# SKEW-INSENSITIVE JOIN PROCESSING IN SHARED-DISK DATABASE SYSTEMS

Holger Märtens
Department of Computer Science, Database Group
University of Leipzig, Germany

Skew effects are still a significant problem for efficient query processing in parallel database systems. Especially in shared-nothing environments, this problem is aggravated by the substantial cost of data redistribution. Shared-disk systems, on the other hand, promise much higher flexibility in the distribution of workload among processing nodes because all input data can be accessed by any node at equal cost. In order to verify this potential for dynamic load balancing, we have devised a new technique for skew-tolerant join processing. In contrast to conventional solutions, our algorithm is not restricted to estimating processing costs in advance and assigning tasks to nodes accordingly. Instead, it monitors the actual progression of work and dynamically allocates tasks to processors, thus capitalizing on the uniform access pathlength in shared-disk architectures. This approach has the potential to alleviate not only any kind of data-inherent skew, but also execution skew caused by query-external workloads, by disk contention, or simply by inaccurate estimates used in predictive scheduling. We employ a detailed simulation system to evaluate the new algorithm under different types and degrees of skew.

## 1 INTRODUCTION

For high-volume commercial database systems (DBS) such as data warehouses, parallel computation is the key to high performance. To achieve high throughput of transactions as well as low response times for complex queries, both inter- and intra-transaction parallelism have to be employed. The efficient use of parallel processing requires sophisticated load balancing techniques to equalize the workload across processing nodes (PNs). This is especially important for operations that involve large volumes of data, such as the relational join that we will consider in this paper.

In a parallel join, the input data is divided into *buckets* by a hash or range partitioning on the join attribute. The same partitioning is applied for both input relations, and the complete join product is computed by joining corresponding pairs of buckets from both relations and merging the results. The most frequently used method for a *local join* (i. e. the join of a single bucket pair) is the *hash join*. From the bucket of the *inner* relation (usually the smaller one), a hash table is built in main memory against which the contents of the *outer* bucket are probed to find matching tuples. Hash joins have been found superior to nested-loop or sort-merge techniques for most cases (Schneider and DeWitt, 1989).

When a join is executed in a parallel DBS, load balancing must be applied to distribute the buckets evenly across all PNs assigned to the query. One of the main problems that arise in this case is *skew*. This term describes a set of phenomena characterized by an uneven distribution of objects such as data or workload among subjects like disks or processors. Walton et al. (1991) distinguish the following types of *data skew* in the context of parallel join processing:

- *Attribute value skew* (AVS) describes an uneven distribution of values with regard to a certain attribute of a relation. AVS is a property of the data and independent of any algorithmic features.
- *Partition skew* (PS) is present when a data set (either a base relation or a join bucket) is fragmented unevenly across disks or processing nodes. It is often a result of AVS and/or an inappropriate partitioning method (e. g. an unfair hash function) and occurs in four common forms:
  - *tuple placement skew* (TPS), where a base relation is unfairly distributed, so that local scan operations executed in parallel work on different amounts of data;
  - *selectivity skew* (SS), characterized by different selectivities between sub-scans;
  - *redistribution skew* (RS), caused by different amounts of data being routed from the scan processors to the various join nodes;
  - *join product skew* (JPS), meaning unequal result sizes among sub-joins.

A frequent consequence of the various types of data skew is *execution skew* (ES) which describes the disparate execution times between sub-queries that lead to suboptimal response times.

AVS is most commonly modelled as a so-called *Zipf-like distribution* in which the $i$-th most frequent value occurs

$$f(i) = |R| \Big/ \left( i^s \cdot \sum_{j=1}^{|D|} \frac{1}{j^s} \right)$$

times. Here, $|R|$ is the number of tuples in relation $R$ and $|D|$ is the domain size on the attribute in question; $s$ (non-negative) is the degree of skew. For $s = 0$, the distribution is uniform, whereas $s = 1$ is considered severe skew. Distributions of this type have been found appropriate for many examples of real-world data (Wolf et al., 1993).

Since AVS is an inherent property of the data, it cannot be avoided. However, the other types of data skew and also execution skew can be managed to a certain extent by intelligent allocation and scheduling techniques. It has been argued that *shared-disk* (SD) architectures offer a higher

potential for load balancing than *shared-nothing* (SN) systems do (Rahm, 1993 and 1996). This is because in SD environments, each PN can access any data at uniform cost so that expensive data redistribution – as necessary in SN – can be avoided. For scan operations, this concerns the base relations which can always be scanned by the least loaded processors. For joins, hash buckets can be stored on any disk that has free capacity and bandwidth, to be retrieved later by any node with enough memory and processing power available.

Several commercial database systems such as Oracle and DB2/MVS have successfully employed the shared-disk approach. Still, few academic researchers have tried to explore its load balancing potential in the context of join processing. In an attempt to fill this gap, we will examine in this paper very large join queries on shared-disk systems as they frequently occur in decision support environments. Such queries often process millions of tuples at a time, thus representing a class of challenging load balancing problems.

