Control strategies for complex relational query processing in shared nothing systems

Lionel Brunie and Harald Kosch
LIP-CNRS
Ecole Normale Supérieure de Lyon
69364 Lyon cédex 07, France

Abstract
In this paper, we present an original and complete methodology for supervising relational query processing in shared nothing systems. A new control mechanism is introduced which allows the detection and the correction of optimizer estimation errors and load imbalance. We especially focus on the management of intra-processor communication and on the overlapping of communication and computation. Performance evaluations on an hypercube and a grid interconnection machine show the efficiency and the robustness of the proposed methods.

Key words *: parallel databases, control of parallel query processing, load balancing.*

1 Introduction

The increasing amount of data manipulated in very large databases[1] and the complexity of the queries to be processed make parallelism appear as one of the most promising research axes for future database applications [Val93].

However many technical issues remain widely open. Among them, the control of load imbalance and run-time adaptation of static parallelization strategies are certainly the most crucial problems. In that purpose, we propose to first detect such critical situation using a special hierarchy of supervisors. These supervisors can decide to trigger a re-optimization of the parallelization strategies and the computation of a new relation partitioning in order to balance the load. Data redistribution measures are achieved using an elastic redistribution technique. Experiments performed on an Intel iPSC860 and a Paragon machine show the effectiveness of that methodology.

This paper is organized as follows. After some short basic notes about parallel query processing (section 2), a brief description of the main open problems and a presentation of previous works follows (section 3). The basics of the control mechanism (run-time detection and correction of static optimization errors and load imbalance) are developed in section 4. Then, we present in section 5 the experiments we have performed and discuss in section 6 the global methodology follows. Finally, section 7 concludes this paper and points out future developments.

[1]Terabyte databases containing billions of records are no more exotic.

2 Some rudimentary notes about parallel relational query processing

The purpose of this paragraph is to give a basic introduction on parallel relational query processing.

Relational queries actually implement relational algebra operators and various representation formalisms have been proposed for modeling the processing of those queries (e.g. object graph, algebra expression [JK84]). However **query processing trees** [VZ94] are the most usually used :

*The leaves of a **query processing tree** represent the base relations that participate in the query and intermediate nodes model operations. These latters receive their input relations via the incoming edges and send the result relation through the outgoing edge to the next operation. The root of the tree produces the result of the whole query.*

The relational model is especially well-suited for parallelism. Processing trees have been usually chosen since they allow expressing different kinds of parallelism :

inter-operation parallelism : Operations lying on different paths of a query processing tree can be executed concurrently.

intra-operation parallelism : Each relational operation can be decomposed into several sub-operations to be executed on different partitions of the same relation.

pipeline parallelism : Two nodes lying on the same edge can be executed concurrently. Suppose a selection node is to be performed on the output of a join node. Clearly the selection node can start its job as soon as the first tuple has been processed by the join node then working in parallel with that latter.

A classical criterion to classify parallel query processing strategies is the shape of the processing tree. Two major forms are actually distinguished [VZ94], *linear* and *bushy* trees. In *linear* trees, only one join operator is executed at the same time. In bushy trees[2], two or more join operators can be processed at the same time (*inter-operation parallelism*)[3].

[2]See fig. 5 for an example of a bushy tree.

[3]More precisely, two classes of *linear trees* can be distin-

3 Problem formulation and previous works

Two basic optimization strategies have been studied. In a first class of methods [Hon92, HM94], the query is first processed by a sequential optimizer and parallelized afterwards. In a second class,[4] [SD90, VZ94] the parallelization strategy is included into the optimization process. In particular, the query optimizer is in charge of defining the *degree of inter-parallelism* (i.e. the number of relational operations to be executed in parallel) and the *degree of intra-parallelism* (i.e. the number of processors to be attributed to each operation).

The problem of optimizer estimation errors : Whatever the optimization strategy, it is necessary to estimate unknown relation parameters (e.g. relation size of intermediate relations and query parameters [MW94]).

Based on these parameters, some systems [HM94, SD90] determine the degree of inter(intra)-parallelism *at compile-time*. Unfortunately, a bad parameter estimation can cause big problems. Thus, as showed in [IC91] or [BK95], the relation size is specially sensible to error propagation: even if estimation errors on a join operation are relatively small, errors for whole queries can become enormous.

On the contrary, systems like DBS3 [CB91] and Informix [Inf94] determine the processors allocation at run-time. However this makes the code very complex and slow the execution down [Val93].

