

A DYNAMIC APPROACH FOR HANDLING DATA SKEW PROBLEMS
IN PARALLEL HASH JOIN COMPUTATION

Xiaofang Zhou and Maria E. Orlowska
Department of Computer Science
The University of Queensland, St Lucia, QLD 4072, Australia

ABSTRACT

Data skew can result in load imbalance in parallel hash join. We introduce a new term, *solid data skew*, and classify data skew into plane and solid categories in this paper. The existing algorithms consider plane data skew only thus not adequate to approach solid data skew problem. A dynamic approach for handling solid data skew problem is proposed. It assigns join subtasks to processors during runtime using a method based on the estimation of execution time required for the computation.

INTRODUCTION

The multiprocessor environment of distributed database systems provides the potential to execute database operations in parallel. The current trends in database applications indicate that relational queries are becoming more complex and relations are growing larger. It has attracted extensive investigations to explore parallelism in distributed database systems. As join is a time-consuming, yet frequently-used database operation, parallel join algorithm design is one of the most active research areas in parallel database research.

Hash join algorithms can map the relations to be joined into a number of horizontal fragments (*buckets*) in linear time such that a pair of buckets from each relation can be joined independently, i.e., without referencing any other buckets. It is potentially efficient for a parallel hash join algorithm to produce such buckets and distribute them to processors where the joins can be computed locally [5]. There exist a considerable number of hash-based parallel join algorithms [1, 4, 5, 6, 8, 10, 12]. Similarly to hash conflicting and overflowing problems in sequential hash join algorithms, there are some factors to be handled to improve the performance of parallel hash join algorithms. For example, as the response time is determined by the slowest processor involved in parallel join processing, the fluctuation of bucket sizes (in terms of the number of tuples in them) has to be smoothed to avoid the situation that some processors assigned with larger buckets are still busy while those light-loaded processors are idle. Nonuniform data distribution, referred to as *data skew*, can result in such problems of load-imbalance. To keep load-balancing even in the presence of data skew is the key issue for parallel hash join algorithm design [4, 6, 7, 8, 10, 12]. The GRACE join approach [5] maps the relations into many small subbuckets and then pack them into equal-sized buckets of the number of available processors. [6] proposes storing each bucket across the network distributively, instead of at a

specific node where it will be joined, and also proposes to rebuild equal-sized buckets after network-wide *bucket turning*. This communication-intensive procedure is supported by a specially-designed hardware Ω network. [4] suggests three ways to get equal-sized buckets within a general-purpose multiprocessor environment after obtaining many small subbuckets as in the GRACE join algorithm. [12] uses hierarchical hashing and adds a scheduling phase between the hash mapping phase and the local join phase to even up bucket sizes. These efforts provide efficient methods to smooth bucket sizes before the local join computation on each processor begins.

As we will point out in this paper, data skew may occur either before or after hash mapping, as well as in the join result. We name the data skew associated with hash mapping as *plane data skew*. It is a special case of the *solid data skew* which additionally takes into account the data skew in join result. To the best of our knowledge, all the previous work, which tries to achieve equal-sized buckets before joining, attempts to handle only plane data skew problems. As solid data skew is usually hard to predict before joining relations and can only be fully specified when a join is computed, thus it is impossible to maintain load balancing by assigning tasks to processors prior to join commencement. That is, *rebalancing* during join computing is necessary. This paper proposes a dynamic load balancing algorithm for solid data skew problems in parallel hash join operations. It assigns join subtasks to processors during runtime based on the estimation of execution time in the presence of data skew. The impreciseness of the estimation will be compensated by a dynamic schedule strategy.

DEFINITIONS AND ASSUMPTIONS

Let R and S be two relations with nondisjoint relational schemas. To join relations R and S , written as $C = R \bowtie S$, each tuple in one relation R is examined against all the tuples in relation S , and the join result C is generated by matching corresponding tuples. We only consider the equi-join operations on a single attribute. Without loss of generality we assume that a tuple is restricted to the field of the join attribute, thus a relation is represented by its join column only. The size of a relation R is the number of tuples in R , denoted as $|R|$. A tuple of R is represented as t .

