

Pushing Down Bit Filters in the pipelined execution of large queries ^{*}

Josep Aguilar-Saborit, Victor Muntés-Mulero, and Josep-L. Larriba-Pey^{**}

Departament d'Arquitectura de Computadors
CEPBA-IBM Research Institute
Universitat Politècnica de Catalunya - Campus Nord UPC
C/Jordi Girona, 1-3 08034 Barcelona
Mòdul D6 Despatx 005
{jaguilar, vmuntes, larri}@ac.upc.es

Abstract. We propose a new strategy to use Bit Filters for complex pipelined queries on large databases that we call Pushed Down Bit Filters. The objective of the strategy is to make use of the Bit Filters already created for upper nodes of the execution plan, in the leaves of the plan. The aim of this strategy is to reduce the traffic between the nodes of the execution plan. When traffic is reduced, the amount of CPU work is reduced and, in most of the cases, I/O is also reduced. In addition, this technique shows no degradation in cases with little effectiveness.

1 Introduction

The execution time of queries to large relational databases is greatly influenced by the I/O. However, the pipelined nature of execution engines makes the task of minimizing the I/O very complex, especially for large multi-join queries.

There are many strategies to reduce the I/O of multi-join queries like *hash teams* [9], the use of bitmaps to implement joins [10] or the use of Bloom Filters, also called Bit Filters [1].

In this paper we deal with Bit Filters. They are an efficient tool to reduce the amount of I/O in the execution of pipelined queries. In particular, they are used in commercial DBMSs like DB2 [6], Oracle [5] and SQLServer [6] to reduce the I/O of Hash Join operations.

In those commercial DBMSs, Bit Filters are created during the build phase of the Hash Join operation. Then, during the probe phase, they are used to filter out records of the probe Relation. Quite interestingly, they may reduce considerably the amount of I/O of the Hash Join nodes [12] at the cost of keeping a bit map structure in memory during a part of the query execution. In addition, Bit Filters adapt naturally to the execution of Hash Joins; they are created during

^{*} This work was supported by the Ministry of Education and Science of Spain under contract TIC2001-0995-C02-01, CEPBA, CIRI and an IBM CAS Fellowship grant.

^{**} The authors want to thank Calisto Zuzarte and Adriana Zubiri for their helpful comments in preliminary versions of this paper.

the build phase, which is sequential by nature, and are used during the probe phase, which is pipelined by nature.

In this paper we want to take a step further in the use of Bit Filters. We explain and analyze how, being created for their local use in one Join node, they can be used by the Scan nodes of pipelined execution plans. We call these Pushed Down Bit Filters (PDBF) because, with our strategy, Bit Filters are Pushed Down to the leaves of execution plans.

We evaluate Pushed Down Bit Filters with queries implemented in PostgreSQL [11] on a 1 GB TPC-H populated database. We measure the amount of data traffic among nodes. We also measure the I/O and execution time for different join heap sizes. While the data traffic among nodes is independent of the DBMS, the I/O depends on the amount of memory available to perform the query and the execution time depends on the platform used and the DBMS.

We compare the Pushed Down Bit Filters to the basic PostgreSQL execution and the classic use of Bit Filters. The results show that our strategy reduces the execution time of some queries in more than a 50% both compared to the basic implementation and to the classic use of Bit Filters.

This paper is organized as follows. Section 2 describes our strategy, Pushed Down Bit Filters, explains in detail when the strategy can be used and the implementation details of the strategy in the execution engine. In Section 3 we describe and analyze the results obtained. Section 4 describes the previous work in the area. Finally, in Section 5 we conclude.

2 Pushed Down Bit Filters

The objective of Pushed Down Bit Filters is to obtain the maximum benefit from the Bit Filters created in the Join nodes of a query. In this paper we concentrate on Hash Joins and the Bit Filters created by them although our strategy could be generalized to other join operations with little changes.

PDBF are as simple as using the Bit Filters generated in the Hash Join nodes of the query plan in the lower leaves of the plan. This will be possible in those leaf nodes scanning a Relation with attributes on which the already created Bit Filters can be applied. Even the attributes that have non-functional dependencies with the attribute used to create the Bit Filter can be filtered.

The aim of this early elimination of tuples is to reduce the amount of data to be processed by the plan, and consequently, to decrease the work of its nodes.