On the other hand. some authors propose to decide a processor allocation at compile-time and to eventually readjust it at run-time. Thus, Hong [Hon92] mentions such a readjustment for *shared memory systems*, but his ideas have not been implemented. Rahm et al. [RR95] and Mehta et al. [MD95] study reallocation strategies for *shared nothing systems*. Both authors only consider single join queries for which they propose a readjustment of the processors allocation based on the I/O- and the CPU- consumption of the operation. DB2 Version3 [TW94] also performs a dynamic reallocation, but only rearrange the allocation of I/O-streams and does not consider the processors allocation.

Finally, it should be noticed that none of latter systems integrate data skew processing strategies.

The problem of data skew : During query processing, the data of a same relation can be non-uniformly distributed over the processors (*data skew*).[5] In presence of such a phenomenon, ensuring good performances requires triggering explicit strategies of load-balancing. Indeed experiments [YT91] have shown that *data skew* could make computation times explode.

A lot of strategies of parallel join processing in presence of data skew have been proposed (see the survey of Mishra et al. [ME92]). Research interest has been mainly focused on hash-partition (or range-partition) join algorithms. In these methods, the tuples of the relations are redistributed over the available processors (*split phase*) before a local join (e.g. nested-loop or hash-join [NS92]) is applied (*join phase*). Query processing based on these algorithms can suffer from two kinds of data skew : **redistribution skew**, in which the processors receive different numbers of tuples after the *split-phase* ; **join product skew**, in which data skew occurs on the output tuples of a join. If the *redistribution skew* has been intensively studied [YT91, LT94], only DeWitt et al. considered the *join product skew* [NS92]. They showed that it might have the same slow-down effects as the *redistribution skew*. In order to smooth *join product skew* situations, they generate in the *split-phase* more partitions than processors. Even attenuated, join product skew can still be sufficient to considerably reduce the performance. Furthermore this work considered only simple queries.

guished : *left linear trees* (based on a data-parallel strategy and *right linear trees* (more pipelined oriented). For more information see [SD90].

[4] This is more particularly true for *shared nothing systems*.

[5] Even if the processors allocation is optimal with respect to the operators processing.

4 Basics of a control mechanism

In this paper, we focus on *shared nothing systems*. Indeed, this type of architecture is usually considered as the favorite architecture for highly scalable and extensible parallel database applications [Val93].

We propose to first perform a compile-time query optimization (see section 3), in order to fix the degree of inter- and intra-parallelism and choose the implementation methods (we used our optimizer described in [BK95]). This optimization process is based on an estimation model of the intermediate relations cardinality. Then at run-time, a control mechanism surveys the query processing and triggers data redistribution procedures when severe data skews or optimization estimation errors occur.

These control mechanism is based on two main phases: the **detection** and the **correction**.

In the detection phase, we first determine if the estimated relation cardinalities do not differ too much from the actual relation cardinalities and if the join product skew is not too important. This is done using a hierarchy of supervisors, specially adapted to the structure of relational queries (section 4.1). In case of problem, a new processor allocation and data distribution is computed according to the actual relation characteristics.

In the correction phase, load balancing and processors reallocation is performed by a two-step elastic redistribution. The first step (**local redistribution**) can be started as soon as one relational operator has terminated. The second phase (**global redistribution**) must wait for the termination of all operators running in parallel (section 4.3).

Remark that oppositely to DeWitt et al. [NS92] (see section 3), *join product skew* is corrected when it happens and not during the join split phase. This presents the advantage that it can be performed at the same time as a processor reallocation due to optimizer estimation errors.

4.1 Detection of optimizer estimation errors and data skew

Let X be a join operation and R the result relation. We note $card\_real(R)$ the real cardinality of R and $card\_estim(R)$ the estimated cardinality of R. Finally we note $card_p(R)$ the real cardinality of the partition of R processed by processor p.

A special processor, called **operation supervisor**, is in charge of collecting all the termination messages and calculating $card\_real(R)$. Then it detects if the *degree of intra-parallelism* must be readjusted and if *load balancing processing* must be performed :

The *degree of intra-parallelism* for the successor operation is readjusted if and only if :

$$\left\lceil \frac{card\_real(R) - card\_estim(R)}{card\_estim(R)} \right\rceil > k$$

The *Load balance processing* is undertaken, iff:

$$\frac{max(card_p(R)) - min(card_p(R))}{average(card_p(R))} > r$$

The constants r and k are system-specific and are to be defined by the database administrator.
Once an *operation supervisor* has finished its detection work, it notices a special processor, called **query supervisor**. All modification requests of the *degree of intra-parallelism* are examined with respect to the available processors. When a new processor allocation is decided, the *query supervisor* calls the parallel query optimizer [BK95] with the actual relation characteristics. This latter computes a new parallel query execution plan for the remaining of the query. Then this plan is communicated to all processors.
Using the same framework, one could also imagine to modify the *degree of inter-parallelism*, i.e. to change the number of operations to be processed in parallel, in the case of massive estimation errors. This point has not yet been implemented.