In the following, we will first discuss some of the most popular approaches to join load balancing. We will then describe a new algorithm we have developed based on an idea we have termed "on-demand scheduling". This algorithm has been tested in a newly developed simulation system that we will lay out before we present some of the results it has provided, comparing a traditional "predictive" scheduling method to our new on-demand technique. Finally, we offer our conclusions about the utility of the new approach.

## 2 PREVIOUS SOLUTIONS

While many parallel join algorithms have been presented in the literature, comparatively few of them address the problem of skew handling, and those that do are usually restricted to either shared-nothing or *shared-memory* (SM) environments. Furthermore, although multi-user operation is a prerequisite of modern decision support systems, many studies do not consider the special problems that stem from inter-query parallelism. Still, we will briefly review some of the basic concepts proposed and evaluate them with respect to their load balancing potential. Later, we will try to transfer the main ideas to multi-user SD systems.

Hua and Lee (1991) investigated *partition tuning* methods to compensate skew effects. With regard to partition skew, they differentiated *skew resolution* (coping with unequal partitions during processing) and *skew avoidance* (equalizing buckets before they materialize). They found it useful to delay the transfer of data between nodes until bucket sizes are known and an allocation to PNs has been done. They also advocated *bucket tuning*, during which small buckets are merged for better memory utilization.

DeWitt et al. (1992) compared five different hash-based algorithms under non-uniform value distributions. They employed the simple model of *scalar skew*, characterized by a single very frequent value and a uniform distribution among all others. They stated that the best choice of algorithm depends on the degree of skew within their data. For a universal solution, they suggested to scan a sample of the input and choose the join technique based on the findings. For high skew, they proposed *virtual processor partitioning*, which means dividing the input into a large number of buckets and allocating several of them to each processor.

Wolf et al. (1993 and 1994) suggested *hierarchical hashing* – using a coarse and a fine hash function – to achieve a fine granularity in which to schedule sub-tasks. For very large bucket pairs, they applied a *fragment/replicate* scheme which divides the inner bucket among several processors and broadcasts the outer bucket to all of these nodes. The remaining bucket pairs are scheduled using the *LPT* (*longest processing time first*) policy. It assigns tasks to PNs in descending order of estimated cost, giving each task to the node that so far has the least accumulated work. LPT is a simple heuristic but has been shown to provide very good results for so-called *minimal makespan* problems. Although the authors assume the more realistic Zipf-like value distribution in general, their cost estimates are based on scalar skew within each bucket.

Poosala and Ioannidis (1996) developed sophisticated histogram techniques for the collection of statistics from which to estimate join result sizes and thus deduce accurate cost functions. In fact, they found a class of histograms that produced minimum errors for a given histogram size and developed a join algorithm to exploit it.

Without explicitly comparing the pros and cons or even the performance of all these algorithms, we note that many important ideas have been brought forth and implemented:

- the differentiation between skew resolution and skew avoidance;
- the minimization of data transfer;
- the avoidance of special-case solutions;
- the advantage of a fine granularity for LPT scheduling;
- the application of fragment/replicate schemes for subjoins that cannot be handled by the LPT algorithm;
- the need for good cost estimates in advance scheduling.

However, all of the join load balancing methods described so far represent a *predictive scheduling* (PS) approach. This means that the distribution of work among the processing nodes is determined before processing actually starts and never changed later on. In our opinion, this method has two important drawbacks:

- In order to distribute buckets evenly among nodes, the workload caused by each bucket has to be estimated, usually based on its size. With little information about the bucket contents, such estimates are notoriously inaccurate, which leads to imbalances in the workload and suboptimal response times. This problem will be aggravated by simplified assumptions about the general distribution of attribute values, such as scalar skew. Better estimates, on the other hand, require additional effort to gather the appropriate metadata, either by pre-join sampling of the input relations or by periodically updated statistics. This extra work can harm throughput and also response times.
- Even if perfect estimates were available for the query under consideration, they would still differ considerably from the actual progression of work. This is because in multi-user mode, other queries – or even DBMS-external workloads – will compete for the same resources and interfere with the schedule so carefully developed for the

query in question. Such interferences are inherently unpredictable because new queries can arrive at any time. The only way to avoid this problem would be to delay incoming queries, in effect putting the system into single-user mode which is, of course, unacceptable. As mentioned earlier, this aspect has been neglected in most studies published so far, including all those cited above.

For these reasons, predictive scheduling is inherently suboptimal in many situations. Some researchers have addressed this problem and suggested ways of reacting to skew effects rather than predicting or controlling them:

Harada and Kitsuregawa (1995) developed an algorithm that computes a predictive schedule as above, but monitors the progress of its execution. When a substantial deviation from the schedule is detected, the system compensates by relocating some of the workload between PNs. It uses either *result redistribution* (writing result tuples to another node's disk to lighten the load on the local one) or *process task migration* (switching to a fragment/replicate scheme by transferring parts of the input data to a remote node). In an SN environment, this requires a lot of communication so that, essentially, disk or CPU load is rebalanced between nodes at the price of additional network traffic.

