Dynamic Multi-Resource Load Balancing in Parallel Database Systems

Erhard Rahm

University of Leipzig
Institute of Computer Science
Augustusplatz 10-11
04109 Leipzig (Germany)
E-mail: rahm@informatik.uni-leipzig.de
Fax: +49-341-719-2399

Robert Marek

University of Kaiserslautern
Dept. of Computer Science
67653 Kaiserslautern (Germany)
E-mail: marek@informatik.uni-kl.de

Abstract

Parallel database systems have to support the effective parallelization of complex queries in multi-user mode, i.e. in combination with inter-query/inter-transaction parallelism. For this purpose, dynamic scheduling and load balancing strategies are necessary that consider the current system state for determining the degree of intra-query parallelism and for selecting the processors for executing subqueries. We study these issues for parallel hash join processing and show that the two subproblems should be addressed in an integrated way. Even more important, however, is the use of a multi-resource load balancing approach that considers all potential bottleneck resources, in particular memory, disks and CPU. We discuss basic performance tradeoffs to consider and evaluate the performance of several load balancing strategies by means of a detailed simulation model. Simulation results will be analyzed for multi-user configurations with both homogeneous and heterogeneous (query/OLTP) workloads.

Keywords:

Parallel Database Systems; Shared Nothing; Query Processing; Parallel Join Processing; Dynamic Load Balancing; Performance Analysis; Resource Allocation

1 Introduction

A significant trend in the commercial database field is the increasing support for parallel database processing [DG92, Va93]. This trend is both technology-driven and application-driven. Technology supports large amounts of inexpensive processing capacity by providing "super servers" [Gr95] consisting of tens to hundreds of fast standard microprocessors interconnected by a scalable high-speed interconnection network. The aggregate memory is in the order of tens to hundreds of gigabytes, while databases of multiple terabytes are kept online within a parallel disk subsystem. New application areas requiring parallel database systems for processing massive amounts of data and complex queries include data mining, digital libraries, new multimedia services like video on demand, geographic information systems, etc.. Even traditional DBMS applications increasingly face the need of parallel query processing due to growing database sizes and query complexity [Pi90, En94]. In addition, high transaction rates must be supported for standard OLTP applications.

The effective use of super-servers for database processing poses many implementation challenges that are unsolved in current products [Se93, Gr95]. One key problem is the effective use of intra-query parallelism in multi-user mode, i.e., when complex queries are executed concurrently with OLTP transactions and other complex queries. Multi-user mode (inter-transaction/inter-query parallelism) is mandatory to achieve acceptable throughput and cost-effectiveness, in particular for super-servers where a high number of processors must effectively be utilized. While proposed algorithms for parallel query processing also work in multi-user mode, their performance may be substantially lower than in single-user mode. This is because multi-user mode inevitably leads to data and resource contention that can significantly limit the attainable response time improvements due to intra-query parallelism. Resource contention is particularly critical because of the high resource demands (CPU cycles, memory space, disk bandwidth, communication bandwidth) of complex queries*. Furthermore, intra-query parallelism causes increased communication overhead compared to a sequential query execution on one node. Hence, the effective CPU utilization and thus throughput (e.g., for OLTP) are reduced.

In order to limit and control resource contention in multi-user mode, dynamic strategies for resource allocation (scheduling) and load balancing become necessary. Within a processing node, local scheduling components have to be extended to control local resource contention, e.g., by adding support for transaction priorities [CJL89, En91]. To limit resource contention in a distributed system, the workload must be allocated among the processing nodes such that the capacity of different processing nodes be evenly utilized (load balancing). At the same time, workload allocation should support a compromise with respect to communication and I/O overhead such that both intra-query parallelism and a sufficiently high throughput can be achieved. This requires a dynamic query processing approach where the degree of intra-query parallelism as well as the determination of which processing nodes

* Data contention problems between read-only queries and update transactions may be solved by a multiversion concurrency control scheme [CM86, BC92].

should process a given query are made dependent on the current system state at query run time.

