Reduction filters for minimizing data transfers in distributed query optimization

J.M. Morrissey

School of Computer Science
University of Windsor
Windsor, Ontario
Canada N9B 3P4

Introduction It has long been recognized that query optimization in distributed database systems is an important research issue. The challenge is to determine a sequence of operations which will process the query while minimizing the chosen cost function. Approaches include the use of semijoins [3, 4, 2, 5, 8, 11, 6, 12], joins [1, 10] or a combination of both operations [7]. Finding the optimal optimization for a general query is an NP-hard problem so, in general, heuristics are employed to find a cost-effective and efficient processing method.

In this paper we present a novel approach to the problem, which uses ***reduction filters***, with the objective of minimizing the total volume of data transferred in the network. We assume a distributed relational database management system and select-project-join queries[1]. This means that we have a number of relations, each located at a different site in the network, which must be joined and the result made available at some distinct query site. Our technique is to reduce the relations, before shipment to the query site, using reduction filters and thereby significantly decrease the total communication cost.

Reduction filters A reduction filter is simply an array of bits that is used as a compact representation of the values of an attribute in a relation. It is constructed using basic hashing techniques [9] as follows:

1. Initialize all the bits in the filter to zero.
2. For each value of the attribute, use a hash function to produce an address.
3. For each address produced, set the corresponding bit to one.

The reduction filters are used in a new operation, called a ***quasi-join***. It works by filtering out tuples which cannot belong to the result. A quasi-join, from relation R_1 to relation R_2 over the join attribute d_{i1}[2] is executed as follows:

- Construct a reduction filter for attribute d_{11} of R_1. This reduction filter is denoted by $h(d_{11})$.
- Ship the filter to the site of R_2.

[1] This is not a restriction as all relational queries can be expressed in this format.

[2] Notation: d_{ij} denotes attribute j of relation i.

0-7803-3143-5 /96/$4.00

☐ For each tuple in R_2:

- Hash on the attribute value d_{21} to produce an address.
- If this address is set in the reduction filter then retain this tuple.

As R_2 is processed a new filter, representing the values in both R_1 and R_2, is constructed. This can be used to reduce R_1.

In the more general case where we have a number of relations with a common join-attribute we can construct a very powerful reduction filter by intersecting the filters from each relation. For example, we have four relations, R_1 to R_4, with the common join-attribute d_{i1}. The filters $h(d_{11})$, $h(d_{21})$, $h(d_{31})$ and $h(d_{41})$ are constructed and shipped to some convenient site, most probably the site of one of the relations. The filters are combined — using an "and" operation — to produce the reduction filter $h(d_{i1*})$. The filter $h(d_{i1*})$ is important because it represents the intersection of all the attribute values. If a tuple of a relation in the query has an attribute value which is not in the intersection then it can't be part of the result and need not be shipped. In this way, $h(d_{i1*})$ is extremely good at filtering out tuples from relations. These are tuples which need not be shipped to the query site and thus they represent a saving in communication costs.

In most hashing applications collisions[3] are a problem that must be avoided or handled in some consistent manner. That is not the case here. The only consequence of collisions is that a few extra tuples, which are not part of the result, will be shipped to the query site. They will not lead to an incorrect answer.

[3] Here, a collision occurs when two values hash to the same address in the bit array.

We have developed a heuristic which uses quasi-joins to process general queries involving any number of relations and join attributes[4]. Local processing costs are kept to a minimum by using and building all possible filters each time a relation is processed.

The quasi-join heuristic We assume a query that consists of a number of relations which must be joined over several join-attributes. The relations are R_1 to R_m; the join attributes are d_{i1} to d_{im}.

Step 1: For each join attribute d_{ij}, construct the reduction filter $h(d_{ij*})$ by intersecting the filters for each relation with that attribute.

Step 2: Order the relations by the number of join-attributes they contain, from largest to smallest.

Step 3: The forward reduction phase: reduce each relation, in turn in the above order, using each applicable reduction filter, and construct the updated filters. The details of this step are as follows:

☐ Ship each reduction filter required to the site of the relation.

☐ Execute the quasi-joins concurrently: that is, for each tuple of the relation

- hash on each join-attribute value
- check if the bit is set in the corresponding reduction filter
- if every bit is set then keep this tuple as part of the answer.

As each tuple is processed, we construct an updated reduction filter for each join-attribute in the relation, using the method outlined previously.

[4] Note that the treatment of a general query is much more complex than the case we presented above involving just one join attribute.

The newly updated filters are then shipped on to the next site, where the next relation will be processed.

Step 4: The reverse reduction phase: reduce each relation in the reverse order, updating filters as necessary.

Step 5: The reduced relations are shipped to the query site for final processing.