Bouganim et al. (1996) applied a similar scheme on a DSM (*distributed shared memory*) architecture. Here, processors share one address space, allowing them to access the same data that is transparently moved through the network when necessary; such hardware support somewhat reduces the redistribution work required by pure SN systems. The authors employ the concept of *activation queues* in which tuples are routed through query operators that can be processed on any node. This approach is even extended to hierarchical systems consisting of DSM nodes in an SN network.

Dewan et al. (1994) have their algorithm compute performance statistics of the participating nodes as the query is processed. The sub-joins are scheduled in batches, and as soon as one node has finished its share of a batch, all processing is stopped except for buckets already begun. The remainder of the batch is then rescheduled based on the performance measures, using a *Weighted LPT* scheme. Buckets are relocated as necessary and processing continues, possibly rescheduled several times until the batch is finished. Then the next batch is started etc. While this algorithm can respond to performance disparities within the system, it may cause multiple relocations for a single bucket. Further, it leaves PNs idle at rescheduling time, when the coordinating node waits for all sites to finish their current buckets.

Zhou and Orlowska (1993 and 1995) present the most promising approach in our opinion. Realizing that join product skew is practically impossible to predict, they dispense with any efforts of advance scheduling. Instead, they allocate sub-tasks to processors one at a time as processing continues. When a processor has finished a bucket, it is given the next one in descending order of size. This also implements an LPT scheme, but it is applied dynamically so that the exact execution times need not be calculated in advance. In fact, it need not even be known what type or degree of skew, if any, is present; all it takes is a rough cost estimate.

Lu and Tan (1992) present the only dynamic scheme for SD systems, extended with a *task stealing* option. However, they require a shared memory segment for communication and ignore the special problems of multi-user mode.

## 3 SCHEDULING ALGORITHMS

Since we consider predictive scheduling inherently suboptimal, we suggest an alternative technique that we have termed *on-demand scheduling* (ODS) and which is similar to the Zhou/Orlowska approach.

The basic idea is not to do any advance scheduling at all. Instead, join buckets are assigned to the PNs involved according to the actual progression of work. Only when a node has finished processing a bucket is it assigned the next one. As a consequence, a node where processing is slowed down for whatever reason will be dynamically assigned less work than others, leading to a more balanced workload.

When Zhou and Orlowska implemented this approach, they were unable to avoid a certain amount of data redistribution during scheduling due to their SN architecture. When the join input relations are hashed into buckets, these buckets must be stored on disk until they are processed. Since it is not previously known which node will process which data, Zhou and Orlowska chose to decluster each bucket across all nodes; the resulting *sub-buckets* must be collected across the network later on. While such redistribution work cannot be avoided completely, it is at least balanced between PNs.

The need for such data shipping has been a main argument of the proponents of shared-disk systems. In SD, all data can be accessed by any processor at equal cost so that load balancing can be achieved without the overhead typical of SN. Wherever a bucket is stored, it can be retrieved directly to its destination without additional effort. Note that, in comparison to SN, this opportunity for data exchange via shared disks comes without any overhead for large queries whose buckets must be stored on disk anyway. For this reason, we chose a shared-disk setting to compare the performance of ODS to that of conventional, predictive scheduling. In short, it is the purpose of this paper to demonstrate that on-demand scheduling is superior to predictive scheduling for high-volume join processing in shared-disk systems.

### 3.1 PROCESSING MODEL

Before we describe in detail the two algorithms that we are comparing in this paper, we shall first discuss the basic processing model as implemented for our study. When a new query arrives from outside the DBS at an arbitrary processing node, this node becomes the *coordinator* in charge of optimization and supervision for this query. The coordinator first determines the degree of parallelism for the query: Starting from the degree of declustering of the base relations, the number of PNs for scan and join as well as the number of disks to hold the buckets are derived through bandwidth considerations such that each operator can process the output of the previous one without delay.

The number of buckets is computed based on a rough es-

timate of AVS (which we assume to be available to the coordinator). It is calculated in such a way that even with the worst possible redistribution skew, the largest bucket of each relation can fit into a single PN's main memory if a hash table is built from it. This leads to a rather large number of buckets, some of which may be very small for heavy skew. On the other hand, it provides a fine task granularity so that the LPT scheme can distribute the load very well.

Theoretically, severe AVS may cause the hash tables of one or more buckets to exceed the main memory of any PN even if a very fine hash function is used that splits partitions down to single domain values. Such *skew buckets* cannot be processed by a local hash join without some sort of overflow management, which may become quite costly because it requires multiple reads on the same data. Instead, we recommend to handle them separately in a fragment/replicate scheme across all nodes as Dewan et al. (1994) did with their *skew pool*. While such a distinction is rather straightforward to implement, we did not include it in our study because this technique is orthogonal to our approach. When skew buckets are processed separately, the remainder of the task is fairly similar to a join of two smaller relations with a lower degree of skew, which actually simplifies load balancing. For this reason, we decided to restrict our experiments to cases where the skew is just soft enough for all buckets to be included in the PS or ODS scheme itself.

When the degree of parallelism has been determined for all stages of the query, the coordinator chooses the least loaded PNs in the system based on utilization statistics. These nodes are assigned parts of the scan. In general, each node will scan multiple disk partitions following the bandwidth calculations above. Different nodes, however, never access the same partition to avoid unnecessary disk contention. The scan output is partitioned into the disk spaces allotted for the buckets. After all sub-scans are finished, the coordinator begins scheduling the join phase as described below. When the join is complete, the query is terminated.