Despite the high practical relevance of such dynamic scheduling and load balancing strategies to effectively support inter- and intra-query parallelism, very little research has been performed in this area (see Section 2). In a previous paper, we have begun to address these problems with respect to CPU resource contention [RM93]. The study focussed on parallel join processing in parallel Shared Nothing [DG92] database systems. Join processing was based on a dynamic redistribution of both input relations among multiple join processors. With such an approach there is high potential for dynamic load balancing since both, the degree of join parallelism as well as the selection of join processors, constitute dynamically adjustable parameters. In this paper, we investigate the much more complex problem of dynamic load balancing for multiple bottleneck resources. While considering only a single bottleneck resource is appropriate as a first step, such an approach is clearly ineffective if performance problems are caused by other resources. Dealing with multiple bottleneck resources is complicated by the fact that there are typically many scheduling and load balancing alternatives per resource type. Hence, the total solution space increases with the number of resource types to consider. Furthermore, in a parallel database systems resource utilization often varies largely at different nodes. As a result, the current bottleneck may constantly change and multiple bottlenecks may exist at the same time complicating dynamic scheduling and load balancing.

The present study deals with memory and CPU as bottleneck resources and focuses on parallel hash join processing [ME92, Gr93] in Shared Nothing systems. Disks constitute another critical bottleneck resource, in particular because CPUs are becoming faster at a high pace while disk access times improve only slowly [PGK88]. Unfortunately, the potential to dynamically influence disk contention is limited. This is because disk access frequencies to permanent data are primarily determined by the chosen database allocation*. However, the database allocation on disk is largely static and cannot be changed for individual queries or based on temporary overload situations. On the positive side, our load balancing schemes are able to limit disk contention for temporary files by optimizing usage of the available memory.

The remainder of this paper is organized as follows. The next section contains a brief survey of related studies on dynamic scheduling and load balancing in centralized and parallel database systems. In Section 3, we discuss some basic performance tradeoffs to motivate the choice of our dynamic multi-resource load balancing schemes. The various load balancing approaches that have been implemented within a detailed simulation model of a Shared Nothing database system are described in Section 4. Section 5 contains an overview of our simulation model and hash join implementation. In Section 6 we present and analyze simulation experiments for various database and workload configurations. In particular, we are

* In Shared Nothing systems the database allocation further reduces the potential for workload allocation since it prescribes at which processors scan operations have to be processed. Fortunately, dynamic load balancing is feasible for operations (e.g., joins) on intermediate results that can dynamically be redistributed.

studying multi-user experiments with homogeneous workloads (concurrent join queries) and heterogeneous (query/OLTP) workloads. Finally, we summarize the major findings of this investigation.

2 Related Work

Dynamic scheduling and workload allocation strategies for database processing have found considerable interest recently, but most studies concentrated on centralized DBMS. Furthermore, most studies only dealt with a single bottleneck resource. For instance, several researchers looked at the problem of controlling lock contention by dynamically adjusting the multiprogramming level [CKL90, Th92, MW92, WHMZ94]. Other studies coped with dynamic memory allocation strategies for multi-class workloads consisting of complex queries and OLTP transactions [JCL90, ZG90, PCL93, BCL93, BMCL94, DG94]. In particular, [ZG90, PCL93, DG94] proposed adaptive memory allocation schemes for hash joins that change a query's memory assignment during execution according to the memory requirements of higher-priority transactions. [MD93] addressed the scheduling problem when multiple hash join queries are to be processed at the same time. Different alternatives to allocate memory to join queries were considered, but the memory allocation was left unchanged during query execution. Scheduling strategies have also been investigated for real-time DBMS in order to satisfy transaction deadlines [AG89, PLC92].

The problem of dynamic load balancing in parallel database systems has mainly been considered for parallel Shared Everything (multiprocessor) DBMS so far [HSIT91, Om91, Ho92, LT92]. In these systems, dynamic load balancing is easily achieved since the operating system can automatically assign the next ready process/subquery to the next free CPU. Also, the memory load balancing problem does not exist for Shared Everything because there is no private main memory per processor. Furthermore, the shared memory supports very efficient interprocess communication so that the overhead for starting/terminating subqueries is much lower than for Shared Nothing. On the other hand, the number of processors is typically small for Shared Everything (≤ 30) thus restricting the degree of inter-/intra-query parallelism and the potential for dynamic load balancing.

