A Superjoin Algorithm for Deductive Databases

James A. Thom, Kotagiri Ramamohanarao, Lee Naish

University of Melbourne, Parkville, Victoria 3052, AUSTRALIA

Abstract

This paper describes a join algorithm suitable for deductive and relational databases which are accessed by computers with large main memories. Using multi-key hashing and appropriate buffering, joins can be performed on very large relations more efficiently than with existing methods. Furthermore, this algorithm fits naturally into top-down Prolog computations and can be made very flexible by incorporating additional Prolog features.

Keywords: Partial match retrieval, Prolog, hashing, joins, optimization, database, relational, deductive

1. INTRODUCTION

The join operator is both a frequently used relational operator and an expensive one in relational database systems. Several join algorithms have been discussed in the literature; for example, nested-loops, sort-merge, and hash-join [Bratbergsengen 84], [DeWitt 84], [Jarke 84], and [Ullman 82].

In this paper we propose another join algorithm, the *superjoin* algorithm. This algorithm, based on multi-key hashing, partitions the join to enable efficient buffer management. The superjoin algorithm is suitable for large relational databases accessed from computers with large main memories. One of the properties of the superjoin algorithm is that it maintains excellent performance from very small relations to very large relations. The superjoin algorithm also supports queries containing arbitrary constraints, disjunctions, negations, and existential quantifiers. These additional properties are especially useful for deductive database systems using top-down computation.

Section two provides a background to the superjoin algorithm. It describes the notation used in this paper, multi-key hashing and partial match joins. These are illustrated using some examples.

Section three describes how, using multi-key hashing and appropriate partitioning, joins can be performed on very large relations in a very efficient way. If there is suitable buffering of pages then each page of each relation needs to be read into main memory at most once. An algorithm is described which minimizes the number of buffers required. An implementation of this algorithm, in Prolog, is given in Appendix A and some examples are listed in Appendix B.

Section four describes how the superjoin fits naturally into Prolog's top-down computation, including Prolog features such as negation and existential quantifiers. A complete deductive database system would include a higher level which would break the query up into sub-queries to be processed efficiently by the superjoin. The superjoin would access a low-level interface to the external database.

Section five analyses the performance of the superjoin algorithm, and compares it with other well known join algorithms.

The conclusion, presented in Section six, is that the superjoin provides a flexible and powerful database "primitive", and provides superior efficiency to that of existing join algorithms.

2. BACKGROUND

2.1. Notation

Throughout this paper we shall use a notation based on Prolog [Clocksin 81]. The term *predicate* is used as equivalent to the term *relation*, and *fact* is used as equivalent to *tuple*.

Predicate (relation) names are written in lower case, strings are written with double quotes, and variables are written in upper case without any quotes. A fact (tuple) can be written

p("x1", "y1")

which is a fact about "x1" and "y1" from the predicate 'p'. A *query* can be written

?- p(X, "y1")

which would find all values for the first attribute (the variable X) for those facts in 'p' where the second attribute is "y1". Conjunctives are constructed with ',' and disjunctives with ';'. Thus, the query

?- p(X,Y), q(Y,Z) QUERY 1

would join 'p' with 'q', joining on the second attribute of 'p' and the first attribute of 'q' (the *join variable* Y).

2.2. Multi-key hashing

There are various schemes for accessing predicates stored using multi-key hashing. For example, hashing with descriptors [Ramamohanarao 83], recursive linear hashing [Ramamohanarao 85a], hashing with superimposed codeword indexing [Ramamohanarao 85b]. Other methods are described in [Aho 79], [Lloyd 80], and [Rivest 76]. Many schemes are variations and enhancements on a common scheme which we shall describe here.

Consider a partial match query on a single predicate. We assume a predicate is stored on disc in the pages of one file. Each tuple has a fixed number of attributes, denoted $a_1, a_2, \ldots, a_k$. A *partial match query* is a specification of the value of zero or more of these k attributes. The answer to a query is the collection of all tuples in the file which have the specified values for the specified attributes.