Let $P = \{p_1, p_2, \dots, p_n\}$ be a set of processors to be used for parallel join computation. Assume that all these n processors are identical and interconnected. We do not distinguish the communication cost between each pair of processors. One

may assume that they are fully connected or connected by a bus. The network transmission rate is assumed to be very high, so to distribute a pair of buckets (R_i, S_i) to an under-utilized processor to perform the join is always profitable provided that the granularity of the buckets is well-controlled.

We assume also that a pre-defined hash function f on the join attribute is available. Its image domain is partitioned by a set of boundary elements, which form an increasing $(m+1)$ -elementary sequence $q(f, m) = (l_0, l_1, \dots, l_m)$ where l_0 and l_m are always defined as $-\infty$ and ∞ respectively.

A *trivial partition* of R is a set of fragments R_i , $1 \leq i \leq m$ such that $R = \bigcup_{i=1}^m R_i$ and $R_i \cap R_j = \emptyset$ for $i \neq j$. A *bucket partition* of R , obtained by *bucket partitioning* with a hash function f and a boundary element sequence $q(f, m)$, is a set of fragments R'_i , $1 \leq i \leq m$ such that $R'_i = \{t \in R : l_{i-1} < f(t) \leq l_i\}$. R'_i is called a *bucket*. Bucket partition is a special case of trivial partition. It is obvious that $R = \bigcup_{i=1}^m R'_i$ and all buckets are pairwise disjoint.

Data skew can be measured by skew degree β . Let $\pi : X \rightarrow Y$ be a function where $X = \{x_i : 1 \leq i \leq \ell\}$, $Y = \{y_i : 1 \leq i \leq p\}$, and $p \leq \ell$. For $y \in Y$, define $\varphi(y) = \{x \in X : \pi(x) = y\}$. Data skew is measured by data skew degree $\beta(\pi, X)$:

$$\beta(\pi, X) = \max_{y \in Y} \frac{|\varphi(y)|}{|X|}$$

It is obvious that $1/p \leq \beta(\pi, X) \leq 1$. If X is evenly mapped into Y , i.e., with no data skew, then $\beta(\pi, X) = 1/p$. Otherwise, $\beta(\pi, X) > 1/p$. $\beta(\pi, X) = 1$ means all data in X are skewed in a $y \in Y$.

By saying some data are skewed in a $y \in Y$, we mean that the elements in $\varphi(y)$ are not distinguishable by π , and there are no, at least we do not have at hand, any fine function to provide distinction between them. To run a parallel hash join algorithm on n processors, we need a bucket partition consisting of at least n buckets for relation R and S respectively. If $\beta(f, R) > 1/n$ or $\beta(f, S) > 1/n$, it is not always possible to keep load balancing, because the largest data skew, which can not be split by f , may already make a processor overloaded. As in GRACE join algorithm, we consider mild data skew only. That is, we assume that both $\beta(f, R) \leq \alpha/n$ and $\beta(f, S) \leq \alpha/n$, where α , $0 < \alpha \leq 1$, is a system parameter used to determine whether a parallel hash join algorithm is to be employed. Under this assumption, the bucket partitioning will generate more than n/α pairs of subbuckets which gives room for load balancing among n processors.

Other assumptions about join operations are: (1) Relations to be joined are large enough, so it is worth while distributing the join operation among the n processors; (2) The join result is pretty large (otherwise solid data skew reduces to plane data skew); (3) No index on the join attribute is available; (4) The final result is not required to be sorted on the join attributes. The parallel hash join procedure will not be invoked if a pre-join check finds that these conditions do not hold, and then some other parallel join algorithm, or even sequential join algorithm, can be employed.

PARALLEL HASH JOIN AND DATA SKEW

1. Parallel Hash Join Algorithms

To join R and S using a set of processors P in parallel, we

first need to distribute these two relations to each processor $p_i \in P$. To avoid a heavy communication penalty during join computation, two kinds of data distribution patterns can be used, as illustrated in Figure 1. We estimate the time of a