4.2 Interaction between the detection and the correction phases

Join product skews are locally detected at the operator level by *operation supervisors*. *Local redistributions* can therefore be triggered without waiting for the termination of other operations running in parallel. Once a global redistribution (i.e. a new processor allocation) is decided, all local running redistributions are stopped before the global redistribution starts. This strategy allows to save time when no global redistribution is necessary (without prejudice when no global redistribution is done).
The strategy described in this section is an adaptation of a load-balancing algorithm first developed in the field of image-processing [MR91]. We mainly have generalized this approach by integrating disk management and the readjustment of the *degree of intra-parallelism*.

4.3 Redistribution algorithm

In this section we suppose that a logical ring topology is mapped into the interconnection network. Processors are arranged along this ring so that processors holding two relations to be joined are neighbors on the ring. This allows to minimize the data movements and to maintain the locality of the relations.

Let $node(R)$ be the number of processors holding the relation R before redistribution, and $node'(R)$ the number of processors after redistribution.

Let P_q $(1 \leq q \leq node(R))$ the processor holding the segment (i.e. the part) of R $[s_1(q), s_2(q)]$ before the redistribution. Let $P_{q'}$ $(1 \leq q' \leq node'(R))$ the processor holding the segment $[s'_1(q'), s'_2(q')]$ after redistribution (see fig. 1, left scheme).

Let $rank_q(t)$ be the rank of the tuple t on the processor P_q. We define the global rank of this tuple t, $grank(t)$, as :

$$grank(t) = s_1(q) + rank_q(t)$$

The goal of the algorithm is to uniformly redistribute the relation over the processors $P_{q'}$. The destination processor $dest\_proc(t)$ for a tuple t on a processor P_q expresses by :

$$dest\_proc(t) = \left\lceil \frac{grank(t)}{card(R) \ / \ node'(R)} \right\rceil$$

Our redistribution algorithm works in 2 steps:
1) Distributed computation of the new indices $s'_1(q')$ *and* $s'_2(q')$ $(1 \leq q' \leq node'(R))$.
2) Redistribution of the tuples.

In the first step, every processor scans its relation segment in order to detect the tuples for which the value of the function dest_proc() is different from the value computed for the preceding tuple. Let t be such a tuple. Clearly, a new index s'_1 can be defined as : $s'_1(dest\_proc(t)) = grank(t)$.
These new indices are communicated to every processor (personalized broadcast), which therefore knows which tuples it has to exchange with its neighbors in the ring.

The second step performs the redistribution of the tuples using the information obtained in the first step (see fig. 1, right scheme). Each processor P_q first sends tuples to its successor and then to its predecessor on the ring. Remark that communications to neighbors are overlapped one to the others. These communications are then repeated until every tuple has arrived to its destination processor.

When working with very large databases, the relation partitions usually can not fit in main memory. In order to avoid expensive overflow processing, most of hash-based join algorithms work on partitions of relations, called *buckets* (e.g. *Grace-hash join* [NS92]). Thus, in such approaches, the last processed bucket of the join result relation is kept in main memory. Two cases are then distinguished. When the amount of data to be redistributed does not exceed the last bucket size, only memory-resident tuples must be moved (see fig. 2). Otherwise some tuples must be fetched from disk. This disk access can be overlapped with the redistribution procedure.
Fig. 2 shows an example of redistribution performed on buckets. Each relation is split into three buckets (dotted lines show the bucket frontiers). In this example, the number of tuples to be redistributed from proc.1 and 3 is sufficiently small so that only memory-resident tuples have to be moved.

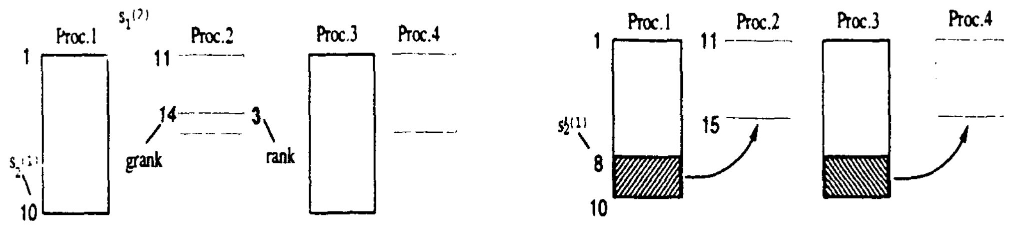

Figure 1: Example of a local redistribution. *Left:* Start situation. *Right:* Redistribution step.