Tuples are allocated to pages within the file by means of a hashing scheme which allocates zero or more bits to each attribute of the tuple. Consider a static file consisting of 2^d pages, where $d \geq 0$. The pages are numbered from 0 to 2^d-1. Each attribute a_i has a hashing function h_i which maps from the key space of a_i to a bit string. From this string d_i bits are selected, such that $d_1+\ldots+d_k = d$. By computing the hash values of all the attribute values of a tuple it is possible to compute the page number in which the tuple is stored. This requires the use of a *choice vector*, which defines a mapping between the strings of d_i bits and the page number. The choice vector was introduced for handling dynamic files in [Lloyd 82] and [Ramamohanarao 83]. Here we use a slightly simplified notation for choice vectors.

Suppose the hash functions h_i map "y0" to the string of bits "...000" and "y1" to "...001" and "y2" to "...010" and similarly for "z0", "z1", "z2" *et cetera*. For each attribute, we choose the last d_i bits to construct the choice vector.

Consider the predicate q(Y,Z), then the hash functions h_1 and h_2 would map the value Y to the string "$\ldots Y_3Y_2Y_1$" and the value Z to the string "$\ldots Z_3Z_2Z_1$" respectively, where Y_i and Z_i can take values 0 or 1. We loosely refer to these strings as bit strings. If q(Y,Z) is stored in 2^3 pages, then we might set $d_1 = 2$ and $d_2 = 1$. Thus the bits Y_2Y_1 and Z_1 would be selected from the bits strings generated by the hash functions. The choice vector could be any one permutation of these bits strings, such as $Y_1Z_1Y_2$. Thus tuples would be stored in the file as shown in the 'q' predicate in Figure 1.

p

page	tuples
X_1Y_1	p(X,Y)
0 0	p("x0", "y0"), p("x2", "y2")
0 1	p("x0", "y1"), p("x2", "y1")
1 0	p("x1", "y0")
1 1	p("x1", "y1")

q

page	tuples
$Y_1Z_1Y_2$	q(Y,Z)
0 0 0	q("y4", "z2"), q("y4", "z2")
0 0 1	q("y2", "z2")
0 1 0	q("y4", "z1")
0 1 1	q("y2", "z1")
1 0 0	q("y1", "z2")
1 0 1	q("y3", "z2")
1 1 0	q("y1", "z1")
1 1 1	q("y3", "z1"), q("y7", "z1")

r

page	tuples
X_1Z_1	r(X,Z)
0 0	r("x0", "z0"), r("x2", "z2")
0 1	r("x0", "z1"), r("x2", "z1")
1 0	r("x1", "z0")
1 1	r("x1", "z1")

s

page	tuples
$X_1Y_1Y_2$	s(X,Y)
0 0 0	s("x0", "y0"), s("x2", "y4")
0 0 1	s("x0", "y0")
0 1 0	s("x0", "y1"), s("x2", "y1")
0 1 1	s("x0", "y3")
1 0 0	s("x1", "y0")
1 0 1	s("x1", "y2")
1 1 0	s("x1", "y1")
1 1 1	s("x1", "y3")

Figure 1: 'p', 'q', 'r' & 's' predicates

For a query containing variables, only some of the bits will be specified. Thus several pages will need to be searched. Consider the query:

?- q("y3", Z) QUERY 2

Answering this query would require the following steps:

(1) Hash "y3" to the bit string "...011" from which we select just the last two bits since $d_1 = 2$, namely $Y_2Y_1 = 11$

(2) Since the second attribute is a variable, the bit $Z_1 = *$, where * represents an unknown bit

(3) Search the pages where $Y_1Z_1Y_2 = 1*1$ (namely pages 101 and 111) for tuples which match the query; this would retrieve the tuples:

q("y3", "z2")
q("y3", "z1")

2.3. Partial match joins

It is possible to apply these partial match schemes directly, for instance, in a Prolog system getting answers a tuple at a time ([Naish 83], [Ramamohanarao 85b]). Consider query 1 again.

?- p(X,Y), q(Y,Z)

In answering the query, a straightforward implementation in Prolog would get the first solution from the first page of the 'p' relation, namely:

p("x0", "y0")

then try to find facts from the second predicate which match. This would require reading page 00 from 'p' and, since "y0" hashes to "...00", only pages 000 and 010 from 'q'. However, there are no matching tuples, so Prolog would proceed with the second solution from the first page of 'p' in a similar fashion. Thus, the second solution

p("x2", "y2")

would join with the tuples

q("y2", "z2")
q("y2", "z1")

This would require reading additional pages 001 and 011 from 'q' since "y2" hashes to "...10". After exhausting all the tuples in the first page of 'p' Prolog would proceed in a similar fashion with subsequent pages of 'p'.