Figure 1 shows a simple example of the strategy. The upper Hash Join operation starts with the build phase creating a Bit Filter on attribute a of relation $R3$. Once the build phase is finished, the join node invokes the probe subtree to obtain tuples. In the example, the subtree is formed by a join on attribute b , and two Scan nodes. When the Scan node of relation $R2$ is invoked, it will look up the Bit Filter given that Relation $R2$ contains attribute a . With this, we reduce the output cardinality of the Scan node, thus, the volume of data to be processed by join node " $R1.b = R2.b$ ". Also, the output cardinality of join node " $R1.b = R2.b$ " is reduced and, consequently, the amount of data to be processed

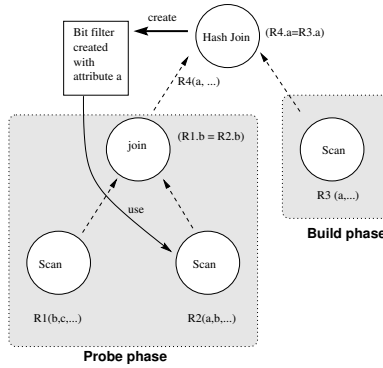


Fig. 1. An example of the Pushed Down Bit Filters.

by the Hash Join node. If R1 contained attribute a , the Scan of R1 could also look up the Bit Filter. This would happen in the case of composed primary keys.

Our strategy does not require the creation of any special large data structure. The only information necessary in the Scan nodes is a link between each attribute and its corresponding Bit Filter.

Applicability. The special characteristics of Bit Filters and query plans impose, at least, the following restrictions to our strategy:

- A Scan node can only look up the Bit Filters created by Hash Join operations within the same **select** statement. Two different **select** statements have data flow streams that do not interfere.
- A Bit Filter can only be pushed down to the Scan nodes of the subtree that belongs to the probe Relation of that Hash Join.

Implementation details. Once the target Scan nodes are chosen, it is necessary that those Scan nodes maintain some information about the Bit Filters they have to look up and the attribute affected by each Bit Filter. We define the set S^* of pairs $\langle \text{Bit Filter}, \text{attribute} \rangle$ as the data used by a Scan node to apply the Pushed Down Bit Filters. With this information, for every tuple processed by a Scan node, each Bit Filter of the set S^* will be applied to the corresponding attribute. If any of the Bit Filters returns a zero for a tuple, that tuple can be immediately discarded.¹

It is important to note the low cost of our strategy:

- Memory space: It is only necessary to keep the set S^* for each Scan node. Thus, the memory space necessary is not significant at run time.
- CPU time: If the Bit Filters are looked up during the Scan operations, it will not be necessary to look them up during the probe phase of any of the upper Hash Joins. Thus, our strategy does not imply any replication of work.

¹ All the Bit Filters are accessed with one only hash function, h

3 Evaluation

In this section we present different results. First, we evaluate a query with different parameter sets to show the features of our strategy. The evaluation results measure traffic volume and execution time. Then, we give measures for the I/O volume using one of the parameter sets. Before starting with the detailed evaluation, we explain the query that we analyze and the software and hardware setups we used.

Query description. We have executed a synthetic query with four different parameter sets on a TPC-H populated database. Figure 2.a shows the query used and Figure 2.b shows its execution plan. The query counts the number of units of each product of brand z (“ $p.brand = z$ ”) sold starting on date x (“ $o.orderdate > x$ ”) for available quantities of stock larger than y (“ $ps.availqty > y$ ”). We have varied the query restrictions to obtain different behaviors of the same query. This leads to the four different parameter sets that we call query executions, shown in Table 1.

It is frequent to have queries with several joins on a central table (called Star Joins) as it happens here on *Lineitem*. This does not mean that our strategy can only be applied to this type of queries (see Section 2 for the applicability of our strategy). However, this is a good example for the evaluation of our technique.

Figure 2.b also shows (in bold) the application of the PDBF in this query. There, we can see how the Scan on *Lineitem* looks up the Bit Filters of HJ1, HJ2 and HJ3, while the Scan on *Part* only looks up the Bit Filter of HJ2.