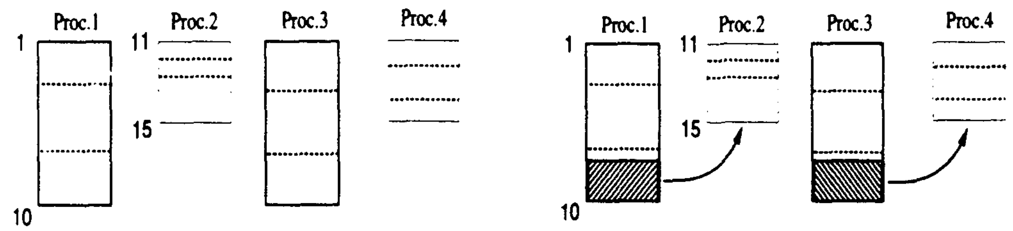

Figure 2: Example of a local redistribution performed on buckets.

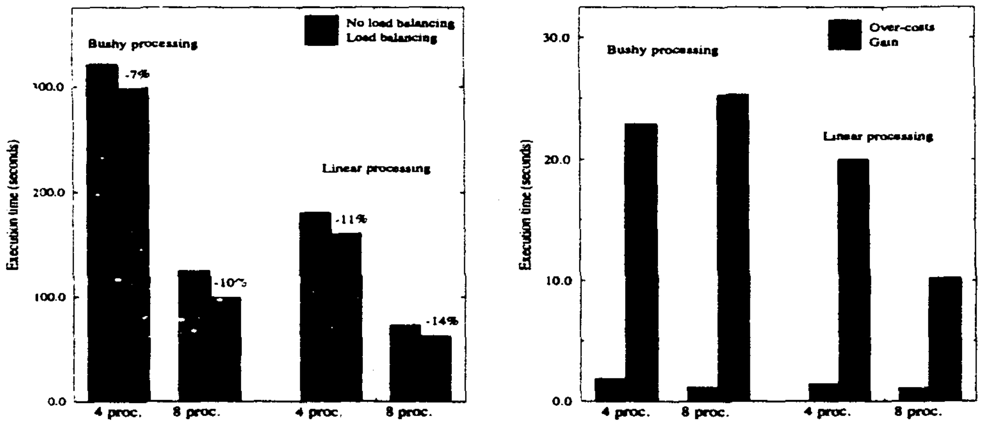

Figure 3: *Left:* Comparison of execution times (iPSC860) with(out) load balancing. *Right:* Overcosts of load balancing in comparison with the gain (iPSC860).

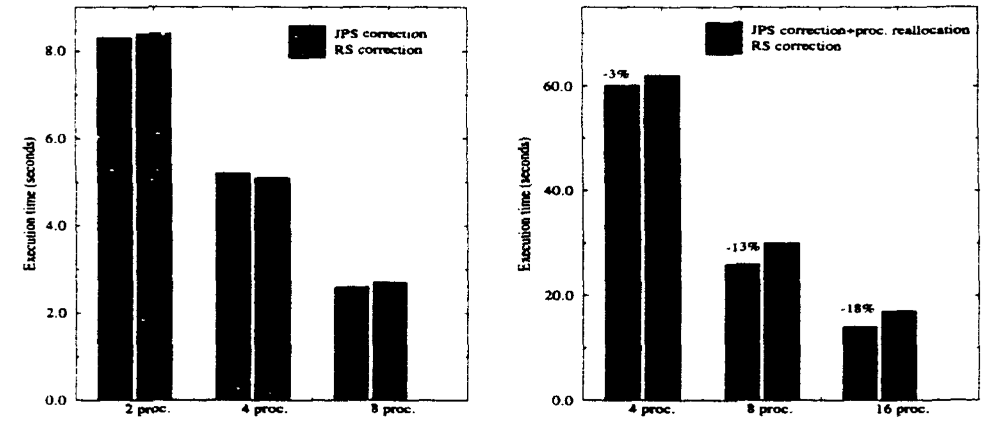

Figure 4: Comparison of our approach with a classic technique used for correcting redistribution skew (Paragon). *Left:* No query optimization error. *Right:* Query optimization error.