This scheme for implementing joins fits in naturally with other relational operators such as selection. Consider the query

?- p("x0",Y), q(Y,Z) QUERY 3

This query performs a selection on 'p' while performing the join, reducing the number of pages which need to be considered.

Although using partial match retrieval to implement joins creates a workable system, there are some important inefficiencies. For most queries this approach requires that many pages need to be read more than once.

3. THE SUPERJOIN ALGORITHM

Multiple reading of pages from disc in the naive tuple at a time algorithm can be eliminated by keeping pages in main memory buffers. In query 1 above, if we use twelve buffers (four for the 'p' predicate and eight for the 'q' predicate) then no page needs to be read from disc more than once. This simple method, of buffering all pages of both predicates, is not feasible with large predicates. Since main memory size is limited it is necessary to reduce the number of buffers required.

If we use multi-key hashing and the join attributes have the same hash function, then it is possible to partition the join into a number of *sub-joins*. Each sub-join accesses only some of the tuples required for the full join. The result of the full join is simply the union of the results of all the sub-joins. The superjoin algorithm partitions a join and orders the execution of the sub-joins so as to minimize the number of buffers required, while still only reading the pages of each predicate from disc at most once.

3.1. Partitioning into sub-joins

The simplest partitioning of query 1 has each page of the 'p' predicate in a separate partition. This results in four sub-joins, where each sub-join only considers those tuples with a common hash value for Y_1 and X_1. Each partition is a join of one page of the 'p' predicate with four pages of the 'q' predicate. If we read the pages of 'p' sequentially this requires nine buffers (one for the 'p' predicate and eight for the 'q' predicate).

By changing the order in which we access the pages of the 'p' predicate we can reduce the number of buffers needed. Suppose, instead of accessing tuples from 'p' in the order the pages are numbered (00, 01, 10, 11), we read the pages in the order 00, 10, 01, 11. That is, we read all the pages in 'p' such that Y_1 is 0 before reading the pages in 'p' in which Y_1 is 1. While we are joining tuples from the pages 00 and 10 in 'p' the tuples will all hash to $Y_1 = 0$ thus matching tuples will only appear in the pages 000, 001, 010, 011 from 'q'. Later when we join tuples from the pages 01 and 11 in 'p' the tuples will all hash to $Y_1 = 1$ and matching tuples will only appear in the pages 100, 101, 110, 111 from 'q'. When this is done the number of buffers required is reduced to only five (one for the 'p' predicate and four for the 'q' predicate).

It is possible to further reduce the number of buffers required by a more complex partitioning. In the optimal partitioning, each partition consists of tuples with common hash values for Y_1, Y_2 and Z_1. This partitions the join into eight sub-joins. Each sub-join consists of all tuples in one page of the 'q' predicate and, on average, half the tuples in two pages of the 'p' predicate. For example, the sub-join where $Y_1Z_1Y_2 = 001$, consists of all tuples in page 001 of 'q' and the tuples in pages 00 and 10 of 'p' with hash value such that $Y_2 = 1$. If the sub-joins are done in increasing values of $Y_1Z_1Y_2$ then the number of buffers required is three (two for 'p' and one for 'q'). The same is true with increasing values of $Y_1Y_2Z_1$, but with no other permutation (see section 3.3).

3.2. Sfb-vector

Our use of bit strings, such as $Y_1Z_1Y_2$, in the previous section can be formalized by defining an *sfb-vector* (slow-fast bits vector). The bits in the sfb-vector define the partitioning and the order of the bits defines the order in which the sub-joins are done (compare with the choice vector which specifies the order of hashed page addresses in which tuples are stored).

Given an sfb-vector of length f, a choice vector and a number in the range 0 to 2^f-1 a set of pages in the file is identified. For example, if the sfb-vector was $Y_1Y_2Z_1$ and the choice vector was X_1Y_1 then the number 101 would identify the pages *1 (that is 01 and 11).

The superjoin generates a sequence of numbers from 0 to 2^f-1 and uses the sfb-vector to access the corresponding pages of the file. Tuples are retrieved from each predicate in such a way that all matching tuples which hash to a particular value of the sfb-vector (an *sfb-value*) are retrieved together. These tuples need only be joined with tuples in the other predicates which match the same sfb-value.