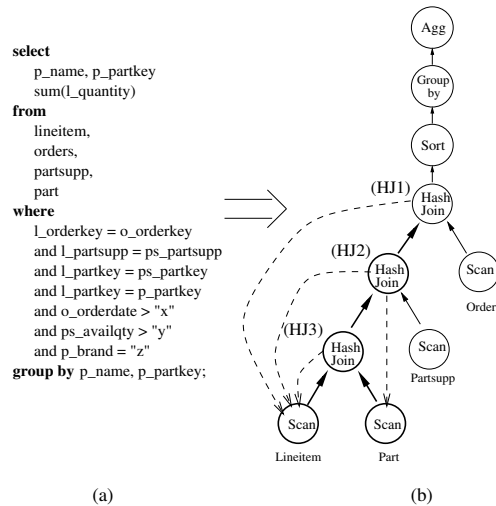


Fig. 2. Query used (a), execution plan and use of Bit Filters (b).

Table 1. Percentage of tuples eliminated by each combination of parameters and efficiency of the Bit Filters. (P) for Part, (PS) for Partsupp and (O) for Orders.

set	Fraction of Tuples eliminated			Efficiency of Bit Filter		
#	P."z"	PS."y"	O."x"	HJ3	HJ2	HJ1
1	0.0	0.7	0.8	0.0	0.16	0.81
2	0.0	0.7	0.15	0.0	0.16	0.18
3	0.8	0.15	0.15	0.81	0.0	0.18
4	0.0	0.15	0.15	0.0	0.0	0.18

We have used restrictions "x", "y" and "z", to control the amount of tuples projected from the build Relation of each Hash Join. Table 1 shows the fraction of tuples eliminated with each restriction for each different query execution. The restriction is one of the factors determining the effectiveness of a Bit Filter. The efficiency of the Bit Filters (fraction of tuples filtered out from the scan where the Bit Filter is applied) is also shown in Table 1 for each Hash Join. We have computed these figures using the data obtained after the execution of the query.

Software Setup. We have generated a TPC-H database with scale factor 1 (1 GB) and executed the 4 queries on it.

We have used PostgreSQL to implement the use of Bit Filters and PDBF. Therefore, we have used three versions of the PostgreSQL query engine; the original engine without Bit Filters, the engine with Bit Filters and the engine with PDBF. The size of the Bit Filters we use is larger than the number of distinct values of the attributes used to build the Filters.

The implementations do not include any modification of the Optimization engine. Instead, they only implement the query plan and the modifications of PostgreSQL necessary to execute the query that we designed for the tests.

PostgreSQL decides the number of partitions in the Hybrid Hash Join at optimization time. With the application of our strategy, some Hybrid Hash Join nodes end up having a very small number of tuples in the build Relation. This means that the number of partitions could be smaller than planned by the optimizer of PostgreSQL. However, our implementations keep the number of partitions planned by the optimizer at the cost of making our strategy less efficient.

The memory space assigned to each Hash Join node is called *Sortmem* in PostgreSQL. For the following analysis, we use different *Sortmem* sizes ranging from very restrictive to a size that allows the execution of Hash Joins in memory with one only data read per Relation. The *Sortmem* sizes range from 5 MB (very restrictive) to 60 MB (one data read). The Bit Filters that we have implemented are always used unless we know its effectiveness is zero, in which case, they are not created. The Bit Filters used are stored in a *Bit Filter Heap* that we created.

Hardware Setup

We run all the experiments on a 550M Hz Pentium-III based computer, with 512 MB of main memory and a 20 GB IDE Hard Disk. The Operating System used was Linux Red Hat 6.2.

Traffic and Execution time

Figure 3 shows results of execution time, and tuple traffic. The plots show results for the basic implementation of PostgreSQL, for the use of Bit Filters in the nodes where they are created, and PDBF. The execution time plots show results for different *Sortmem* sizes while the traffic plots show the total traffic per node in the execution plan (the traffic is independent from the *Sortmem* size). The traffic of a node is equivalent to the output cardinality of that node. It is important to note that only the traffic of nodes HJ3, HJ2, Scan Lineitem and Scan Part are affected by PDBF.

While the execution time is specific to the PostgreSQL implementation, the traffic avoided depends on the data and can be regarded as a general result of applicability to any DBMS.

We divide the analysis into two groups of queries.