Both PS and ODS employ an optimization concerning the choice of inner and outer relations for the join phase: Conventionally, this decision is made for the relations as a whole, building hash tables from the buckets of the smaller relation and probing those of the larger relation against them. However, we found it preferable to decide on a per-bucket basis instead. This way, when the degree of skew is different in both relations, memory may be used more economically while execution times are largely unaffected. As a consequence, task suspension due to memory shortages in multi-user mode can be avoided or at least reduced to the benefit of response times.

### 3.2 PREDICTIVE SCHEDULING (PS)

For predictive scheduling, the coordinator chooses the least loaded nodes among those that have enough memory available to process the largest bucket. If there are less nodes with sufficient memory than were originally considered appropriate, only the available ones are used because the lack of memory is interpreted as an indication of high system load that should be reacted on by a reduced degree of parallelism. If no processing node has enough memory, the coordinator suspends the query until one or more nodes have sufficient memory available.

Next, the coordinator generates a list of all buckets ordered by their estimated processing cost. We use a cost function from Zhou and Orlowska (1995)

$$c(i) = |R_i| \cdot |S_i| \cdot \left(1 + \frac{|R_i|}{|R|} \cdot n\right) \cdot \left(1 + \frac{|S_i|}{|S|} \cdot n\right),$$

where $R_i$ and $S_i$ are the $i$-th buckets of the relations joined, and $n$ is the number of buckets. It returns a *relative* cost measure for all buckets, i. e. an abstract value proportional to the projected processing time. This is quite sufficient for an approximately even distribution of work, which is done using the common LPT (longest processing time first) policy discussed in chapter 2. (Note, however, that the formula was developed and tested in the context of Zipf-like skew. For other value distributions, it may be less accurate.)

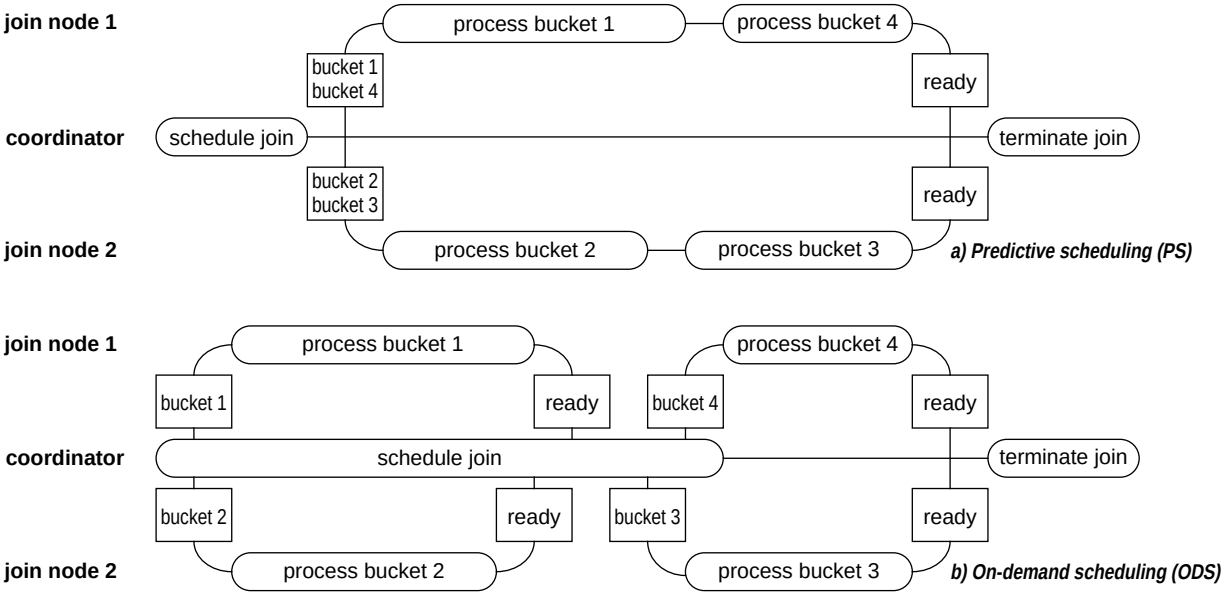
The list of buckets to be processed by each node is packaged as a sub-join task and sent to its node where the buckets are hash joined sequentially in order of size, so that the memory allocated for the sub-join can be released in portions as the task progresses from larger to smaller buckets (and thus from larger to smaller hash tables). The – presumably large – join output is stored on disk in partitions previously allocated by the coordinator, to be retrieved later by the frontend that initiated the query.