For Shared Nothing dynamic forms of load balancing have been proposed for join processing in order to deal with data skew [WDJ91, WDYT91, DNSS92, HS93]. These approaches dynamically determine the size of intermediate results in order to redistribute the data among join processors such that they have to perform about the same amount of join work (in order to minimize execution skew). However, all these studies assumed single-user mode corresponding to a best-case situation with little or no resource contention. Hence, only intra-query load balancing is supported and the effectiveness of the proposals in multi-user mode must be questioned. Other performance studies of parallel join processing for Shared Nothing (without data skew) also assumed single-user mode, e.g., [SD89, SD90, MS91]. Except for our own work [RM93], we are not aware of other studies on dynamic load balancing for Shared Nothing in multi-user mode. In [RS95], we investigate the potential of Shared Disk database systems for dynamic load balancing for parallel scan queries.

3 Basic Performance Tradeoffs

We study the load balancing problem for parallel hash join processing and the most general case where both input relations are distributed among several join processors [ME92, Gr93]. In a first phase (building phase), a parallel scan is performed on the smaller (inner) relation at the *data processors* owning fragments of this relation. The scan output is dynamically distributed among several *join processors* according to a partitioning function (range or hash) on the join attribute. The join processors maintain a memory-resident hash table for the inner relation and support an overflow mechanism (leading to temporary I/O on local disks) if not all tuples of the inner relation fit into memory (see Section 5). In the second phase (probing phase), the outer relation is read in parallel at its data processors an distributed among the join processors. By using the same partitioning function for both join inputs, it is guaranteed that all matching tuples arrive at the same join processor. At the join processors, arriving tuples from the outer relation are probed against the hash table to find matching tuples from the inner relation.

The performance of such a join method is influenced by many factors like the chosen database allocation (number of data processors, fragmentation, etc.), size of the base relations, selectivity of scan operations, number of join processors, memory sizes, CPU speed, communication bandwidth, disk characteristics etc. Given a fixed database allocation and hardware configuration however, the optimal join strategy that minimizes the response time for a given join query is mainly determined by the number of join processors p and selection of these p join processors from the set of eligible processors*. In single-user mode, i.e. when there is only one join query in the system, the optimal number of join processors can be determined fairly easily by means of an analytical model. As outlined in [WFA92, Ma95], this can be achieved by developing an analytic formula for calculating the average join response time for a given number of join processors. The typical response time curve is shown in Fig. 1a indicating that response time can only be improved until a certain degree of parallelism. This is because the actual work per processor decreases, while the communication overhead for starting the subqueries, redistributing the scan output, merging the results and for termination (commit) increases with a higher number of join processors. The optimal degree of join parallelism in single-user mode, $p_{su\text{-}opt}$, is obtained by setting the derivative of the response time formula to zero. Selection of the join processors is also unproblematic since all processors are lightly loaded in single-user mode.

The study [RM93] showed however, that this changes significantly in multi-user mode. It was found that under high CPU utilization the optimal number of join processors is lower than in single-user mode (Fig. 1b) and that it is generally the lower the higher the system is utilized. This is because the communication overhead associated with a high degree of intra-

* We assume that any processor may act as join processor.

