in *Data Base Systems,* ed. R. Rustin,Prentice-Hall, New York (1972).
3. D. Maier, *The Theory of Relational Databases,* Computer Science Press, Rockville, MD. (1983).
4. A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," *Journal of the ACM* **29**(3) pp. 699-717 (July 1982).
5. M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson, "System R: A Relational Approach to Database Management," *ACM Transactions on Database Systems* **1**(2) pp. 97-137 (June 1976).
6. M. Stonebraker, E. Wong, P. Kreps, and G.D. Held, "The Design and Implementation of INGRES," *ACM Transactions on Database Systems* **1**(3) pp. 189-222 (September 1976).
7. M. Blasgen and K. Eswaran, "Storage and Access in Relational Databases," *IBM Systems Journal* **16**(4)(1977).
8. D. Knuth, *The Art of Computer Programming,* Addison-Wesley, Reading, MA. (1973).
9. L.D. Shapiro, "Join Processing in Database Systems with Large Main Memories," *ACM Transactions on Database Systems* **11**(3) pp. 239-264 (September 1986).
10. D.J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems," *Proceedings of the ACM SIGMOD Conference,* pp. 1-8 (June 1984).
11. D.J. DeWitt and R.H. Gerber, "Multiprocessor Hash-Based Join Algorithms," *Proceedings of the Conference on Very Large Data Bases,* pp. 151-164 (August 1985).
12. M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of Hash to Data Base Machine and Its Architecture," *New Generation Computing* **1**(1) pp. 63-74 (1983).
13. M. Kitsuregawa et al., "Architecture and performance of relational algebra machine GRACE," *Proc. Int. Conf. Parallel Processing,* pp. 241-250 (1984).
14. S. Fushimi, M. Kitsuregawa, and H. Tanaka, "An Overview of The System Software of A Parallel Relational Database Machine GRACE," *Proceeding of the Conference on Very Large Data Bases,* pp. 209-219 (August 1986).
15. R. Gerber, "Dataflow Query Processing using Multiprocessor Hash-Partitioned Algorithms," *Ph.D. Thesis,* University of Wisconsin, (October 1986).
16. D. DeWitt and D. Schneider, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment," *Proceedings of the ACM SIGMOD Conference,* p. 110 (May-June 1989).

17. J.P. Richardson, H. Lu, and K. Mikkilineni, "Design and Evaluation of Parallel Pipelined Join Algorithms," *Proceedings of the ACM SIGMOD Conference,* pp. 399-409 (May 1987).

18. E. Omiecinski and E. Tien, "A Hash-Based Algorithm for a Cube-Connected Parallel Computer," *Information Processing Letters* **30**(5) pp. 269-275 (March 1989).

19. R. Epstein, "Techniques for Processing of Aggregates in Relational Database Systems," *UCB/ERL Memorandum,* (M79/8)University of California, (February 1979).

20. A. Klug, Statistical Query Facility"" "Investigating Access Paths for Aggregates using the "Abe" Statistical Query Facility," *IEEE Database Engineering* **5**(3)(September 1982).

21. G. Graefe, "Volcano: An Extensible and Parallel Dataflow Query Processing System," *Oregon Graduate Center, Computer Science Technical Report,* (89-006)(June 1989).

22. M. Conway, "A Multiprocessor System Design," *Proceedings of the AFIPS Fall Joint Computer Conference,* pp. 139-146 (1963).

23. G. Graefe, "Relational Division: Four Algorithms and Their Performance," *Proceedings of the IEEE Conference on Data Engineering,* pp. 94-101 (February 1989).

24. G. Graefe, "Parallel External Sorting in Volcano," *Oregon Graduate Center, Computer Science Technical Report,* (89-008)(June 1989).

25. G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," *Oregon Graduate Center, Computer Science Technical Report,* (89-007)(June 1989).

26. P.A. Bernstein, N. Goodman, E. Wong, C. Reeve, and J.B. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Transactions on Database Systems* **6**(4)(December 1981).

27. C.J. Date, "The Outer Join," *Proceedings of the Second International Conference on Databases,* (September 1983).

28. F. Bry, "Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited," *Proceedings of the ACM SIGMOD Conference,* p. 193 (May-June 1989).

29. R.H. Gerber, "The Hash-Partitioned Algorithms," *Preliminary Proposal,* University of Wisconsin, (January 1985).

30. G. Graefe, "Heap-Filter Merge Join: A New Algorithm for Joining Medium-Size Relations," *Oregon Graduate Center, Computer Science Technical Report,* (89-012)(June 1989).

31. H. Boral and D.J. DeWitt, "Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines," *Proceeding of the International Workshop on Database Machines,* Springer, (1983).

32. D. Bitton, D.J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach," *Proceeding of the Conference on Very Large Data Bases,* pp. 8-19 (October-November 1983).