Query executions #1 and #2: These are the query executions that obtain better results both in execution time and tuple traffic. As shown in the plots of those query executions, the major improvements in the traffic are obtained by nodes HJ2, HJ3 and the Scan of Lineitem. This is because the Bit Filter created in HJ1 with Relation *Orders* has significant efficiencies in query execution #1 (see Table 1). When pushing down that Bit Filter, the Lineitem table is filtered out significantly, reducing the traffic in all its upper nodes. The plots show that the smaller the traffic, the larger the reduction in execution time.

Another aspect to understand is the fact that the size of the *Sortmem* has an insignificant influence on query execution #1 for our strategy while it has a somewhat more significant influence on the execution time of executions #2. The reason is that the Bit Filter of HJ1 has a large efficiency (81%) in query execution #1 which reduces the Lineitem table to an extent that minimizes the I/O to the compulsory reads from disk even for small *Sortmem* sizes.

One final aspect to note is that the executions on *Sortmem* sizes of 60 MB show the reduction in CPU effort of the total execution time. In the three PostgreSQL implementations, the size of the *Sortmem* reduces the amount of disk reads to the compulsory ones. It is important to note that, even in those cases, the amount of CPU time saved is significant which gives strength to our strategy.

Query executions #3 and #4: These executions achieve good results but are not as significant as those in the previous query executions. They also show that PDBF are not a burden even when there is little benefit to obtain. Query execution #3 has a significant restriction on Part. This means that HJ3 only has the compulsory reads from disk because the build Relation fits in memory. Therefore, the output cardinality of HJ3 is very small in any of the three engines tested. Also, almost all the tuples filtered out by our strategy on Lineitem are caused by the Bit Filter created in HJ3 and the only work saved is that of projecting the tuples from the Scan of Lineitem up to HJ3, which implies a very little amount of execution time.

Query execution #4 does not have a restriction on Part and it has very weak restrictions on Partsupp and Orders. Most probably, Bit Filters would not

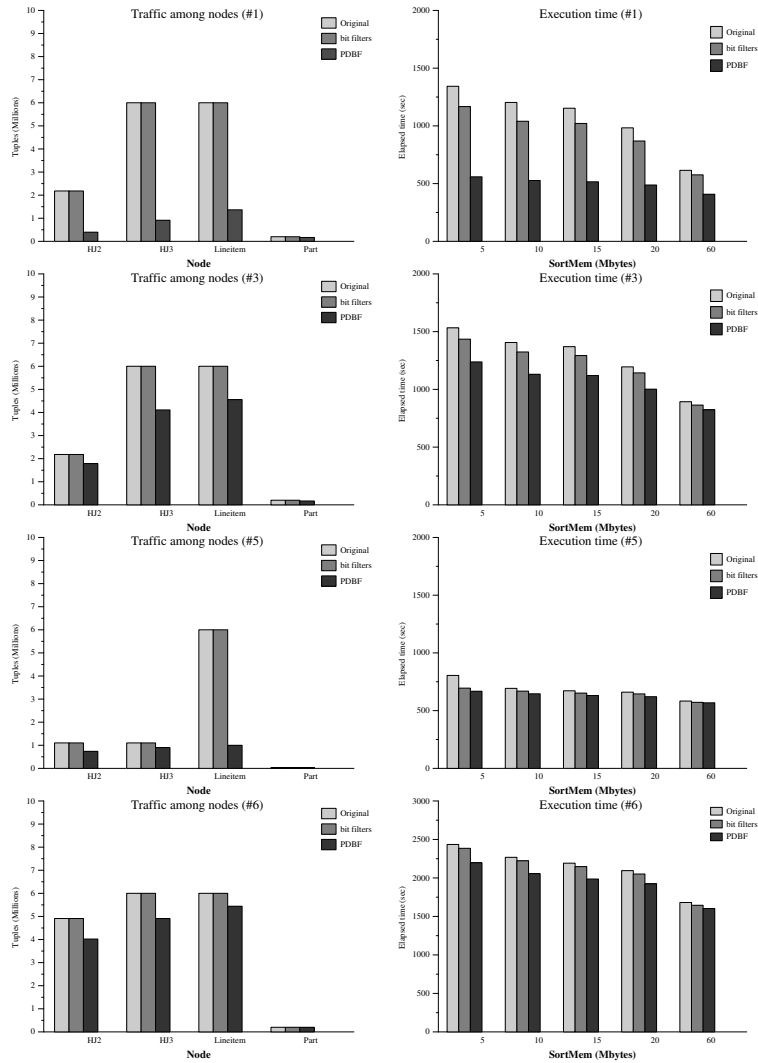


Fig. 3. Execution time and traffic across nodes for executions #1, #2, #3 and #4. PDBF stands for Pushed Down Bit Filters.

be used in this case (the optimizer would not trade memory for an uncertain execution time benefit).