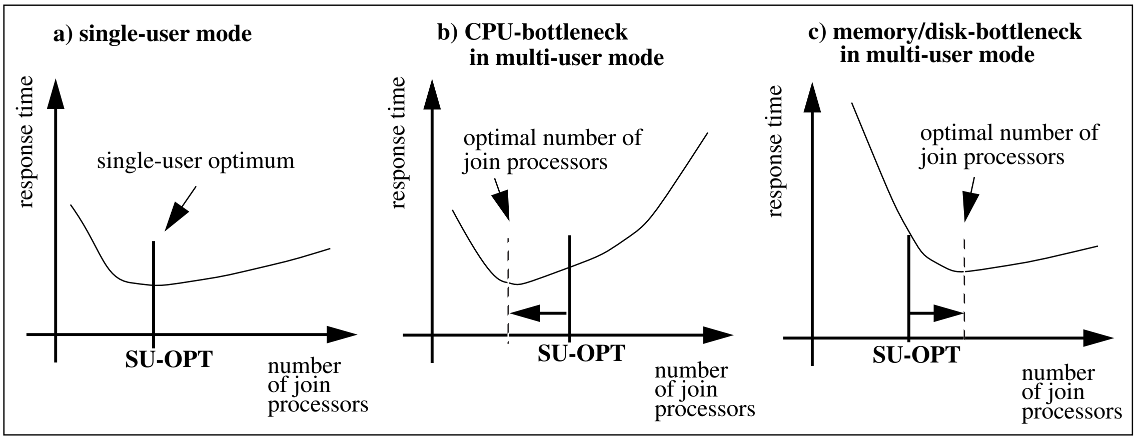

Fig. 1: Parallel join processing in single-user and multi-user mode:
basic response time development and optimal number of join processors

query parallelism is less affordable when processors are highly utilized. Furthermore, the least utilized CPUs should be selected for join processing.

In [RM93] we used sort-merge as the local join method and did not consider memory utilization for load balancing. However, for hash joins optimizing memory usage is likely to be more significant than CPU load balancing and must therefore be considered for dynamic load balancing in multi-user mode. As our simulation results will show it is of primary importance for hash joins to avoid overflow I/O as much as possible, i.e. to keep as much as possible of the inner relation memory-resident. Hence, the optimal degree of join parallelism in single-user mode is at least as high as required to avoid temporary file I/O. If the aggregate memory of all N processors is too small for keeping the inner relation memory-resident, then N constitutes the single-user optimum. Multi-user mode leads to memory contention so that only a subset of a node's memory may be available for join processing. Hence, the optimal number of join processors is expected to be the higher the less memory is available. As a result, under high memory (disk) utilization the optimal degree of join parallelism is typically higher than in single-user mode (Fig. 1c).

The discussion illustrates some basic tradeoffs to consider for memory and CPU load balancing (Fig. 2). On one hand, the degree of join parallelism must be high enough to limit memory and disk contention. On the other hand, it should be low enough to limit CPU contention. Hence, the degree of join parallelism must be chosen dynamically based on the current memory, disk and CPU utilization. As with all multi-resource scheduling strategies there is a danger of instability because removal of bottleneck 1 may create bottleneck 2 and vice versa (Fig. 2).

4 Load Balancing Strategies

The previous discussion showed that effective support for multi-user mode requires dynamic strategies for determining the degree of join parallelism as well as for selecting the join processors that consider both CPU and memory/disk bottlenecks. For CPU bottlenecks, the approaches proposed in [RM93] can be used that reduce the degree of join parallelism ac-