### 3.3 ON-DEMAND SCHEDULING (ODS)

For on-demand join scheduling, the coordinator also chooses the least loaded PNs in the system, reducing parallelism or suspending the query if necessary. Again, the buckets are listed in order of estimated cost, using the same formula as for predictive scheduling. Note, however, that the coordinator merely needs the order and does not use the relative cost in any way. Thus, ODS can make do even with very inaccurate estimates – as long as they roughly preserve the correct order – or with extravagant data for which no good cost function can be found. The coordinator assigns exactly one bucket from the top of the list to each processing node involved, then waits until a node reports its task complete. This node will get the next bucket from the list etc. until all buckets have been processed and the join is finished. As mentioned earlier, this is in fact the same LPT algorithm used in predictive scheduling, but applied dynamically as processing advances. The difference between PS and ODS is illustrated in figure 1.

As in the predictive case, memory allocated at the join nodes can be released stepwise as the bucket sizes decrease. Also, the output is stored in the same way as with PS.

Comparing ODS to the list of important concepts collected in chapter 2, we state that it uses skew resolution rather than skew avoidance. It prevents multiple data transfer and schedules in a fine granularity of many small buckets. Most importantly, it is independent of good cost estimates and should perform well for all situations because it automatically adapts to the observed progression of work.




With PS, the entire schedule is computed in advance and transmitted to the join processors for execution. With ODS, join nodes request new work each time they finish a bucket; scheduling overlaps with execution and finishes only when the last bucket has been assigned to a processor.

**Fig. 1 Predictive and on-demand scheduling**

## 4 SIMULATION ENVIRONMENT

In this study, we will use a simulation system to compare the performance of PS and ODS in terms of query response times under different degrees of skew and inter-query competition. We consider these aspects more important than speed-up or scale-up experiments because it is obvious that operations on skewed data do not scale very well. In the following, we will describe the basic simulation environment as well as the parameter settings chosen for our study.

### 4.1 SIMULATION SYSTEM

In our implementation, based on the C++-based simulation package CSIM, we emphasized the aspects of particular relevance to this work. No locking or other synchronization mechanism was needed due to our read-only workload typical of data warehouses.

The data model includes domain sizes, skew degrees and a hash-based declustering for base relations. Queries carry parameters for selectivity, projectivity, result sizes and storage locations. Zipf-like skew of adjustable degree is assumed for the join attributes of all base relations.

The implementation of PS and ODS follows the description in chapter 3. The parallelization and scheduling overhead is identical for both algorithms[1]. Sub-queries are initiated by messages causing further CPU work and a constant delay. We assume a high-speed interconnect and ignore the possibility of network contention.

1. Note that PS and ODS have the same algorithmic complexity, namely $O(n \log n)$ to sort the bucket lists and $O(n)$ to dispatch the tasks.

Our disk model varies access times according to an item's location. Disk contention in multi-user mode is represented by the fact that each disk can serve only one request at a time. An analogous restriction applies to the processors.

During scan processing, each page access leads to a disk read request because we assume full relation scans that – given the large amounts of data – cannot take advantage of a page buffer. The disk controller uses a prefetching mechanism, but there is a small chance that prefetched pages will be purged from the disk cache before they are read. Each request incurs a fixed CPU overhead.

### 4.2 SYSTEM AND WORKLOAD PARAMETERS

As shown in table 1, we model a system with ten PNs and thirty disks; this ratio proved appropriate to alleviate the disk-bound property of our workload. The size of main memory is chosen such that shortages are rare during processing. Only at very high arrival rates do the queries fill the entire memory, causing a reduction of parallelism or suspension of further queries as described in section 3.2.

As we are interested in very voluminous queries as found in decision support environments, we use large relations of up to 400 megabytes and give the initial scans a rather high selectivity of 50 %. This leads to input sizes of 150 to 300 MB for a single join, depending on the base relations used. The join output may be a multiple of this when both base relations are skewed, and the selectivity on this join output is 100 %. The result size for a join of relations *A* and *B* for different degrees of skew is shown in figure 2.

When both inputs in a join are skewed (*double skew*), one must define in which way the value distributions in both

**Table 1 General simulation parameters**

| **Relation** | ***A*** | ***B*** | ***C*** | Number of disks | 30 | Number of PNs | 10 |
|---|---|---|---|---|---|---|---|
| Number of tuples | 500 000 | 1 000 000 | 2 000 000 | Average access time | 14 ms | CPU speed | 20 MIPS |
| Tuple size | 200 B | 200 B | 200 B | Prefetch factor | 8 | Memory per PN | 32 MB |
| Relation size | 100 MB | 200 MB | 400 MB | Buffer purge rate | 0.05 | Network message delay | 10 µs |
| Degree of declustering | 10 | 20 | 30 | Scan selectivity | 0.5 | Join selectivity | 1.0 |
| Domain size | 100 000 | 100 000 | 200 000 | Output tuple size (scan) | 200 B | Output tuple size (join) | 300 B |

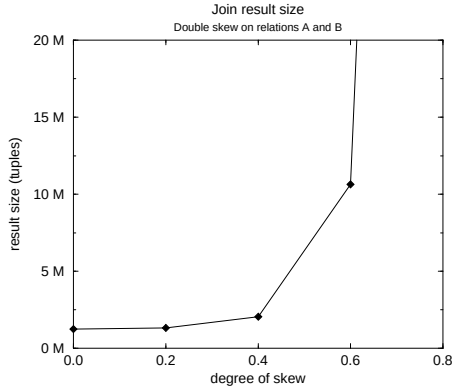



**Fig. 2 Join result sizes under double skew**

relations correlate. We decided to assume a strict correlation, i. e. the *i*-th most frequent values of both relations will match in the join. If the domains have different cardinalities, the "trailing" values of the larger domain will find no match at all. This leads to maximum join product skew and thus to the most challenging scheduling problems.[2]

While we consider only two-way joins in our study, ODS can be used for queries of arbitrary complexity. In fact, queries will probably be more complex in most cases for real-world systems. We chose a simpler setting in our study to obtain a controlled environment in which performance disparities can be more clearly detected and interpreted.

