# Collisions and reduction filters in distributed query processing.

**J.M. Morrissey**
School of Computer Science
University of Windsor
Windsor, Ontario N9B 3P4
joan@uwindsor.ca

**W.K. Osborn**
Department of Computer Science
University of Calgary
Calgary, Alberta T2N 1N4
osborn@cpsc.ucalgary.ca

**Y. Liang**
School of Computer Science
University of Windsor
Windsor, Ontario N9B 3P4
yanliang3@yahoo.com

**Keywords**: optimization, hash-semijoin, evaluation.
**Topics**: Computer networks and communications, software engineering.

## Introduction

The optimization of general queries in a distributed database management system is an important research issue. The problem is to select the best sequence of database operations that will process the query efficiently and minimize costs. Approaches include algorithms which are join-based [1, 2], semijoin–based [3, 4, 5, 6, 7, 8, 9, 10], or a combination of both [11]. Many algorithms concentrate on special types of queries [5, 12, 13, 14, 15, 6]. Some improvement algorithms have been developed [16, 17]. However, it has been shown [18, 19] that finding an optimal solution for a given general query is NP-hard so the aim is to produce an efficient but (perhaps) suboptimal evaluation strategy. A new approach is to use hash-semijoins, which employ reduction filters (also known as Bloom filters), to reduce costs. Bloom filters are first described in [20], and have mostly been used to improve the relational join operation [21, 22, 23, 24, 25]. Bloom filters have been applied to other relational operations [26, 27, 28] and file processing operations [29]. Mullin [30, 31] employs Bloom filters to improve the semijoin operation. His method is limited to the case where there are only two sites involved in the query and it has not been expanded to cater for general queries. Tseng and Chen [32] present a new relational operator called a hash-semijoin. They propose a method which will transform a sequence of semijoins, produced by any existing semijoin-based algorithm, into a more cost effective sequence by replacing some of the semijoins by hash-semijoins. But their approach has not been experimentally evaluated.

## The algorithm

In our recent work [33, 34, 35, 36] we use reduction filters. Each filter is an array of bits that functions as a very compact representation of the values of a join attribute in a relation. A hash function is used to set bits in the filter. It is then used to identify tuples (in other relations) which cannot belong to the result. The relations can be reduced significantly in size, data transfers can be reduced and costs minimized.

Our algorithm, which is based on reduction filters, can process general queries consisting of an arbitrary number of relations and join attributes. Each query is represented by a graph and an adjacency list. Each relation is usually only processed once — to minimize data transfers. However, if a filter changes during use then certain relations must be processed again — a queue is used to record this information. The algorithm consists of two phases: during phase one the adjacency list is used to determine the order in which the filters are constructed and used.

***Phase one:***

1. Select the relation with lowest in-degree for processing.
2. Scan adjacency list to see which filters must be constructed. If a filter is already available then concurrently use it to reduce the relation and produce all required filters.
3. If a filter has changed the use the following “ filter rule”: if a filter for a relation

changes then add that relation to the queue only if it has already been processed; it is not already on the queue; and it is not the most recently processed relation.

4. Use adjacency list to “remove edges” from query graph — that is, reduce the in-degree of each relation in the list by 1.
5. Mark relation as processed.

Repeat all steps until each relation has been processed once.

***Phase two:***

1. Remove relation from queue.
2. Reduce relation using all appropriate filters.
3. If a filter changes then use the “filter rule”.

Repeat all steps until the queue is empty.

Early experiments showed that our algorithm could reduce data transfers very substantially [33]. But this work had one serious limitation — we used a “perfect” hash function which meant that collisions were not possible. However, in a real-world database collisions in the reduction filter[1] are a reality and a problem. Such collisions mean that the hash-semijoin cannot achieve the same amount of reduction as a traditional semijoin would. In this paper we investigate the effect that collisions have on the performance of our algorithm. We expected that collisions would seriously hinder the algorithm and drastically affect our performance rates.

## The experiments

We have an experimental framework consisting of a collection of software to generate random queries with certain characteristics. Software is also used to generate the relations needed to execute each query. We assume a distributed relational database management system connected via a point to point network. There is no fragmentation nor replication. We consider only select-project-join (SPJ) queries, each consisting of a number of relations located at different sites which must be joined and the result made available at the query site. To evaluate, we compare our method not against another algorithm but against the effects of a “full reducer” — an algorithm which reduces the relations to just those tuples which can be part of the final result. We believe that this method is more objective and gives a better assessment of the performance of the algorithm.