I/O for query execution #1

Figure 3 shows that PDBF on query execution #1 improves the execution time of the basic and Bit Filter implementations of PostgreSQL by more than a 50%. Now we analyze the implications of the I/O on that query execution.

Figure 4 shows the I/O performed by each of the three Hash Joins involved in query execution #1. We do not include the I/O for the Scan operations because

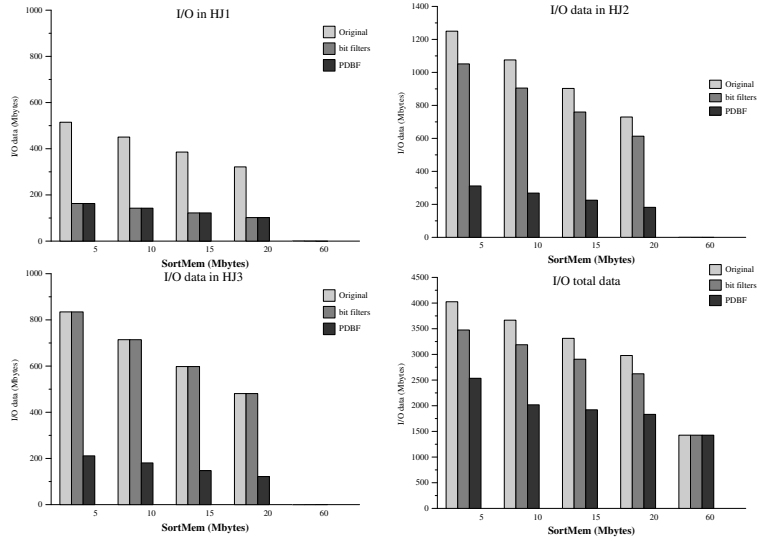


Fig. 4. I/O for query execution #1. PDBF stands for Pushed Down Bit Filters.

it does not change. The amount of I/O in Hash Join operations varies depending on the amount of data in the build Relations and the amount of memory available for the Hash Joins. If the amount of data in the build Relations fits in memory, the amount of I/O is zero in those operations, as shown in Figure 4.

The plots show a significant improvement of more than a 50% of PDBF compared to the other implementations. There is no improvement of our strategy in HJ1 compared to the only use of Bit Filters. This is because the Bit Filter is in the topmost Join and the Bit Filter already reduces the probe table before partitioning it. The amount of I/O on HJ2 and HJ3 is reduced due to the fact that the Bit Filters are applied in the bottom leaves of the plan. Notice that as the *SortMem* size is incremented, the I/O of the basic and Bit Filters executions is reduced but is still very far from that of the I/O of our strategy.

As shown in Figure 4, the execution on 60 MB *Sortmem* just uses the amount of IO necessary for compulsory reads, even for the original implementation in PostgreSQL. However, PDBF show a significant reduction of execution time (see Figure 3) because the amount of CPU work incurred by the tuple traffic saved is significant.

4 Related work and discussion

The use of Bit Filters is a popular strategy to support decision support queries. They have been used both in parallel Database management systems [7, 12, 13] and in sequential DBMSs [2] in order to speed-up the execution of joins. The original idea of the Bit Filter comes from the work by Bloom [1] where he designed them to filter out tuples without join partners.

The most relevant related work is [4]. Its aim is also to reduce the traffic between query nodes in order to reduce the I/O and the CPU costs. However, this proposal has a couple of limitations that we overcome with our Pushed Down Bit Filters. First, the algorithm proposed in [4] creates Bit Filters whenever an attribute belonging to one Relation or to the outcome of an operation can filter out tuples from another Relation. This leads to additional work and an extra need for memory space. Our technique relies on the Bit Filters already created, as we have shown in the paper. This saves time and memory space.