3.3. Superjoin execution

We now describe the execution mechanism in more detail, including the management of buffers. Associated with each predicate call is a pool of buffers. Consider a predicate with a choice vector of length c, say. Let the sfb-vector be $B_1,...,B_e,...,B_f$ where $B_1,...,B_e$ are all in the choice vector and, if e is less than f, B_{e+1} is not. That is, the most significant e bits of the sfb-vector are used for indexing the predicate. These are called the *slow bits* for this predicate.

During execution, different partial match queries are made to the predicate for each partition of the join. In answering these queries, we only consider tuples whose hash values match both the query and the entire current sfb-value. That is, only those matching tuples which are in the current partition are retrieved. If pages on disc need to be accessed, new buffers are allocated on demand. These pages (in the buffers) may be accessed several more times while the current values of the slow bits remain the same. However, when these bits change, the pages are never accessed again, since the sub-joins are done in order of increasing sfb-value. Therefore, the whole buffer pool for that call can be deallocated.

During execution of the superjoin, the addresses of the pages in the buffers always match the current value of the slow bits. Thus there is a maximum of 2^{c-e} buffers in the buffer pool of the relation at any one time. The number of buffer pages for an n-way join is the sum of those needed for each relation:

$$\text{nbufs} = \sum_{i=1}^{n} 2^{\text{free}_i}$$

where

$$\text{free}_i = c_i - e_i$$

c_i is the choice vector length of the ith predicate and e_i is the number of slow bits for the ith predicate.

The sfb-vectors should be constructed so that this cost function is minimized.

3.4. An algorithm to construct sfb-vectors

A greedy algorithm (implemented in Prolog) to construct sfb-vectors is given in Appendix A. We will illustrate the algorithm with the following example.

?- q(Y,Z), r(X,Z), s(X,Y) QUERY 4

With a null sfb-vector the cost (number of buffer pages) for the join would be $2^3+2^2+2^3=20$ pages.

The bits for the sfb-vector can come from any of X, Y, or Z. If we chose X_1 for the first bit this would reduce the cost to $2^3+2^1+2^2=14$ pages. Similarly, if we chose Z_1 this would reduce the cost to $2^2+2^1+2^3=14$ pages. However, if we chose Y_1 the cost would only be $2^2+2^2+2^2=12$ pages. The greedy algorithm chooses Y_1 as the first bit of the sfb-vector since that results in the greatest reduction in the number of buffers pages required.

The next bit to be chosen would again be from Y since the sfb-vector "Y_1Y_2" would mean the join only required $2^1+2^2+2^1=8$ pages. With the sfb-vector "Y_1X_1" we need $2^2+2^2+2^1=10$ pages, and with "Y_1Z_1" we need $2^1+2^2+2^2=10$ pages. Note that after choosing a Y bit, we cannot reduce the number of buffers required for the 'r' predicate, since the number of slow bits of 'r' has been fixed at zero.

The final bit to be chosen will be either X_1 or Z_1 since both the sfb-vectors "$Y_1Y_2X_1$" and "$Y_1Y_2Z_1$" would reduce the number of buffers required to $2^1+2^2+2^0=7$ pages and $2^0+2^2+2^1=7$ pages respectively. Any more X, Y or Z bits will not further reduce the number of buffer pages required.

Some larger examples are given in Appendix B. One property of the algorithm is that the number of buffers needed will be one for at least one of the predicates. It also produces an optimal sfb-vector for two way joins. However, for multi-way joins, the algorithm may produce sub-optimal solutions. We are currently investigating other algorithms for constructing sfb-vectors. Other methods for defining partitions and orderings are also being considered.

Even using the minimal number of buffers, it is possible that there will be insufficient main memory for the superjoin algorithm. This may be because there is only a small amount of main memory available or because the relations themselves are so huge. In such cases it is possible to resort to a secondary partitioning of the problem using the method described in [Bratbergsengen 84].

4. SUPERJOIN AND DEDUCTIVE DATABASES

We now show how the superjoin algorithm is ideally suited to Prolog-style deductive databases. It can be implemented with a few quite simple system predicates and can be made very flexible by incorporating more of the features of Prolog. With these additions the superjoin can be applied to almost any query, including those containing ordinary Prolog predicates as well as external database predicates. We outline a possible implementation on top of MU-Prolog [Naish 85c].