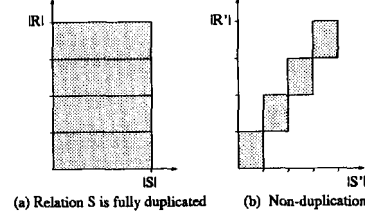


Figure 1: two basic data distribution patterns

join operation as the product of the sizes of its two operand relations. Thus a rectangle in Figure 1 is the work-load on a processor, or the parallel computation cost of the join, while the sequential computation cost is represented by the union of all the shaded rectangles. In Figure 1(a), relation S (usually the smaller one) is fully duplicated on all processors, and relation R is assigned to the processors using a trivial partitioning method. It is easy to see that to duplicate both operand relations on a processor is not necessary. Figure 1(b) shows a data distribution pattern without duplication. From Figure 1 one can see that the non-duplication pattern reduces both parallel and sequential computation cost. However, bucket partitioning must be used here. To map R and S to R' and S' respectively, sorting these two relations before distributing [11], or hash mapping are necessary. Hash mapping is normally a linear procedure ($O(|R| + |S|)$) and it is easy to parallelize, while sorting needs $O(|R| \log |R| + |S| \log |S|)$ time and it is not simple to parallelize. So hash mapping is widely used in parallel join algorithms.

A typical hash-based parallel join algorithm can be outlined as follows. Recall that $n = |P|$.

Algorithm1: parallel hash join algorithm

Stage 1 data distribution: Partition both R and S by trivial partitioning into n equal parts. Each p_i , $1 \leq i \leq n$, receives R_i and S_i .

Stage 2 partition: Each processor p_i , $1 \leq i \leq n$, maps R_i and S_i using the same f and $q(f, n)$ into n subbuckets R_{ij} and S_{ij} , $1 \leq j \leq n$, respectively.

Stage 3 data exchanging: Each processor p_i collects all R_{ji} and S_{ji} , $1 \leq j \leq n$, $1 \leq i \leq n$, from other $n-1$ processors; $R'_i = \bigcup_{j=1}^n R_{ji}$ and $S'_i = \bigcup_{j=1}^n S_{ji}$.

Stage 4 independent join: Each processor p_i computes $C_i = R'_i \bowtie S'_i$, $1 \leq i \leq n$.

Stage 5 result assembling: All resultant tuples on n processors are collected and returned to the processor where the final result $C = \bigcup_{i=1}^n C_i$ is required.

The first three steps of Algorithm1 perform bucket partitioning by hashing in parallel. The response time of this algorithm is determined by the processor which finishes last. To improve the performance (response time) of Algorithm1-like

parallel join algorithms, it is imperative to keep load balancing among all the processors such that they can terminate at almost the same time.

2. Data Skew Classification

As join may occur on any attribute, and sometimes the join relations are the intermediate results from some previous operations, it is not easy to find a suitable elementary sequence $q(f, m)$ such that all the buckets are equal or nearly equal in size before they are actually generated. And also, it is not uncommon in database applications that some values in the joining column appear more frequently than others. The tuples of the same value will be mapped into the same bucket, no matter what f or $q(f, m)$ will be used. Data skew arises when there are more than expected data being put into a place. In hash-based join algorithms, data skew can cause load-imbalancing, and consequently it affects response time because the number of buckets and then the sizes of buckets may not be chosen in an optimal way.

Figure 2 depicts five general stages of a typical parallel hash join algorithm. Data skew may occur at each stage and in one of (*single skew*) or both of (*double skew*) the two operand relations. While single skew is not easy to handle, double skew is even harder to approach. The factors causing data skew

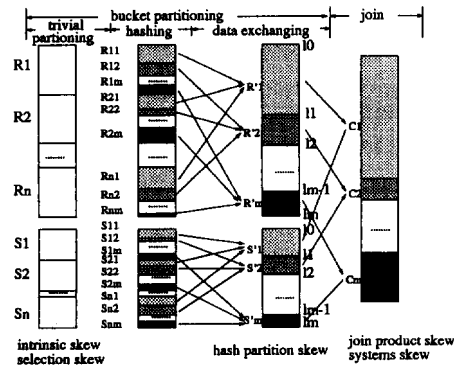


Figure 2: parallel hash join with data skew

have been studied in several research papers, and their classification can be found in [10]. More specifically, data skew can be classified into five classes. (a) *Intrinsic skew* This kind of data skew occurs when attribute values are not uniformly distributed. This situation often occurs in practice. It makes impossible to divide relations evenly into buckets by choosing a well-designed hash function because the tuples of the same value are always mapped into the same bucket regardless of what hash function is in use. (b) *Selection skew* Because of performance considerations, the join operations are usually postponed to as late as possible by query optimizers. These preceding operations could create two problems: data distribution in the intermediate relations is hard to predict after selections and projections, and there may not be statistical information available about the relations to be joined. This kind of skew amplifies the negative effects of intrinsic skew. (c) *Hash partitioning skew* This is caused by the possible mismatch between the distribution of join attribute values and the expected distribution by the hash function.