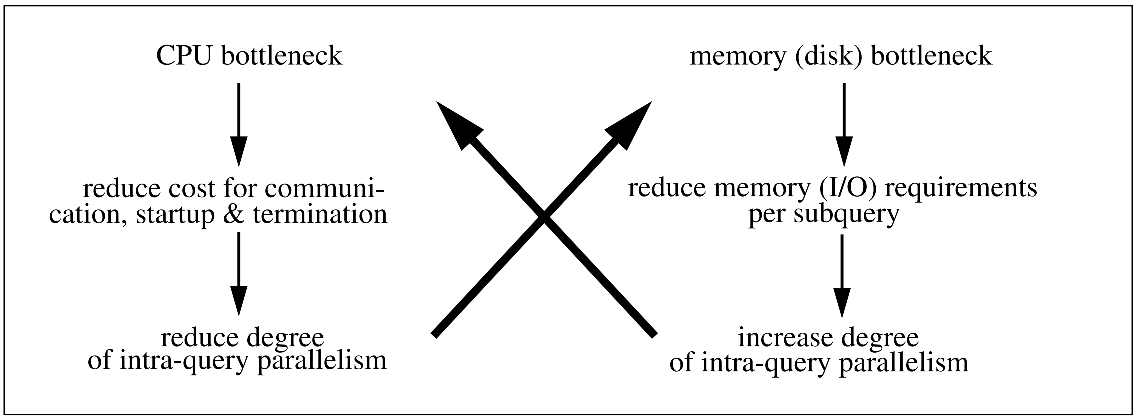

Fig. 2: Tradeoffs in dynamic load balancing with multiple bottlenecks

cording to the average CPU utilization and select the least utilized processors for join processing. Determining the optimal number of join processors under memory bottlenecks is more involved since it requires consideration of the available memory at the individual processors. For instance, is it better to allocate a join to 5 processors with at least 30 MB unused memory per processor or to 10 processors with a minimum of 10 MB available memory? In the former case, the aggregate memory size is higher thus reducing the number of I/Os to temporary files. The latter case, on the other hand, allows a higher degree of I/O and processing parallelism that may outweigh the increased number of I/Os. Dynamic load balancing is most complex for situations with both CPU *and* memory bottlenecks and if almost all processors are affected (global overload). For partial overload situations when only some processors suffer from bottlenecks, load balancing strategies that select the less utilized processors for join processing are likely to be very effective.

In the following we describe the load balancing strategies that have been implemented in our simulation system (Section 5) and that will be used in the performance evaluation (Section 6). We will consider static and dynamic strategies as well as isolated and integrated policies. *Isolated strategies* operate in two consecutive steps. In a first step the number of join processes (degree of join parallelism) is determined. In a second step these join processes are allocated to processing nodes based on some criterion. *Integrated strategies*, on the other hand, determine both the number of join processes and their allocation in a single step. The dynamic policies base their decisions on the current CPU utilization and memory availability. For this purpose we assume that a designated control node is periodically informed by the processors about their current utilization. During the execution of a query, information on the current CPU and memory utilization is requested from the control node to support dynamic load balancing.

We first describe the substrategies used in the isolated control approaches for determining the degree of join parallelism and for selecting the join processors. Afterwards, the integrated policies are presented.

Determining the number of join processors

We consider two static schemes that determine the number of join processors at query compile time and one dynamic approach.

Static degree of join parallelism

In the first policy, we simply choose the optimal number of join processors in single-user mode $p_{su\text{-}opt}$ as the degree of join parallelism. However, since according to [RM93] such a high number of join processors may cause performance problems if the system is CPU-bottlenecked we additionally study an alternative with a smaller number of join processors. In this approach, we use the number of join processors $p_{su\text{-}noIO}$ that avoid temporary I/O in single-user mode (if at all feasible with the given memory sizes). This number of join processors can be determined as follows:

$$(4.1) \qquad p_{su\text{-}noIO} = \mathrm{MIN}\ (N, \lceil (b_i * F) / m) \rceil\ )$$

In this formula, N represents the total number of processors, b_i the number of pages of the inner relation, F the overhead for the hash table ("fudge factor") and m the memory size (in pages) per processor. Temporary file I/O is avoided if the aggregate memory size of the $p_{su\text{-}noIO}$ processors exceeds the size of the smaller join input relation and if this relation is equally distributed among the join processors (no or only little redistribution skew).

*Dynamic determination of the degree of join parallelism (**DJP**)*

This dynamic strategy was already presented in [RM93]. It determines the degree of parallelism for multi-user mode p_{mu} by reducing the single-user optimum $p_{su\text{-}opt}$ according to the current CPU utilization:

$$(4.2) \qquad p_{mu} = p_{su\text{-}opt}\ (1 - u_{cpu}^3).$$

Here, u_{cpu} denotes the current average CPU utilization of all processors obtained from the control node. With this formula, a reduction takes place primarily for higher utilization levels ($u_{cpu} > 0.5$) when a high communication overhead for parallelization is not acceptable.

Selection of join processors

We support three strategies (RANDOM, LUC, LUM) that may be combined with any of the three approaches above for determining the degree of join parallelism.

RANDOM

This strategy selects the join processors at random. RANDOM is expected to spread the workload equally across all available nodes. Since RANDOM does not consider information about the current system state, it represents a static approach.