[1] A collision occurs when two attribute values hash to the same address in the filter.

We designed a number of experiments to investigate the effects of collisions on the performance of the algorithm. In the first experiment we executed and optimized queries using a “perfect” hash function, one in which no collisions are possible. This gives a benchmark performance for the rest of the experiments. In the second experiment we simulate one non-perfect hash function and use a single filter for each joining attribute in each relation. Our framework allows us to specify and guarantee a certain percentage of collisions in the filter, so we can test the performance of the algorithm with different collision rates. (For full details of how we generated the different collision levels see [37]). At each collision rate we generated and executed 600 random queries, where each relation has between 200 and 600 tuples; the active domains had between 150 and 250 values; the selectivity of each attribute was set between 0.5 and 0.95; and each relation had approximately 75% of the joining attributes. For convenience the queries are grouped by type, where type 3–2 means that the query has three relations and two join-attributes. The objective was to investigate how different levels of collisions affected the performance of the algorithm. Our “performance measure” is the reduction achieved as a percentage of the full reduction possible. So a reduction of 70% means that our algorithm removed 70% of the tuples that would be removed by a full reducer algorithm.

In the third experiment we simulate two different hash functions and using the same percentages of collisions, we investigate how the use of two filters per joining attribute in each relation improves the performance of the algorithm.

The result tables are presented below:

| Type | Average reduction (%) | Percentage of queries fully reduced |
|---|---|---|
| 3-2 | 93.40 | 49 |
| 3-3 | 93.50 | 84 |
| 3-4 | 100 | 100 |
| 4-2 | 97.68 | 67 |
| 4-3 | 98.50 | 92 |
| 4-4 | 99.82 | 99 |
| 5-2 | 99.14 | 89 |
| 5-3 | 99.45 | 97 |
| 5-4 | 100 | 100 |
| 6-2 | 99.81 | 92 |
| 6-3 | 99.67 | 99 |
| 6-4 | 100 | 100 |
| Avg | 98.41 | 89 |

Table 1 Results for queries with no collisions.

| Type | 10% | 20% | 30% | 40% | 50% |
|---|---|---|---|---|---|
| 3-2 | 87.56 | 87.10 | 84.44 | 78.52 | 80.39 |
| 3-3 | 95.82 | 94.62 | 88.65 | 89.53 | 85.10 |
| 3-4 | 99.21 | 98.89 | 97.86 | 98.23 | 98.67 |
| 4-2 | 94.63 | 92.88 | 92.43 | 90.49 | 87.88 |
| 4-3 | 97.83 | 95.61 | 96.52 | 97.34 | 95.35 |
| 4-4 | 100 | 100 | 100 | 98.99 | 99.23 |
| 5-2 | 98.58 | 97.80 | 96.53 | 96.06 | 93.57 |
| 5-3 | 99.67 | 98.65 | 97.89 | 98.72 | 97.56 |
| 5-4 | 99.94 | 100 | 100 | 99.23 | 100 |
| 6-2 | 99.43 | 98.57 | 98.58 | 97.86 | 97.73 |
| 6-3 | 100 | 99.14 | 98.76 | 99.13 | 99.42 |
| 6-4 | 99.98 | 99.79 | 100 | 100 | 99.95 |
| Avg | 97.72 | 96.92 | 95.97 | 95.34 | 94.57 |

Table 2 Average reduction for 10%–50% collisions using a single filter.

| Type | 10% | 20% | 30% | 40% | 50% |
|---|---|---|---|---|---|
| 3-2 | 85.53 | 89.14 | 88.63 | 88.91 | 87.53 |
| 3-3 | 96.45 | 96.44 | 96.13 | 96.55 | 96.77 |
| 3-4 | 97.48 | 98.20 | 98.11 | 97.99 | 97.86 |
| 4-2 | 93.46 | 96.88 | 95.03 | 95.06 | 93.12 |
| 4-3 | 98.53 | 97.98 | 98.14 | 98.55 | 98.36 |
| 4-4 | 98.79 | 99.95 | 100 | 99.25 | 99.33 |
| 5-2 | 97.76 | 98.09 | 97.30 | 97.41 | 97.39 |
| 5-3 | 99.35 | 98.99 | 99.20 | 99.33 | 98.46 |
| 5-4 | 100 | 99.63 | 100 | 99.28 | 99.56 |
| 6-2 | 99.37 | 99.30 | 99.41 | 99.02 | 99.01 |
| 6-3 | 100 | 99.48 | 99.65 | 100 | 99.99 |
| 6-4 | 100 | 100 | 100 | 100 | 100 |
| Avg | 97.84 | 97.63 | 97.61 | 97.28 | 97.83 |