(d) *Join result skew* Join selectivity [9] may vary from bucket to bucket. Two buckets with greater join selectivity may generate a larger join result and thus need longer time for generating output (I/O), communication (when returning results) and computation (if there are some other operations on the output, like eliminating duplicates). This means that even when the operand relations are distributed among the processors evenly, each processor still requires a different time to compute the join, i.e., load is still imbalanced; If there is not enough precise statistical information, this kind of skew is unpredictable. (e) *System skew* Computational capabilities of the processors to be involved in parallel joins may be different in a distributed system. This uneven computational capability requires the bucket sizes be proportional to the capabilities of computers rather than just being equal. Generally, how many and which computers can be used for a join operation can vary for each execution of an algorithm, so an adequate $q(f, m)$ must be determined according to the current system configuration. With incomplete knowledge of data distribution of the operand relations, it is not easy to choose such a $q(f, m)$. Inadequate bucket sizes will introduce "system skew", which means some computers are assigned nonproportional tasks. In a typical distributed system, some processors may become idle and then they can be used during the join. Moreover, some processors which are expected to take part in the join computation (or even have been already assigned some subtasks of the join) may be required to quit to respond to their owners or because of some failure. This introduces dynamics into P , and then data skew also becomes time-dependent, i.e., balancing on a current P will be violated when P changes. In this paper we assume an even and fixed computing environment. So system skew is not considered here.

3. Plane And Solid Data Skew

When data skew occurs, bucket partitioning will generate buckets of the sizes as pictured in Figure 3(a), while what one expects to get is shown in Figure 3(b). We refer to data skew, which causes fluctuation of bucket sizes, as *plane data skew*. The first three kinds of data skew, discussed above, contribute to the plane skew problem, so does the last kind of data skew when in an environment with uneven computational capabilities. Plane data skew is well-recognized, and there exist a number of solutions to handle them [4, 6, 7, 10, 12]. Most of them rely on the design of a procedure (which is either decentralized or centralized) to map the situation shown in Figure 3(a) to that in Figure 3(b). We classify this kind of attempt as *static* approaches. However,

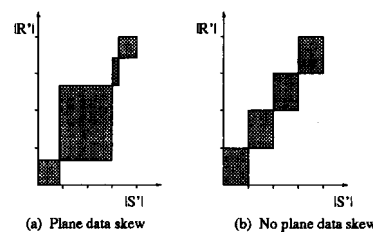


Figure 3: plane data skew

the fact that there is no plane data skew does not guarantee load balancing. Assume that a comparison of two tuples takes a unit of time, and to generate an output takes μ unit of

time, then to join a pair of buckets R'_i and S'_i by nested-loop sequential join algorithm needs time:

$$\tau = |R'_i| \cdot |S'_i| + \mu |R'_i \bowtie S'_i|$$

The first part of the τ expression can be reduced when a more efficient sequential join algorithm is used. For two pairs of buckets (R'_i, S'_i) and (R'_j, S'_j) , where $|R'_i| = |R'_j|$ and $|S'_i| = |S'_j|$, which is the ideal case for all static approaches to expect, the time to join each pair may still be different as the sizes of the join results, $|R'_i \bowtie S'_i|$ and $|R'_j \bowtie S'_j|$ may be different. Because the join result is large as we assumed, and μ cannot be neglected in distributed environment (it includes not only I/O operations but also communication cost), the time differences to join equal-sized buckets have to be taken into account. Moreover, data skew in the operand buckets will make the time differences even bigger. Figure 4 illustrates such an example. R and S in Figure 4 have the same domain $(0, 9)$, their value distribution is nonuniform. The hash function expects an even distribution so it uses the boundary element sequence $(-\infty, 4, \infty)$. Figure 4(a) shows the result after hash partitioning, data skew occurs in both R'_2 and S'_2 . Then one can adjust them, and get even buckets shown in Figure 4(b). The join result $C_1 = R'_1 \bowtie S'_1$ has only 2 tuples, while $C_2 = R'_2 \bowtie S'_2$ has 8 tuples. This example shows that a pair of buckets with data skew will produce much larger join result than those of the same size without data skew. Thus join result skew can affect join execution