*Join processing on processors with Least Utilized CPU (**LUC**)*

In this approach, we select the processors with the lowest CPU utilization as join processors. For this purpose, the adaptive variation suggested in [RM93] is used that artificially increases the CPU utilization of a processor selected for join processing at the control node. This avoids that subsequent join queries are assigned to the same processors due to the delayed updating of information on CPU utilization.

*Join processing on processors with Least Utilized Memory (**LUM**)*
Join processes are assigned to the nodes with the most available main memory. Again, the control node's information is directly adapted for newly selected join processors.

Integrated strategies

Simulation results will be provided for three integrated and dynamic load balancing strategies. We have investigated several additional approaches; however, since they turned out to be less effective and due to space constraints we omit them from further consideration. The integrated schemes primarily use the control node's information on the current memory availability to determine the number of join processors and to select them according to the LUM strategy. For this purpose, we assume that the control node maintains the following data structure: *AVAIL_MEMORY [1..N] of (node-ID, free).*

This array indicates for each of the N processing nodes the available memory (*free*) and is sorted on the amount of free memory, i.e. AVAIL_MEMORY [1] refers to the processor with the least memory utilization etc.

All strategies try to avoid temporary file I/O by selecting p_{mu} join processors with a minimum of b pages so that p_{mu}*b exceeds the size of the smaller join input relation. Note that from the p_{mu} selected processors the one with the minimum amount of available memory is critical since it is likely to cause the highest I/O delays from all subqueries. Hence, it is the one that determines response times under memory or disk bottlenecks. As a result, it is desirable to find a processor selection so that temporary file I/O can be avoided even at the processor with the least available memory. The three strategies differ when there are several selections avoiding temporary I/O and in how CPU utilization is additionally considered.

*Minimal number of join processors that avoids temporary file I/O (**MIN_IO**)*
This strategy tries to find the minimal number of join processors k that avoids temporary file I/O. More formally, p_{mu} is determined such that

(4.3) $$p_{mu} = \text{MIN}\ (k \mid \text{AVAIL\_MEMORY}\ [k].\text{free} * k > b_i * F) \qquad k = 1, 2, ..., N$$

If the available memory does not allow avoidance of all temporary file I/Os, the number of join processors is selected so that the amount of overflow I/O is minimized. Join processing takes place on the processors specified in the first p_{mu} positions of AVAIL_MEMORY (LUM policy). This policy does not consider the current CPU utilization.

*Avoid temporary file I/O with number of join processors closest to $p_{su\text{-}opt}$ (**MIN_IO_SUOPT**)*
This strategy is different from MIN_IO only if there are multiple selections that avoid temporary file I/O. MIN_IO is expected to provide good performance compared to higher degrees of parallelism when CPU is bottlenecked since overhead for parallel processing is kept low. However, in the case of low CPU utilization it may unnecessarily restrict the degree of parallelism. To avoid this potential problem we select the number of processors closest to $p_{su\text{-}opt}$ for which temporary file I/O is avoided.

*Optimize I/O for maximum of join processors according to DJP (**OPT_IO_DJP**)*
This strategy is an extension of the previous one that restricts the maximal degree of join parallelism to limit CPU contention. The maximal number of join processors is determined according to the DJP policy (formula 5.2) based on the current CPU utilization. Such an approach is likely to be effective under higher CPU utilization. It also supports a low number of temporary file I/Os under light CPU load where the number of processors is only restricted by $p_{su\text{-}opt}$.

5 Simulation model

For the present study, we have extended our Shared Nothing simulation system already used in [RM93] by adding implementations for parallel hash join processing and for the various load balancing schemes. The gross structure of this simulation system is depicted in Fig. 3. In the following, we briefly describe the used database and workload models as well as the processing model to keep the paper self-contained. The simulation system is highly parameterized. In Section 6.1, we will provide an overview of the major parameters and their settings used in this study.