4.1. Implementation in Prolog

The method of evaluating joins in Prolog, tuple at a time using backtracking, is very similar to the superjoin algorithm. Two additional features are needed. The first is the use of buffer pools. The implementation uses the primitive pool_open(Pool) to create an empty buffer pool and pool_close(Pool) to deallocate and close a pool of buffers. Secondly, we need the ability to call a predicate using a buffer pool and only return answers matching the sfb-value. For this, we use the primitive db_call(Call, Pool, Sfb), which allocates new buffers on demand. Sfb is a data structure specifying the current sfb-value and the mapping between the sfb-vector and the choice vector.

A conjunction of calls to database relations in Prolog is translated into a conjunction of calls to db_call, plus a call to db_superjoin. For example,

```
?- p(X,Y), q(Y,Z), r(X,Z)                    QUERY 5
```

is translated into

```
?- db_superjoin(...), db_call(p(X,Y), PPool, PSfb),
     db_call(q(Y,Z), QPool, QSfb),
     db_call(r(X,Z), RPool, RSfb)
```

The call to db_superjoin calculates PSfb, QSfb and RSfb using the current sfb-value, choice vectors, and sfb-vector, and allocates and deallocates buffer pools. It returns one answer for each sfb-value, in increasing order. The set of answers to the query is found by Prolog's normal backtracking mechanism.

4.2. Additional Prolog features

Since the join is implemented as a Prolog conjunction, arbitrary constraints can be implemented, by simply adding extra calls. For example, if Y was an integer rather than a string, the constraint $Y < 100$ could be inserted after p(X,Y). Calls to procedures using the full power of Prolog's recursion and nondeterminism can be used. The superjoin algorithm can ignore all extra calls which do not bind any variables. Calls which generate bindings for variables can also be handled, by adding tests to check the hash values match the current sfb-value. There is also considerable flexibility in how calls to database relations are arranged. The method of partitioning works for disjunctions as well as conjunctions. The superjoin algorithm can ignore the connective.

It is also possible to incorporate negation and an if-then-else construct. Prolog uses the negation as failure rule [Clark 78], which is a weaker form of the closed world assumption [Reiter 78]. Use of the partitioning scheme described in this paper and only negating ground calls ensures soundness of the negation as failure rule. Any ground call must match the current sfb-value, so the relevant facts in the relation are all considered. However, the use of some other partitioning schemes or negating non-ground calls can result in unsoundness.

Considerably more power can be provided, at very little cost, by allowing quantifiers [Naish 85b]. Negations with quantified variables, for example $\forall X \neg p(X,Y)$, are implemented by calling the predicate with the quantified (local) variables uninstantiated. Soundness is still guaranteed providing these local variables do not appear in the sfb-vector. The insertion of existential quantifiers (also known as decomposition [Wong 76] and isolating independent sub-queries [Warren 81]) is particularly useful for optimization. For example in query 5, if only the values of X and Y are needed, Z can be existentially quantified:

```
?- p(X,Y), ∃Z (q(Y,Z), r(X,Z))
```

The effect of the existential quantifier is that if one value of Z is found for particular values of X and Y, then no more are sought. Subsequent backtracking skips over the calls to 'q' and 'r' entirely, potentially saving much computation and disc reading. If Z is included in the sfb-vector the computation is sound, but for particular values of X and Y more than one value of Z may be found. Using the scheme we have outlined so far, it is most efficient to exclude the local variable from the sfb-vector (possibly increasing the number of buffers needed).

Another important higher level optimization is the creation of temporary predicates. This can be implemented by allowing Prolog's assert and retract primitives in superjoins. To avoid reading and writing pages more than once, they are also translated into primitives which use buffer pools: db_assert(Fact, Pool, Sfb) and db_retract(Fact, Pool, Sfb). The only type of goal we must avoid is one which modifies some of the relations we are reading. Even when Prolog does not include the superjoin the result of such goals is not well defined.

In summary, superjoins can be applied to any reasonable Prolog goal containing calls to database predicates. This great flexibility enables many higher level optimizations.