### 4.3 EXPECTED RESULTS

The quality of predictive scheduling hinges on two factors: the accuracy of its cost estimation function and the amount of inter-query competition in multi-user mode. Consequently, we expect ODS to outperform PS in cases of high skew (for which the estimates may be inadequate) and/or high query arrival rates. For low skew in combination with low arrival rates or single-user mode, both techniques should be equivalent. We do not expect PS to be significantly faster than ODS for any situation because in the best possible case, it can base its schedules on perfect estimates that correspond exactly to the real execution times that ODS uses. In this case, both algorithms will produce the same schedule. There is, however, a small overhead of additional messages passed in ODS that may influence the results.

2. It is easily verified that if relations *R* and *S* are skewed with degrees $s_R$ and $s_S$, the join result will possess a skew degree of $s_R + s_S$.

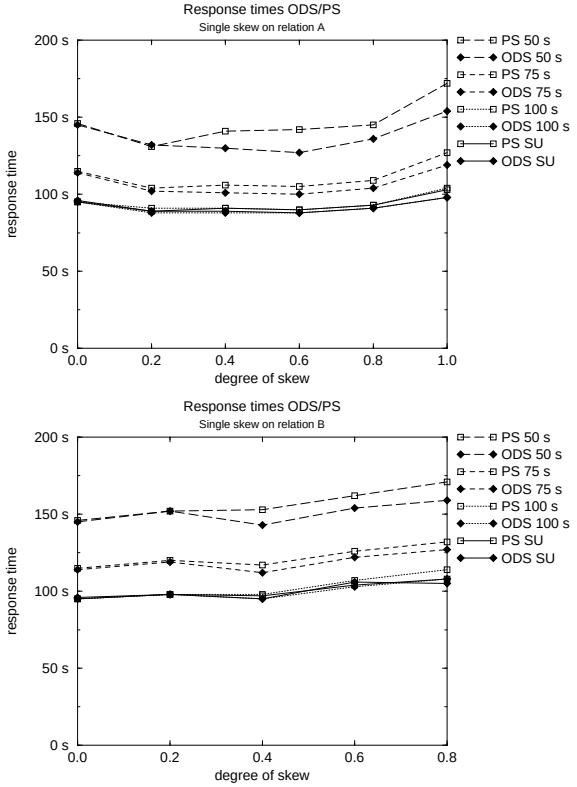



**Fig. 3 Simulation results for uniform queries, single skew**

## 5 SIMULATION RESULTS

### 5.1 UNIFORM QUERIES

In our first series of experiments, we used a single query that we submitted to the system repeatedly (100 times). It was a join on the relations *A* and *B* with the parameters discussed in the previous chapter. We monitored query response times under different degrees of skew and varying arrival rates. The results can be found in figures 3 and 4.

**Single skew on relation *A*.** Processing times in single-user mode (SU) are almost constant (≈ 90 s) because the size of the join result does not change with single skew. Apparently, our cost estimation function serves well in this case. Even in multi-user mode, results do not differ significantly between PS and ODS as long as the interarrival time corresponds to the single-user response time (≥ 100 s). For higher query rates (75 and 50 s interarrival time), differences of up to 12 % begin to emerge[3], increasing with the degree of skew.