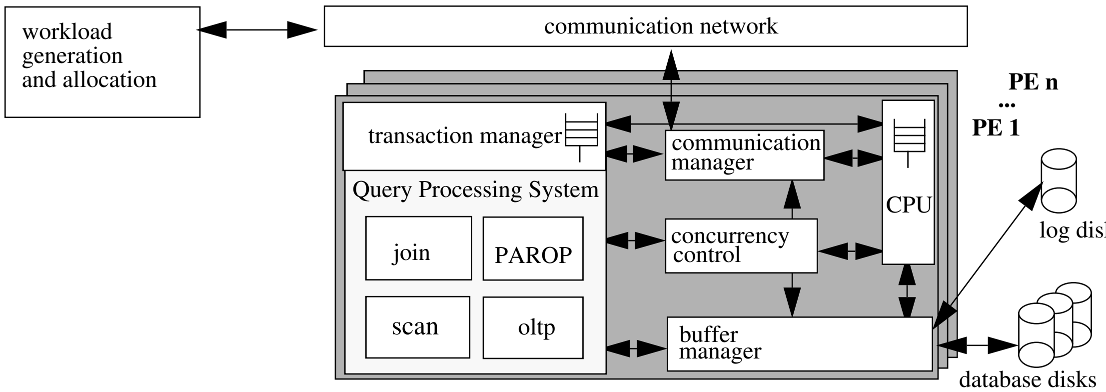

Fig. 3: Gross structure of the simulation system

Database and workload model

The database is modeled as a set of partitions. A partition may be used to represent a relation, a relation fragment or an index structure. It consists of a number of database pages which in turn consist of a specific number of objects (tuples, index entries). The number of objects per page is determined by a blocking factor which can be specified on a per-partition basis. Each relation can have associated clustered or unclustered B^+-tree indices. Relations and indices can be horizontally declustered across an arbitrary number of disks and processors.

We support heterogeneous (multi-class) workloads consisting of several query and transaction types. Queries correspond to transactions with a single database operation (e.g., SQL statement). Currently we support the following query types: relation scan, clustered index scan, non-clustered index scan, two-way join queries, multi-way join queries, and update statements (both with and without index support). We also support the debit-credit benchmark workload (TPC-B) and the use of real-life database traces [MR92]. The simulation system is an open queuing model and allows definition of an individual arrival rate for each transaction and query type.

Workload allocation takes place at two levels. First, each incoming transaction or query is assigned to one processor acting as the coordinator for the transaction/query. For this placement we support different strategies, in particular random allocation. The second form of workload allocation deals with the assignment of suboperations to processors during query processing and depends on the operators to be executed. For scan operators, the processor allocation is always based on a relation's data allocation. For join processing, we support several static and dynamic strategies for determining the degree of join parallelism and for allocating the join processes to processors as described in the previous section.

Workload processing

Each processor or processor element (PE) of the Shared Nothing system is represented by a transaction manager, a query processing system, CPU servers, a communication manager, a concurrency control component and a buffer manager (Fig. 3). The transaction manager controls the (distributed) execution of transactions. The maximal number of concurrent transactions (inter-transaction parallelism) per PE is controlled by a multiprogramming level. Newly arriving transactions must wait in an input queue until they can be served when this maximal degree of inter-transaction parallelism is already reached. The query processing system models basic relational operators (sort, scan, join) as well as a parallelization meta-operator (PAROP) that is used for dynamically redistributing data among processors and for merging multiple inputs. Different parallel execution strategies have been implemented for the various operators, in particular parallel hash joins (see below).

The number of CPUs per PE and their capacity (in MIPS) are provided as simulation parameters. The average number of instructions per request can be defined separately for every request type. To accurately model the cost of query processing, CPU service is requested for all major steps, in particular for transaction initialization (BOT), object accesses in main memory (value comparisons, operations on hash tables, etc.), I/O overhead, communication overhead, and commit processing. The communication network models transmission of message packets of fixed size. Messages exceeding the packet size (e.g., large sets of result tuples) are disassembled into the required number of packets.

For concurrency control, we employ distributed strict two-phase locking (long read and write locks). Global deadlocks are resolved by a central deadlock detection scheme. Distributed two-phase commit is supported and involves all processors that have participated during execution of the respective transaction/query. We support the read-only optimization where only one distributed commit phase is required for read-only sub-transactions (to release the read locks).