5 Experimental results

This section reports the experiments we have performed on an Intel iPSC860 (up to 8 processors) and an Intel Paragon (up to 16 processors). The interconnection networks of the iPSC860 and the Paragon parallel machines provide very good performances in the case of intensive personalized communications, which is especially adapted to our problem as large amounts of data have to be moved.
In order to provide industrial-strength experimentations, experiments have been run using the Claude Bernard database of french medicaments [PV83] containing nine relations. The relations cardinality varies from 5200 to 10000 tuples, whereby the relation size comes up to 1MByte. The schema of the database allows up to four possible join combinations, with a join selectivity varying between 0.00036 and 0.00116.

This section is based on the comparison between our approach and various alternative methods. While these latters usually benefit from a hash-partitioning of the relations, our approach is particularly well adapted to range-partitioning. Tests have been performed with the optimal method for each case.

Fig. 3 (left scheme) compares our approach with a classical processing without load balancing. Experiments were made on an iPSC860. Submitted queries consist in three-way joins in which two joins are performed in parallel (*bushy tree processing*), or one after the other (*left linear tree processing* ; see section 2). No optimizer estimation errors are detected, so the processor allocation is not modified.
Through all experiments (see fig. 3 left scheme), our method achieves better execution times. The gain is up-to 14 % for 8 processors, when using left linear tree processing. Furthermore fig. 3 (right scheme) shows that the over-costs are extremely small in comparison with the gain.

Next experiment compares our load balancing technique with an efficient classical technique used for correcting redistribution skew [NS92]. Experiments are now performed on the Paragon machine.

Fig. 4 compares our approach (**Join Product Skew (JPS) correction and processor reallocation**) with a classical dynamic method for correcting redistribution skew (**Redistribution Skew (RS) correction**) [NS92].

When no query optimizer errors occur (i.e. when no processor reallocation is triggered), execution times are similar (fig. 4 left scheme, two-way join).

Consider now the case in which the compile-time optimizer chooses a non-optimal processor allocation. The submitted query is shown in fig. 5. It contains seven joins, executed in the bushyiest possible manner. After processing the first join ($R_1 \bowtie R_2$), it is detected that the intermediate result relation T (see fig. 5) produces two times more tuples than estimated by the compile-time optimizer.
Fig. 4 right scheme shows that our method (triggering a processor reallocation) outperforms the classical method whether 8 or 16 processors are used. Thus for 16 processors, we achieve a gain of 18 %.

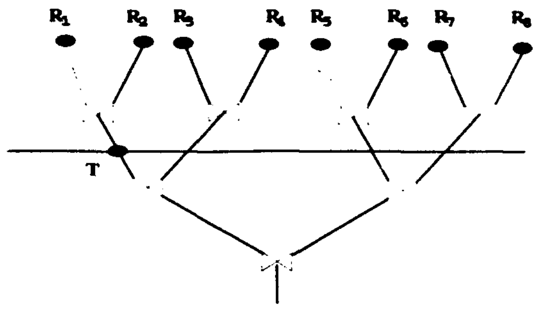

Figure 5: Experiments performed with optimizer estimation errors : the submitted bushy processing tree.

6 Discussion

Previous results for 8 (16) processors show the effectiveness of the proposed control strategies. All experimental results show that an efficient detection and correction of optimizer estimation errors and load imbalance is necessary to ensure good performances. Consider now the different theoretical and experimental issues :

The first noticeable point is the excellent efficiency that is achieved (see fig. 3 and 4). Moreover, redistribution over-costs are extremely small (see fig. 3), as we overlap communication and computation each time it is possible:

Overlapping of elastic redistribution and following relational operations :

Consider once again the fig. 1. Every processor holds some tuples which will remain on that processor. Consequently, while performing the data distribution, one can already apply the following relational operation to these tuples.

Overlapping of local load balancing with global reallocation measures :

Once all processors performing a same operation have terminated, a local load balancing can be started without waiting for the global termination of all the operations running in parallel.

Experiments have shown that reallocating the processors is a very efficient technique for correcting relation size estimation errors of the optimizer. Though various load balancing methods had previously been proposed [LT94, NS92, YT91], it is the first time that a global detection, load balancing and reallocation strategy is introduced in the framework of shared-nothing systems. This strategy allows reacting in real-time before a processor gets idle.