3. Differences are reported relative to the response times for ODS.

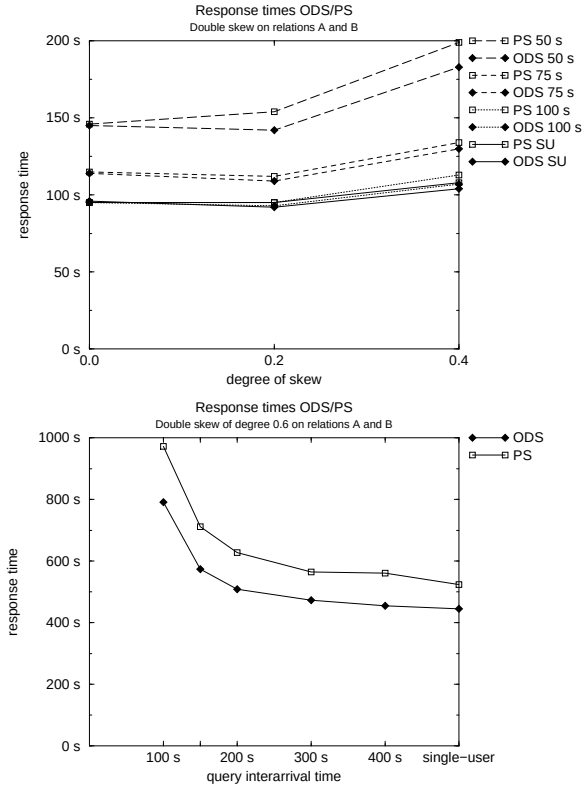



**Fig. 4 Simulation results for uniform queries, double skew**

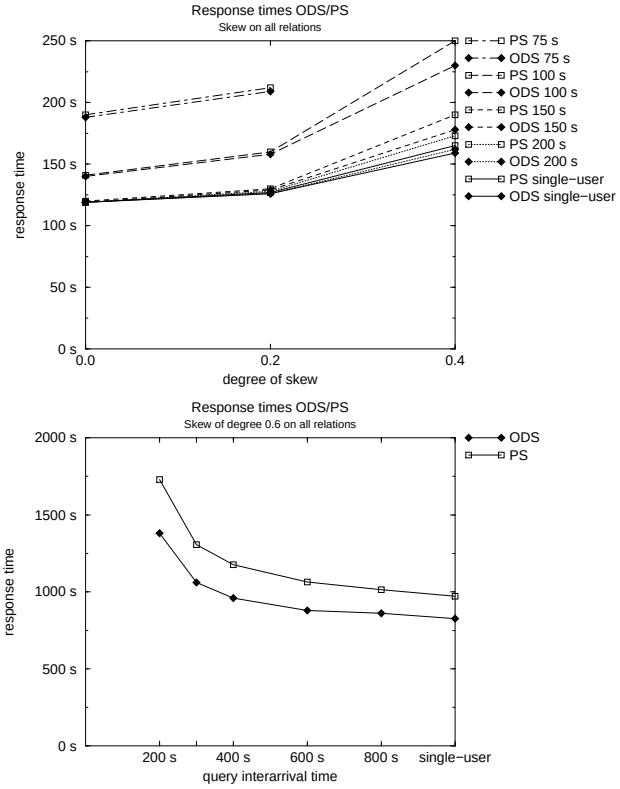



**Fig. 5 Simulation results for mixed queries**

**Single skew on relation *B*.** For single skew on *B*, result sizes also remain independent of the degree of skew. Nevertheless, response times begin to differ slightly even in single-user mode. This is because our optimizer calculates the number of buckets based on a worst-case estimate of memory demand which arrives at a higher value for skew in the larger relation (*B*) than it does for the smaller one (*A*). This high number of buckets requires more sub-queries and somewhat increases processing times. Similar to the previous series, we find a performance advantage for ODS of up to 8 %. Meaningful results could not be achieved for a skew of 1.0 because this case requires more than 33 000 buckets for processing. This is because we deliberately excluded skew buckets from our study as discussed in section 3.1; obviously, this is a case where a fragment/replicate scheme should be applied in a real-world situation.

**Double skew on *A* and *B*.** For low double skew (≤ 0.4), we obtained similar results as for the single skew cases. Although the join product (and join product skew) now increases with the degree of skew in the base relations, response times remain more or less constant, and the advantage of ODS does not exceed 9 %. For a skew of 0.6, however, result sizes and processing times grow dramatically. The performance difference is significant even in single-user mode (18 %); this has to be attributed to inaccuracies of the cost function for cases of higher double skew. In multi-user mode, the difference between PS and ODS is well beyond 20 % and remains so for a rather wide range of inter-arrival times (100 to 400 s). For higher skew, JPS becomes so extreme that the sub-join on a single bucket dominates response time. Again, this is the case we explicitly excluded, and we will not present any results for it here.

## 5.2 MIXED QUERIES

To investigate the behaviour of our two algorithms under slightly more complex circumstances, we included relation *C* in the query parameters and made our simulation system produce queries with random inputs. We excluded the possibility of joining a relation with itself, so that three different joins were generated: $A \bowtie B$, $A \bowtie C$, and $B \bowtie C$[4]. Although the sequence of 100 queries was random, it was the same in all the experiments due to CSIM's deterministic random number streams. Again, we surveyed query response times with varying arrival rates and skew; however, we applied the same degree of skew to all relations, so that double skew was present for all queries. The results are shown in figure 5.

As one would expect, the results are similar to the double skew case of the previous series. While the performance differences are insignificant for no or low skew (up to 0.2), they become more apparent for a skew degree of 0.4, where they reach 9 %. For a skew of 0.6, we find the largest differences between PS and ODS in all our experiments, ranging from 18 % in single-user mode to 25 % for an interarrival time of 200 s.

4. As our query optimizer exploits the commutativity of the join operator, such queries as $B \bowtie A$ do not count as separate cases.

### 5.3 INTERPRETATION

On the whole, our simulation study shows that on-demand scheduling outperforms predictive scheduling by as much as 25 % for high (double) skew, especially when the system is significantly loaded. For low skew and/or single-user mode, the response times of PS and ODS are almost equal. In fact, closer scrutiny reveals that both algorithms sometimes arrive at exactly the same schedule. This proves that the cost estimates of the Zhou/Orlowska function are quite adequate for these cases; it also shows that the message overhead for ODS is negligible.