5. ANALYSIS

In this section we analyse the performance of the superjoin with two relations 'p' and 'q'. We assume there is only one join attribute. We use the following parameters in the analysis:

P	= size of relation 'p' in pages
Q	= size of relation 'q' in pages
kp	= number of attributes in relation 'p'
kq	= number of attributes in relation 'q'
np	= number of tuples in relation 'p'
nq	= number of tuples in relation 'q'
p_n	= number of bits allocated for the nth attribute of relation 'p'
q_n	= number of bits allocated for the nth attribute of relation 'q'
ti	= average insertion cost/tuple in the buffer pool
ts	= average search cost/tuple

Storage requirements:

The maximum number of buffers required is

$$nbufs = 1+2^{\min(pp,qq)}$$

where

$$pp = (\sum_{n=1}^{kp} p_n)-m$$

$$qq = (\sum_{n=1}^{kq} q_n)-m$$

$$m = \min(p_i,q_j)$$

where i and j are the join attributes of 'p' and 'q' respectively

Input/Output cost:

Assuming sufficient buffers are available and ignoring any overhead of the operating system, the number of disc pages read during the superjoin is

$$\text{disc reads} \leq P+Q \text{ pages}$$

CPU cost:

We need to build some form of indexing in main memory for each buffer pool for efficiency. The CPU cost of the superjoin operation would then be

$$\text{cpu cost} = (np+nq).ti+np.ts$$

However we can reduce the join cost by having further indexing, such as superimposed coding, on the disc files. This eliminates the need for additional indexing in main memory and CPU cost is then reduced to

$$\text{cpu cost} = np.ts$$

Values for ts and ti depend on the indexing scheme used, size of the relations and the distribution of join attribute values; that is the selectivity factor. To reduce the CPU cost it is worth considering re-ordering of predicates, for example see [Warren 81] and [Naish 85a].

5.1. Comparison with other join algorithms

The superjoin algorithm does not require the expensive sorting phase of the sort-merge join algorithm and the physical partitioning of the hash-join algorithm. Hence the scheme is very efficient with respect to input/output and CPU time requirements. Another feature of the scheme is that it has at least the efficiency of the nested-loop join algorithm. When one of the relations is small other algorithms can be much less efficient.

One drawback with the scheme is that it may require a large number of buffer pages. However, for binary relations, if we give equal number of bits to both attributes the number of buffers required will be

$$\sqrt{\min(P,Q)}$$

In comparison, the hybrid-hash join algorithm of [DeWitt 84] requires

$$\sqrt{\max(P,Q)}$$

5.2. Interaction with host operating system

When one develops a database system on top an existing file system provided by the host operating system (such as Unix) it is important to control the way the pages are accessed from the files. For example, accessing pages sequentially will be more efficient than a random access. In Unix, for small files both access methods require only one page access. However, accessing pages at random from a larger file (containing up to 2^{20} pages of size 4096 bytes) requires on average two disc reads, whereas a sequential access requires one disc read.

Although the superjoin does not guarantee that the disc pages will be read in sequential order (the optimal order from the point of view of the file system), pages in the disc file will be read in ascending order in one or more passes of the file, which is almost as good. This quasi-sequential access is achieved because the slow bits of the sfb-vector tend to come from the more significant bits of the choice vector.

6. CONCLUSIONS

Research into deductive database systems based on partial match retrieval motivated the development of the superjoin algorithm. Partitioning joins using the superjoin method described in this paper will also be very useful in relational database systems.

Some of the features of the superjoin algorithm which make it particularly attractive when sufficient memory is available are:

- each relation is read at most only once;
- the algorithm outperforms any other algorithm;
- it fits naturally into Prolog's top-down computation; and
- arbitrary constraints, negation and quantifiers can easily be incorporated.

Further work needs to be done to integrate the superjoin with other optimizations. Two areas are choice vector determination and predicate reordering, extending the work of [Ramamohanarao 83], [Warren 81] and [Naish 85a].

ACKNOWLEDGEMENT

This work was performed as part of the Machine Intelligence Project at the University of Melbourne. It was supported by the Commonwealth Department of Science and, during 1985, by Pyramid Technology, Australia.

We would like to thank Justin Zobel and other members of the Machine Intelligence Project for their careful reading of earlier drafts of this paper.