R ¹					R ²				
R'	1	3	4		6	7	9	8	7
S'	0	2		5	8	7	6	7	
S ¹					S ²				

(a) two pairs of buckets with plane data skew

R ¹					R ²				
R''	1	3	4	5	6	7	9	8	7
S''	0	2	5	6	8	7	8	7	
S ¹					S ²				

(b) two pairs of buckets without plane data skew

Figure 4: example of solid data skew

time significantly. We define data skew in both two operand relations and join result as *solid data skew*. Plane skew is just a special case of solid skew when there is no join result skew. Solid data skew occurs when the join result is not very small and the time to process join output cannot be neglected. It can be caused by the skews in input relations of a join operation. It also occurs when the configuration of processor set P is allowed to change dynamically, as working load will be relatively skewed if there are some processors joining or quitting. Solid skew is shown in Figure 5.

It is obvious that solid data skew cannot be approached by only static considerations unless a precise cost estimation model and enough accurate profile information are provided. Generally, it is impractical to make assumptions about availability of such information.

A DYNAMICAL SOLUTION

Static approaches to solving plane data skew problems are basically concerned with achieving load balancing before the local join phase and then never again during run time, so they cannot tackle solid data skew arising from local join phase. We propose a new algorithm which balances the load during join phase (see Figure 2) dynamically, to minimize the response time of a join.

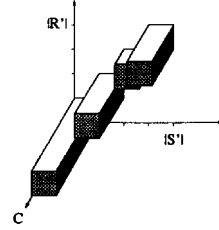


Figure 5: solid data skew

Algorithm 2: dynamic parallel hash join

Stage 1 data distribution: Partition both R and S by trivial partitioning into n equal parts. Each p_i receives R_i and S_i , $1 \leq i \leq n$.

Stage 2 partition: Each processor p_i maps R_i and S_i using the same f and $q(f, m)$ into m subbuckets R_{ij} and S_{ij} respectively, $1 \leq i \leq n$, $1 \leq j \leq m$ and $n < m$. It reports $\{|R_{i1}|, |R_{i2}|, \dots, |R_{im}|\}$ and $\{|S_{i1}|, |S_{i2}|, \dots, |S_{im}|\}$ to a designated processor, say p_0 . p_0 gets global partitioning information:

$$|R'_j| = \sum_{i=1}^n |R_{ij}|, |S'_j| = \sum_{i=1}^n |S_{ij}|$$

Stage 3 parallel join: Each processor p_i , $1 \leq i \leq n$, executes the following steps:

Stage 3.1 task acquisition: Processor p_i asks for a task from p_0 . If there is no task left, p_i terminates. Otherwise, the "largest" remaining task $R'_j \bowtie S'_j$ is assigned to p_i .

Stage 3.2 bucket collecting: Processor p_i broadcasts the task it has just obtained to all the other processors in P , and collects R_{kj} and S_{kj} , $1 \leq k \leq n$, from other $n-1$ processors to get $R'_j = \bigcup_{k=1}^n R_{kj}$ and $S'_j = \bigcup_{k=1}^n S_{kj}$.

Stage 3.3 independent join: Processor p_i computes $C_j = R'_j \bowtie S'_j$ and goes to Stage 3.1 when finished.

Stage 4 result assembling: All resultant tuples on n processors are collected and returned to the processor where the final result $C = \bigcup_{i=1}^n C_i$ is required.

Instead of assigning all the tasks to be processed on a processor before join, which is unavoidably load-imbalancing as we have indicated by previous analysis, Algorithm2 assigns tasks only to the idle processors. In the presence of unknown solid data skew the work-load of processor p_i can be underestimated, and therefore it takes a longer time. Subsequently, it will then be given a smaller task since other earlier-finished processors have requested the larger tasks from p_0 . Therefore, the difficulties in work-load estimation will be compensated for by the order to acquire the next task. This order is determined by the actual work-load. In Stage 3.1, p_0 always gives the currently largest task to the requiring processor. "Largest" task is the join of a pair of buckets which needs longest time to process. That is, in Algorithm2 LPT (longest processing time) scheduling strategy [2, 3] is used as a heuristic rule to balance the load among processors. In fact, LPT

scheduling strategy is also used by some parallel join algorithms of static category [11]. The algorithms which generate equal-sized buckets by packing subbuckets to overcome plane skew problem also face the similar difficulties to achieve precise load balancing. In general, to pack n tasks into m groups optimally, known as *bin-packing* problem, is a NP-complete problem [3].