Comparing single- and multi-user results, we find that about two thirds of the performance gains are due to growing inaccuracies in the cost function for higher skew. Although one might deduce from this that better cost functions could change conditions in favour of predictive scheduling, we come to the opposite conclusion: Since we used a very good cost function for Zipf-like value distributions, we expect the performance of PS to deteriorate even further in real-world situations with data whose properties cannot be approximated that well – unless one employs some costly statistics as discussed (and rejected) in chapter 2.

The other third of the performance advantage for ODS can be attributed to its insensitivity to disparate execution times. PS cannot achieve this even with perfect cost estimates, and we expect this factor to become even more important for more complex workloads. With smaller queries (or even write transactions that entail coherency problems) arriving in rapid succession, conditions in the system will change more quickly and unpredictably, which may cause execution times to fluctuate even worse. In a way, our simulation environment with its well controlled data and execution model represents a best-case scenario for PS.

Finally, one should keep in mind that the response times we regarded in our study include the scan phase as well as the join proper. Since the scans on the three base relations take up to 60 % of the total response time (single-user mode, no skew), performance gains calculated with respect to the join phase only will appear even more convincing.

## 6 CONCLUSION

In this paper, we compared traditional, predictive join scheduling techniques to our new on-demand approach. We found that for very large data volumes, ODS significantly outperforms PS under conditions of data skew and high system load because it is insensitive to inaccurate cost estimates and aberrations in execution speed. The success of ODS also demonstrates the superior load balancing potential of shared-disk architectures, since the impressive performance gains were possible only because expensive redistribution of data could be avoided.

Although the experiments we reported cover only a small range of possible applications for ODS, the results obtained so far are very encouraging. Still, further studies are required especially for more complex, mixed workloads that include multi-way joins, small queries, and write transactions as well as an increased amount of access conflicts. We are also interested in larger, more sophisticated hardware configurations such as NUMA or hybrid architectures (Valduriez, 1993), for which we imagine a hierarchical scheduling concept similar to the one by Bouganim et al. (1996). In addition, an SD adaptation of one of the skew resolution methods from chapter 2 and its comparison to ODS might be quite challenging. Finally, the ODS approach is not limited to join queries. We expect it to be equally applicable to other operations such as scans, sorts, aggregates etc.

## REFERENCES


Bouganim, L., Florescu, D., Valduriez, P., 1996, “Dynamic Load Balancing in Hierarchical Parallel Database Systems”, Proc. 22nd VLDB, Mumbai.

Dewan, H. M., Hernández, M., Mok, K. W., Stolfo, S. J., 1994, “Predictive Dynamic Load Balancing of Parallel Hash-Joins over Heterogeneous Processors in the Presence of Data Skew”, Proc. 3rd PDIS, Austin.

DeWitt, D. J., Naughton, J. F., Schneider, D. A., Seshadri, S., 1992, “Practical Skew Handling in Parallel Joins”, Proc. 18th VLDB, Vancouver.

Harada, L., Kitsuregawa, M., 1995, “Dynamic Join Product Skew Handling for Hash-Joins in Shared-Nothing Database Systems”, Proc. DASFAA ’95, Singapore.

Hua, K. A., Lee, C., 1991, “Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning”, Proc. 17th VLDB, Barcelona.

Lu, H., Tan, K.-L., 1992, “Dynamic and Load-balanced Task-Oriented Database Query Processing in Parallel Systems”, Proc. 3rd EDBT, Vienna.

Poosala, V., Ioannidis, Y. E., 1996: “Estimation of Query-Result Distribution and Its Application in Parallel-Join Load Balancing”, Proc. 22nd VLDB, Mumbai.

Rahm, E., 1993, “Parallel Query Processing in Shared Disk Database Systems”, Proc. HPTS-5, Asilomar.

Rahm, E., 1996, “Dynamic Load Balancing in Parallel Database Systems”, Proc. EURO-PAR 96, Lyon.

Schneider, D. A., DeWitt, D. J., 1989, “A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment”, Proc. ACM SIGMOD, Portland.

Valduriez, P., 1993, “Parallel Database Systems: the case for shared-something”, Proc. 9th ICDE, Vienna.

Wolf, J. L., Dias, D. M., Yu, P. S., Turek, J., 1994, “New Algorithms for Parallelizing Relational Database Joins in the Presence of Data Skew”, IEEE Trans. Knowledge and Data Engineering 6(6).

Wolf, J. L., Yu, P. S., Turek, J., Dias, D. M., 1993, “A Parallel Hash Join Algorithm for Managing Data Skew”, IEEE Trans. Parallel and Distributed Systems 4(12).

Walton, C. B., Dale, A. G., Jenevein, R. M., 1991, “A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins”, Proc. 17th VLDB, Barcelona.

Zhou, X., Orlowska, M. E., 1993, “A Dynamic Approach for Handling Data Skew Problems in Parallel Hash Join Computation”, Proc. IEEE TENCON ’93, Beijing.

Zhou, X., Orlowska, M. E., 1995, “Handling Data Skew in Parallel Hash Join Computation Using Two-Phase Scheduling”, Proc. ICA$^3$PP-95, Brisbane.