APPENDIX A

```
%   usage:   ?- sfb([v,...],[[x,...],...],[[n,...],...],Sfb,Cost).
%
%       [v,...] is list of variables in expression
%       [[x,...],...] is list of variables in each predicate
%       [[n,...],...] is list of sizes of attributes in
%               choice vectors in each predicate
%       Sfb is the resulting sfb-vector
%       Cost is the resulting number of buffer pages

%   Calculate sfb-vector for superjoin (greedy algorithm)
%
sfb(Vs,Preds,Sizes,Choice.Sfb,TotCost):-
    choose(Vs,Preds,NewPs,Sizes,NewSizes,Choice,FixedC,FreeC),
    length(NewPs,L),
    (if FreeC > L then

        sfb(Vs,NewPs,NewSizes,Sfb,Cost),
        TotCost is FixedC + Cost
      else
        Sfb = [],
        TotCost is FixedC + FreeC
    ).
```

```
%
% choose next bit for sfb-vector from list V.Vs so that cost is
%   mininmized when that variable is used for partitioning
%
choose(V.[],Preds,NewPreds,Sizes,NewSizes,V,FixedC,FreeC):-
    partition(V,Preds,NewPreds,Sizes,NewSizes,FixedC,FreeC).
choose(V.Vs,Preds,NewPreds,Sizes,NewSizes,Choice,FixedC,FreeC):-
    choose(Vs,Preds,XPreds,Sizes,XSizes,X,XFixedC,XFreeC),
    Xcost is XFixedC + XFreeC,
    partition(V,Preds,VPreds,Sizes,VSizes,VFixedC,VFreeC),
    Vcost is VFixedC + VFreeC,
    (if Vcost < Xcost; Vcost = Xcost,VFixedC =< XFixedC then
        Choice = V,
        NewPreds = VPreds,
        NewSizes = VSizes,
        FixedC = VFixedC,
        FreeC = VFreeC
    else
        Choice = X,
        NewPreds = XPreds,
        NewSizes = XSizes,
        FixedC = XFixedC,
        FreeC = XFreeC
    ).

%
% partition join using V
%
partition(V,[],[],[],[],0,0).
partition(V,P.Preds,NewPreds,S.Sizes,NewSizes,FixedC,FreeC):-
    partition(V,Preds,NewPs,Sizes,NewSs,Fixed,Free),
    ( pred_partition(V,P,S,PartitionSize) ->
        NewPreds = P.NewPs,
        NewSizes = PartitionSize.NewSs,
        FixedC = Fixed,
        sum(PartitionSize,Sum),
        FreeC is Free + 1 << Sum
    ;
        /* no arguments of P match with V so make P fixed
           and remove from further partitioning */
        NewPreds = NewPs,
        NewSizes = NewSs,
        sum(S,Sum),
        FixedC is Fixed + 1 << Sum,
        FreeC = Free
    ).

%
% partition a predicate using V
%
pred_partition(V,Arg.Args,Size.Sizes,NewSize.NewSizes):-
    V == Arg,
    /* this argument matches */
    Size > 0,
    NewSize is Size - 1,
    ( pred_partition(V,Args,Sizes,NewSizes1) ->
        /* other arguments of this predicate also match */
        NewSizes = NewSizes1
    ;
        /* no other arguments of this predicate match */
        NewSizes = Sizes
    ).
pred_partition(V,Arg.Args,Size.Sizes,Size.NewSizes):-
    (
        V \== Arg
    ;
        Size = 0
    ),
    /* this argument does not match - see if any other matches */
    pred_partition(V,Args,Sizes,NewSizes).

%
%   Sum a list of integers
%
sum([],0).
sum(L.List,Sum):-
    sum(List,S),
    Sum is S + L.
```

APPENDIX B

Examples of sfb-vector generation.

(1)

```
Preds = [[x, y], [y, z]]      % ?- p(X,Y), q(Y,Z)
Bits = [[1, 1], [2, 1]]       % bits for each attribute
Sfb = [y, y, z]               % sfb-vector Y1Y2Z1
Nbufs = 3                     % number of buffers
```

(2)

```
Preds = [[x, y], [y, z], [x, y]]
Bits = [[4, 4], [5, 3], [2, 3]]
Sfb = [y, y, y, y, x, x, x, x]
Nbufs = 21
```

(3)

```
Preds = [[a, b], [b, c], [a, b, c]]
Bits = [[3, 2], [3, 2], [2, 2, 3]]
Sfb = [b, b, a, a, c, c, c]
Nbufs = 11
```

(4)

```
Preds = [[b, d, a], [d, c], [a, b, c], [a, b, c]]
Bits = [[2, 1, 2], [5, 2], [3, 0, 3], [3, 0, 3]]
Sfb = [c, c, a, a, a, c]
Nbufs = 66
```