Finally, in previous experiments we have supposed that all relations can fit in memory (i.e. that no swapping had to be done at run time). This is obviously not true when working on very large databases. Anyway, taking the swapping into consideration should not affect the basics of the methodology developed in this paper since it has been designed to work on relation buckets (see section 4.3). In that framework, we are currently making experiments in order to precisely estimate the efficiency of the overlapping of load balancing with data retrieval from disk.

7 Conclusion and future works

This paper has described an original methodology for the control of complex relational query processing in shared nothing systems. In comparison with previous approaches, this formalism gives a complete methodology for detecting and correcting (via data redistribution and processors reallocation) critical query processing situations.

Experiments on an Intel iPSC860 and Paragon have shown the efficiency of this approach, especially the correction of the degree of intra-parallelism. This gives the key to an efficient management of optimizer estimation errors.

The implementation of this formalism on a 128 processors hypercube is currently under development.

References

[BK95] L. Brunie and H. Kosch. A communication-oriented approach to parallel relational query optimization and processing. Research Report 95-32, LIP-ENS Lyon, November 1995.

[CB91] P. Valduriez M. Couprie and B. Bergstein. Prototyping DBS3, shared-memory parallel database system. In *Proceedings of the 1st International Conference on Parallel and Distributed Information System*, Miami Beach, Florida, December 1991.

[HM94] W. Hasan and R. Motwani. Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism. In *Proceedings of the International Conference on Very Large Databases*, pages 36–47, Santiago, Chile, September 1994.

[Hon92] Wei Hong. Exploiting Inter-Operation Parallelism in XPRS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 19–28, USA, June 1992.

[IC91] Y.E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Denver, USA, 1991.

[Inf94] Informix. INFORMIX-OnLine Dynamic Server (TM) - Technical Brief. 1994.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computer Surveys*, 16(2), June 1984.

[LT94] H. Lu and K.L. Tan. Load-Balanced Join Processing in Shared-Nothing Systems. *Journal of Parallel and Distributed Computing*, 23:382–398, 1994.

[MD95] M. Mehta and D. DeWitt. Managing Intra-operator Parallelism in Parallel Database Systems. In *Proceedings of the International Conference on Very Large Databases*, Zurich, Switzerland, September 1995.

[ME92] P. Mishra and M.H. Eich. Join Processing in relational databases. *ACM Computing Surveys*, 24(1), March 1992.

[MR91] S. Miguet and Y. Robert. Elastic load balancing for image processing algorithms. In H. P. Zima, editor, *Parallel Computation*, Lecture Notes in Computer Science, pages 438–451 Salzburg, Austria, September 1991. 1st International ACPC Conference, Springer Verlag.

[MW94] G. Graefe R.L. Cole D.L. Davison W.J. McKenna and R.H. Wolniewicz. *Extensible Query Optimization and Parallel Execution in Volcano*, page 305. Query Processing for Advanced Database Applications. Morgan Kaufman, San Mateo, CA, 1994.

[NS92] D. Schneider D.J. DeWitt J. Naughton and S. Seshardi. Pratical skew handling in parallel joins. In *Proceeding of the International Conference on Very Large Databases*, Vancouver, British Columbia, August 1992.

[PV83] A. Flory C. Paultre and C. Veilleraud. A relational databank to aid in the dispensing of medicines. In Ball Van Bemmel and Wigertz, editors, *Proceeding of MEDINFO 83*, Amsterdam, North Holland, 1983.

[RR95] E. Rahm and R.Marek. Dynamic Multi Resource Load Balancing in Parallel Database Systems. In *Proceedings of the International Conference on Very Large Databases*, Zurich, Switzerland, September 1995.

[SD90] D. Schneider and D.J. DeWitt. Tradeoffs in processing complex join queries via hashing in multi-processor database machines. In *Proceedings of the International Conference on Very Large Databases*, Melbourne, Australia, August 1990.

[TW94] C. Mohan H. Pirahesh W.G. Tang and Y. Wang. Parallelism in relational database management sytems. *IBM Systems Journal*, 33(2), 1994.

[Val93] Patrick Valduriez. Parallel database systems: open problems and new issues. *Distributed and Parallel Databases - an International Journal*, 1(2):137–165, 1993.

[VZ94] R.S.G. Lanzelotte P. Valduriez and M. Zaït. Industrial-Strength Parallel Query Optimization: Issues and Lessons. *Information Systems - An International Journal*, 1994.

[YT91] J. Wolf D. Dias P. Yu and J. Turek. An effective algorithm for parallelizing hash joins in the presence of data skew. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 200–209, Kobe, Japan, April 1991.