To use LPT scheduling strategy, the work-load of each task is to be known in advance. As solid data skew cannot be totally predicted before the join is finished, the work-load of a join task is not easy to determine by the traditional method. We use a rule of thumb to predict it. The number of buckets, m , is given before bucket partitioning, the expected size of each buckets of R and S are $|R|/m$ and $|S|/m$ respectively. To estimate execution time, τ_j , of joining a pair of buckets R'_j and S'_j , this formula is in use:

$$\tau_j = |R'_j| \cdot |S'_j| \cdot (1 + \frac{|R'_j|}{|R|} \cdot m) \cdot (1 + \frac{|S'_j|}{|S|} \cdot m)$$

We do not attempt estimate the size of $R'_j \bowtie S'_j$ by this formula. What one needs to know for load balancing is only which task is the biggest one. In other words, we just need to know the relative order of work-load of these tasks. This estimation is based on the observation that join selectivity is related to data skew and that double data skew can potentially produce larger output.

Unlike Algorithm1, Algorithm2 does not include *data exchanging* step. Subbuckets are collected when the corresponding bucket is assigned. By doing so it avoids the "traffic jam" in Stage 3 of Algorithm1, which is caused by all processors sending all subbuckets simultaneously. However, the processors performing join computations in Stage 3.3 of Algorithm2 may be interrupted for a while by the processor which just comes to Stage 3.2 of Algorithm2.

CONCLUSIONS

Load-balancing is a critical issue to the performance of parallel hash join algorithms. We have shown that even without plane data skew, the problem of load-imbalance still frequently occurs in applications. To model this phenomenon, a new notion, solid data skew has been introduced. With previous static solutions, solid data skew problems are impossible to overcome. We have presented a dynamic approach to handle data skew problem which is based on scheduling tasks during join execution. In this paper we have presented a heuristic rule to dynamically assign tasks to processors according to the task execution time. We have also shown how to estimate the execution time of a join in the presence of data skew.

References

- [1] D. J. DeWitt and et. al. Implementation techniques for main memory database systems. In *ACM SIGMOD 84*, pages 1-8, 1984.
- [2] R. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Computing*, 17:416-429, 1969.
- [3] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.

- [4] Kien A. Hua and Chiang Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proceedings of 17th International Conference on Very Large Data Bases*, pages 525-535, Barcelona, 1991.
- [5] M. Kitsuregawa, H. Tanaka, and T. Motooka. Application of hash to database machine and its architecture. *New Generation Computing*, 1(1):66-74, 1983.
- [6] Masaru Kitsuregawa and Yasushi Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *Proceedings of 16th International Conference on Very Large Data Bases*, pages 210-221, Brisbane, Australia, 1990.
- [7] M. Seetha Lakshmi and Philip S. Yu. Limiting factors of join performance on parallel processors. In *Proceedings of 5th International Conference on Data Engineering*, pages 488-496, Los Angeles, 1989.
- [8] E. Omiecinski. Performance analysis of a load balancing hash-join algorithm for a shared memory multiprocessor. In *Proceedings of 17th International Conference on Very Large Data Bases*, Barcelona, 1991.
- [9] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.
- [10] Christopher B. Walton, Alfred G. Dale, and Roy M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of 17th International Conference on Very Large Data Bases*, pages 537-548, Barcelona, 1991.
- [11] J.L. Wolf, D. M. Dias, and P.S. Yu. An effective algorithm for parallelizing sort merge joins in the presence of data skew. In *The Second International Symposium on Databases in Parallel and Distributed Systems*, pages 103-115, Dublin, Ireland, 1990.
- [12] Joel L. Wolf, Daniel M. Dias, Philip S. Yu, and John Turek. An effective algorithm for parallelizing hash joins in the presence of data skew. In *Proceedings of 7th International Conference on Data Engineering*, pages 200-209, Kobe, Japan, 1991.