Table 3 Average reduction for 10%–50% collisions using two filters.

Collision rates:

| | 10% | 20% | 30% | 40% | 50% |
|---|---|---|---|---|---|
| 1 filter | 74.66 | 70.66 | 68.00 | 65.08 | 61.92 |
| 2 filters | 81.00 | 81.33 | 79.75 | 79.33 | 76.50 |
| Difference | 6.34 | 10.67 | 11.75 | 14.25 | 14.58 |

Table 4 Average number of queries fully reduced, averaged over all query types, at 10%-50% collision rates.

## Conclusions

The results show that, on average, collisions in the filters do not have a huge impact on the performance of the algorithm. Even with 50% of the values colliding we can still reduce relations by over 90%, using either one or two filters. Moreover, it seems that the difference in the average reduction is not significantly affected by the use of a second filter for each joining attribute of each relation. It would appear that the query type has more of an influence on the reduction rate than the percentage of collisions or number of filters used. This is not an unexpected result since, as the number of joining attributes increases we use more filters on a relation and this reduces the impact of the collisions.

However, there is a very significant difference in performance when we look at the percentage of queries which were fully reduced[2]. The use of an additional filter for each joining attribute significantly increases the number of queries which are fully reduced. This increase is most marked at the higher collision rates.

The conclusions of this paper are very significant. To our knowledge no one has ever examined the effect of collisions on the performance of a hash-semijoin based algorithm. Our main conclusion is that the collisions, even at 50%, do not significantly affect performance.

A full discussion of the results will be presented at conference time.

## References

[1] J. Ahn and S. Moon, "Optimizing joins between two fragmented relations on a broadcast local network.," *Info. Syst.*, vol. 16(2), pp. 185–198, 1991.

[2] P. Legato, G. Paletta, and L. Palopoli, "Optimization of join strategies in distributed databases," *Info. Syst.*, vol. 16(4), pp. 363–374, 1991.

[3] P. Bernstein, N. Goodman, E. Wong, C. Reeve, and J. Rothnie, "Query processing in a system for distributed databases (SDD-1)," *ACM Trans. on Database Systems*, vol. 6(4), pp. 105–128, 1981.

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

[5] P. Apers, A. Hevner, and S. Yao, "Optimization algorithms for distributed queries," *IEEE Transactions on Software Engineering, 9(1)*, pp. 51–60, 1983.

[6] L. Chen and V. Li, "Improvement algorithms for semi-join query processing programs in distributed database systems.," *IEEE Trans. on Computers*, vol. 33(11), pp. 959–967, 1984.

[7] H. Kang and N. Roussopoulos, "Using 2–way semi-joins in distributed query processing," in *Proc. 3rd Int. Conf. on Data Engineering*, pp. 644–650, 1987.

[8] N. Roussopoulos and H. Kang, "A pipeline n-way join algorithm based on the 2–way semi-join program," *IEEE Trans. on Knowledge and Data Engineering*, vol. 3(4), pp. 486–495, 1991.

[9] L. Chen and V. Li, "Domain-specific semi-join: a new operation for distributed query processing," *Info. Sci.*, vol. 52, pp. 165–183, 1990.

[10] 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*, pp. 756–763, 1991.

[11] M. Chen and P. S. Yu, "Combining join and semi-join operations for distributed query processing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 5(3), pp. 534–542, 1993.

[12] D. Chiu, P. Bernstein, and Y. Ho, "Optimizing chain queries in a distributed database system," *Siam Journal of Computing*, vol. 13(1), pp. 116–134, 1984.

[13] A. Chen and V. Li, "A method for interpreting tree queries into optimal semijoin expressions," in *ACM SIGMOD*, 1980.