Second, as a consequence of the exhaustive creation of Bit Filters, the algorithm in [4] requires a non-pipelined execution. This is so because Bit Filters have to be created in full in order to be used. Thus, operations that create Bit Filters have to materialize their results. Materialization of nodes always incurs additional I/O. Our technique does not change the pipelined model of query execution. The pipelined execution of Hash Joins yields a better performance for multi-join queries as shown in [14], which also concludes that pipelined algorithms should be used in dataflow systems. In addition, most of the commercial DBMSs such as DB2 or Oracle use the pipelined execution philosophy to implement their DBMS engines [8]. Our Pushed Down Bit Filters require a small effort to be included in modern pipelined execution engines.

Generalized hash teams also use Bit Filters with a different purpose than we do here [9]. The idea comes from the original idea of *hash teams* in [6].

The technique uses a cascaded partitioning of the input tables using bitmaps. The partitioning is aimed at reducing the amount of I/O during the joining process and on the final aggregation. The bitmaps are used as a partitioning device. Our approach is not limited to multi-joins with aggregation but to a more general class of joins and allows one table being filtered by as many Bit Filters as different join attributes are projected from that table.

Generalized hash teams also have the limitation of false drops. When two different values map into the same Bit Filter position there may be false drops. False drops may cause a considerable amount of I/O in skewed environments. Our approach, given that it is not based on partitioning, does not have the false drops problem.

Other lines of related work include the use of Bit Filters as bitmap indexes [3, 10, 15]. In those cases, Bit Filters are used to perform joins in data warehousing contexts with the only use of bitmaps that reduce the I/O cost.

5 Conclusions

In this paper we propose PDBF, a strategy to use the Bit Filters created in the upper leaves of pipelined query execution plans, in the Scan nodes of the plan.

We can outline the following conclusions for our strategy. First, our strategy reduces the amount of tuple traffic of the query plan significantly. This has a direct influence on the amount of work to be performed by the nodes. Second, as a consequence of the reduction of traffic, it is possible to reduce significantly the amount of I/O of some queries. Third, the direct consequence of the reduction

of traffic and I/O is the reduction of execution time whenever the strategy can be used. The amount of benefit varies depending on whether it comes only from the traffic or both the traffic and the amount of I/O. Fourth, our strategy is simple to use. It just uses the Bit Filters already created by the Join nodes of the plan. PDBF require very simple run time data structures that only have to be managed by the Scan operations of the plan.

As a consequence of the benefits that can be obtained and the low implementation cost, it is most advisable to use PDBF in modern pipelined DBMSs.

Topics of future research related to PDBFs are optimization and scheduling issues and a further understanding of their applicability.

References

1. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
2. K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 323–333, 1984.
3. C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *Proc. of the SIGMOD Conf. on the Management of Data*, pages 355–366, 1998.
4. M.-S. Chen, H.-I. Hsiao, and P. S. Yu. On applying hash filters to improving the execution of multi-join queries. *VLDB Journal: Very Large Data Bases*, 6(2):121–131, 1997.
5. P. Gonglor and S. Patkar. Hash joins: Implementation and tuning, release 7.3. Technical report, Oracle Technical Report, March 1997.
6. G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in microsoft sql server. In *Proceedings of the 25th VLDB Conference*, pages 86–97, August 1998.
7. H.-I. Hsiao, M.-S. Chen, and P. S. Yu. Parallel execution of hash joins in parallel databases. *IEEE-Trans. on Parallel and Distributed Systems*, 8(8):872–883, August 1997.
8. Roberto J. Bayardo Jr. and Daniel P. Miranker. Processing queries for first few answers. In *CIKM*, pages 45–52, 1996.
9. A. Kemper, D. Kossmann, and C. Wiesner. Generalized hash teams for join and group-by. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 30–41, September 1999.
10. P. O’Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, September 1995.
11. PostgreSQL. <http://www.postgresql.org/>.
12. Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proc. ACM SIGMOD*, pages 110–121, 1989.
13. Patrick Valduriez and Georges Gardarin. Join and semi-join algorithms for a multiprocessor database machine. *TODS*, 9(1):133–161, 1984.
14. A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.
15. M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for data warehouses. In *Intl. Conference on Data Engineering*, pages 220–230, 1998.