(5)

```
Preds = [[y, z], [x, y], [z, z1]]
Bits = [[7, 5], [3, 6], [4, 7]]
Sfb = [z, z, z, z, y, y, y, y, y, y, y, z]
Nbufs = 641
```

(6)

```
Preds = [[x, y, z], [y, z1], [y, z, z1]]
Bits = [[7, 4, 3], [4, 7], [6, 4, 4]]
Sfb = [y, y, y, y, z, z, z, x, x, x, x, x, x, x]
Nbufs = 257
```

(7)

```
Preds = [[x, y, x], [y, x, z]]      % note repeated variable
Bits = [[3, 1, 4], [2, 3, 5]]
Sfb = [x, x, x, y, y, z, z, z, z, z]
Nbufs = 3
```

REFERENCES

[Aho 79]
A. V. Aho and J. D. Ullman, "Optimal partial-match retrieval when fields are independently specified", *ACM Transactions on Database Systems 4*, 2 (June 1979), 168-179.

[Bratbergsengen 84]
K. Bratbergsengen, "Hashing Methods and Relational Algebra Operations", *Proceedings of the Tenth International Conference on Very Large Data Bases*, Singapore, August 1984, 323-333.

[Clark 78]
K. L. Clark, "Negation as Failure", in *Logic and Databases*, H. Gallaire and J. Minker (editor), Plenum Press, 1978.

[Clocksin 81]
W. F. Clocksin and C. S. Mellish, *Programming in PROLOG*, Springer-Verlag, Berlin, 1981.

[DeWitt 84]
D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker and D. Wood, "Implementation Techniques for Main Memory Database Systems", *SIGMOD Conference on the Management of Data*, 1984, 1-8.

[Jarke 84]
M. Jarke and J. Koch, "Query Optimization in Database Systems", *ACM Computing Surveys 16*, 2 (June 1984), 111-152.

[Lloyd 80]
J. W. Lloyd, "Optimal Partial-match Retrieval", *Bit 20* (1980), 406-413.

[Lloyd 82]
J. W. Lloyd and K. Ramamohanarao, "Partial-match Retrieval for Dynamic Files", *Bit 22* (1982), 150-168.

[Naish 83]
L. Naish and J. A. Thom, "The MU-Prolog Deductive Database", Technical Report 83/10, Department of Computer Science, University of Melbourne, November 1983.

[Naish 85a]
L. Naish, "Negation and Control in Prolog", Technical Report 85/12, Department of Computer Science, University of Melbourne, 1985. Ph.D. thesis.

[Naish 85b]
L. Naish, "Negation and Quantifiers in NU-Prolog", Technical Report 85/13, Department of Computer Science, University of Melbourne, October 1985.

[Naish 85c]
L. Naish, "The MU-Prolog 3.2 Reference Manual", Technical Report 85/11, Department of Computer Science, University of Melbourne, October 1985.

[Ramamohanarao 83]
K. Ramamohanarao, J. W. Lloyd and J. A. Thom, "Partial Match Retrieval Using Hashing and Descriptors", *ACM Transactions on Database Systems 8* (1983), 553-576.

[Ramamohanarao 85a]
K. Ramamohanarao and R. Sacks-Davis, "Partial Match Retrieval Using Recursive Linear Hashing", *Bit 25* (1985), 477-484.

[Ramamohanarao 85b]
K. Ramamohanarao and J. A. Shepherd, "A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases", Technical Report 85/17, Department of Computer Science, University of Melbourne, November 1985.

[Reiter 78]
R. Reiter, "On Closed World Databases", in *Logic and Databases*, H. Gallaire and J. Minker (editor), Plenum Press, 1978, 55-76.

[Rivest 76]
R. L. Rivest, "Partial match retrieval algorithms", *SIAM Journal of Computing 5*, 1 (1976), 19-50.

[Ullman 82]
J. D. Ullman, *Principles of Database Systems*, Pitman, London, 1982. Second Edition.

[Warren 81]
D. H. D. Warren, "Efficient Processing of Interactive Relational Database Queries Expressed in Logic", *Proceedings of the Seventh International Conference on Very Large Data Bases*, Cannes, France, 1981, 272-281.

[Wong 76]
E. Wong and K. Youssefi, "Decomposition - a strategy for query processing", *ACM Transactions on Database Systems 1*, 3 (1976), 223-241.