[14] D. Chiu and Y. Ho, "Optimizing star queries in a distributed database system," in *VLDB*, pp. 959–967, 1984.

[15] C. Yu, Z. Ozsoyoglu, and K. Kam, "Optimization of distributed tree queries," *J. Comp. Sys. Sci.*, vol. 29(3), pp. 409–445, 1984.

[16] P. Boderick, J. Pyra, and J. Riordan, "Correcting execution of distributed queries," in *Proc. of 2nd Int. Symp. on Databases in Parallel and Distributed Systems*, pp. 192–201, 1990.

[2] The relations had been reduced to just those tuples which can participate in the final join.

[17] P. Boderick, J. Riordan, and J. Pyra, “Deciding to correct distributed query processing,” *IEEE Trans. on Knowledge and Data Engineering*, vol. 4(3), pp. 253–265, 1992.

[18] A. Hevner, *The optimization of query processing in distributed database systems*. PhD thesis, Perdue University, 1980.

[19] C. Wang and M. Chen, “On the complexity of distributed query optimization,” tech. rep., IBM Technical Report RC 18671, 1993.

[20] B. Bloom, “Space/time tradeoffs in hash coding with allowable errors,” *Comm. ACM*, vol. 13(7), pp. 422–426, 1970.

[21] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen, “The GAMMA database machine project,” *IEEE Transactions on Knowledge and data engineering*, pp. 44–62, 1990.

[22] G. Z. Qadah, *Filter-based join algorithms on uniprocessor and distributed-memory multiprocessor database machines*, vol. 303 of *Lecture Notes in Computer Science*, pp. 388–413. Springer-Verlag, 1988.

[23] G. Z. Qadah and K. Irani, “The join algorithms on a shared-memory multiprocessor database machine,” *IEEE Transactions on Software Engineering*, pp. 1168–1683, 88.

[24] P. Valduriez and G. Gardarin, “Join and semijoin algorithms for a multiprocessor database machine,” *ACM Transactions on database systems*, pp. 133–161, 1984.

[25] J. Mullin, “Estimating the size of a relational join,” *Information Systems*, vol. 18(3), pp. 189–196, 1993.

[26] G. Graefe and K. Ward, “Dynamic query evaluation plans,” in *ACM SIGMOD*, pp. 358–366, 1989.

[27] G. Graefe, “Query evaluation techniques for large databases,” *ACM computing surveys*, pp. 73–170, 1993.

[28] C. Mohan, D. Haderle, Y. Wang, and J. Cheng, *Single table access using multiple indexes: optimization, execution and concurrency control techniques.*, vol. 416 of *Lecture Notes in Computer Science*, pp. 29–43. Springer-Verlag, 1990.

[29] D. Severance and G. Lohman, “Differential files: their application to the maintenance of large databases,” *ACM Transactions on database systems*, pp. 257–267, 1976.

[30] J. Mullin, “(1983) A second look at bloom filters,” *Comm. ACM*, vol. 26(8), pp. 570–571, 1983.

[31] J. Mullin, “Optimal semijoins for distributed database systems,” *IEEE trans. on software eng.*, vol. 16(5), pp. 558–560, 1990.

[32] J. Tseng and A. P. Chen, “Improving distributed query processing by hash-semijoins,” *Journal of Information Science and Engineering*, vol. 8, pp. 525–540, 1992.

[33] J. Morrissey and W. Osborn, “Distributed query optimization using reduction filters,” in *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering*, (University of Waterloo, May 1998), pp. 707–710.

[34] J. Morrissey and W. Osborn, “Experiments with the use of reduction filters in distributed query optimization,” in *Proceedings of the 9th Internaltional Conference on Parallel and Distributed Computing and Systems (PDCS’97)*, (Washington, D.C., October 1997), pp. 327–330.

[35] J. Morrissey, “Reduction filters for minimizing data transfers in distributed query optimization,” in *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering*, (University of Calgary, May 1996), pp. 198–202.

[36] J. Morrissey and W. Osborn, “The effect of collisions on the performance of reduction filters.,” in *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering*, (Edmonton, May 1999).

[37] W. Osborn, “The use of reduction filters in distributed query optimization,” Master’s thesis, The University of Windsor, 1998.