An example A simple example will illustrate how the heuristic works. We have four relations, R_1 to R_4, with four join-attributes, d_{i1} to d_{i4}, as shown below. R_1 has join-attribute d_{i1}, R_2

	d_{i1}	d_{i2}	d_{i3}	d_{i4}
R_1	x			
R_2	x			x
R_3	x	x	x	
R_4	x	x	x	x

has join-attributes d_{i1} and d_{i4}, and so on. We construct the $h(d_{i1*})$ reduction filter by intersecting the $h(d_{11})$, $h(d_{21})$, $h(d_{31})$ and $h(d_{41})$ filters. The $h(d_{i2*})$, $h(d_{i3*})$ and $h(d_{i4*})$ reduction filters are constructed in a similar way. The forward reductions can now begin: R_4 is reduced using all four reduction filters; R_3 is reduced using the newly updated filters $h(d_{i1*})$, $h(d_{i2*})$ and $h(d_{i3*})$; R_2 is reduced using $h(d_{i1*})$ and $h(d_{i4*})$; finally R_1 is reduced using $h(d_{i1*})$. The reverse reductions, using only those reduction filters which have been updated, are now performed: R_2 is reduced using $h(d_{i1*})$; R_3 is reduced using $h(d_{i1*})$ (the reduction filters $h(d_{i2*})$ and $h(d_{i3*})$ will not have been modified so they are not used); and finally R_4 is again reduced using all four reduction filters.

Performance Study A performance study can provide a pragmatic assessment of the performance of an algorithm, a direct comparison between algorithms and also valuable insight into the problem at hand. Therefore, we are evaluating our heuristic, comparing its performance with that of more traditional query optimization algorithms such as those described in [2]. We have constructed an evaluation framework wherein we can systematically generate "synthetic" data and queries, produce optimization schedules, execute them and compare performance. It is both necessary and expedient to work with synthesized data since with "real" data it is difficult to guarantee a large enough set of disparate queries so that the results of the study are truly significant and not just due to chance or pathological cases.

Our objective is to investigate the performance of our heuristic with a wide range of select-project-join queries. For our purposes, a query consists of a number of relations, each located at a different site, which must be joined and the result made available at the query site. A query is constructed by first using a random number generator to determine parameters such as the number of participating relations, their respective sizes, the number of join attributes and their respective sizes and selectivities[5]. The data for the query is then built by fabricating relations with the chosen characteristics. The query is executed by performing the sequence of operations specified by the query optimization heuristic.

Currently, we are conducting experiments with a wide range of diverse queries to ascertain the following:

[5] Constraints are imposed to ensure we generate a valid, executable query.

— Can our heuristic perform better than traditional semijoin based heuristics ?
— What is the improvement in performance ?
— Are there significant reductions during the reverse reduction phase ?
— Would the use of multiple filters lead to an improvement in performance ?
— Are we shipping too many tuples in error because of collisions ?

We expect to provide some answers to these questions at conference time.

Conclusions Distributed query optimization is an important research issue. In this paper, we present a novel approach which uses reduction filters and quasi-joins to minimize the total volume of data transferred over the network during query processing. Currently we are evaluating our heuristic, using a comprehensive evaluation framework we have developed. The results of our performance study will be presented at conference time.

Acknowledgments Many thanks to Todd Bealor and Wendy Osborn for coding the heuristics and the evaluation framework; thanks to colleagues for discussions and technical support. This research is funded by NSERC.

References

[1] A. K. Ahn and S.C. Moon. Optimizing joins between two fragmented relations on a broadcast local network. *Info. Syst.*, 16(2), 1991.

[2] P. M. G. Apers, A. R. Hevner, and S.B. Yao. Optimization algorithms for distributed queries. *IEEE Transactions on Software Engineering, 9(1)*, 1983.

[3] P.A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. Rothnie. Query processing in a system for distributed databases (SDD-1). *ACM Trans. on Database Systems*, 6(4), 1981.

[4] P.A. Black and W.S. Luk. A new heuristic for generating semi-join programs for distributed query processing. *IEEE COMPSAC*, 581–588, 1982.

[5] L. Chen and V. Li. Improvement algorithms for semi-join query processing programs in distributed database systems. *IEEE Trans. on Computers*, 33(11), 1984.

[6] L. Chen and V. Li. Domain-specific semijoin: a new operation for distributed query processing. *Info. Sci.*, 52, 1990.

[7] M. Chen and P. S. Yu. Combining join and semi-join operations for distributed query processing. *IEEE Transactions on Knowledge and Data Engineering*, 5(3), 1993.

[8] H. Kang and N. Roussopoulos. Using 2–way semi-joins in distributed query processing. In *Proc. 3rd Int. Conf. on Data Engineering*, 1987.

[9] D. Knuth. *The art of computer programming.* Addison-Wesley, 1973.

[10] P. Legato, G. Paletta, and L. Palopoli. Optimization of join strategies in distributed databases. *Info. Syst.*, 16(4), 1991.

[11] N. Roussopoulos and H. Kang. A pipeline n-way join algorithm based on the 2–way semijoin program. *IEEE Trans. on Knowledge and Data Engineering*, 3(4), 1991.

[12] C. Wang, V. Li, and A. Chen. Distributed query optimization by one-shot fixed precision semi-join execution. In *Proc. 7th Int. Conf. on Data Engineering*, 1991.