Database partitions can be kept memory-resident (to simulate main memory databases) or they can be allocated to a number of disks. Disks and disk controllers have explicitly been modelled as servers to capture potential I/O bottlenecks. Furthermore, disk controllers can have a LRU disk cache. The disk controllers also provide a prefetching mechanism to support sequential access patterns. If prefetching is selected, a disk cache miss causes multiple succeeding pages to be read from disk and allocated into the disk cache. Sequentially reading multiple pages is only slightly slower than reading a single page, but avoids the disk accesses for the prefetched pages when they are referenced later on. The number of pages to be read per prefetch I/O is specified by a simulation parameter.

The database buffer in main memory consists of a global buffer for all transactions/queries as well as private working spaces used for query processing (e.g., hash tables for hash joins). The global buffer is managed according to a LRU replacement strategy and a no-force update strategy with asynchronous disk writes. Private working spaces are dynamically assigned by reserving a certain number of pages for processing a given (sub)query.

Hash join processing

For parallel hash join processing, the input relations can be distributed among an arbitrary number of join processors*. Selection of the join processors depends on the respective approach for load balancing. For local join processing, we have implemented a *memory-adaptive* hash join algorithm, called *Partially Preemptible Hash Join* (PPHJ), that was shown to outperform traditional join methods like GRACE and hybrid hash join for mixed query/OLTP workloads [PCL93]. This is because it adapts the memory assignment for a join query according to the memory requirements of higher-priority OLTP transactions. The PPHJ algorithm partitions both join inputs into p partitions with p $=\sqrt{F \times bi}$ where b_i is the number of pages for the inner relation A. To make sure that each A partition can be held in memory, a minimum of p pages must be available for join processing.

The algorithm tries to keep as many A partitions as possible in memory to allow a direct join processing with the outer relation. In the case that memory has to be taken away from the join due to higher-priority transactions, one or more memory-resident A partition are written to disk. If more memory becomes available for join processing, one or more disk-resident A partition are brought into memory to support a direct join processing. Arriving tuples from the outer relation B can only be processed directly if the corresponding A partition is in memory. Otherwise, the B tuple is inserted into a temporary B partition that is written to disk. For disk-resident partitions the actual join processing is deferred until all tuples from the outer relation have been received. For this, the respective A partition is read and stored in a hash table. Afterwards the associated B partition is read and probed against the hash table.

A join query is only started at a node if the minimal space requirements of p pages are available. Otherwise, the join query is forced to wait in a memory queue that is managed according to a FCFS (first come, first served) scheduling policy. Similarly, executing hash joins are suspended if memory frames are stolen by higher-priority transactions and fewer than the minimal number of pages remain for join processing. Since all hash join queries are assumed to have equal priority, the memory allocation of a running query is not changed due to newly arriving joins.

* If the input relations are already declustered on the join attributes, join processing may also take place at the data processors. This reduces the communication overhead but offers little potential for dynamic load balancing.

6 Performance Analysis

Our experiments concentrate on the performance of parallel join processing in multi-user mode. The focus of the study is to compare the effectiveness of the various static and dynamic load balancing alternatives introduced in Section 4 for determining the degree of join parallelism and for selection of the join processors. Two types of multi-user load profiles are considered: a homogeneous workload consisting of join queries only as well as a heterogeneous (mixed) workload with both short OLTP transactions and join queries.

In the next subsection, we provide an overview of the parameter settings used in these experiments. Multi-user experiments for the homogeneous and heterogeneous workload are analyzed in 6.2 and 6.3, respectively. Many additional experiments have been conducted but cannot be described due to space restrictions. However, these experiments confirm the main findings of the selected experiments.

6.1 Simulation Parameter Settings

Fig. 4 shows the major database, query and configuration parameters with their settings. Most parameters are self-explanatory, some will be discussed when presenting the simulation results. The join queries used in our experiments perform two scans (selections) on the input relations *A* and *B* in parallel and join the corresponding results. The *A* relation contains 250.000 tuples, the *B* relation 1 million tuples*. The selections on *A* and *B* reduce the size of the input relations according to the selection predicate's selectivity (percentage of input tuples matching the predicate). Both selections employ clustered indices. The join result has the same size as the scan output on *A*. The number of processing nodes is varied between 10 and 80.

Both relations are uniformly declustered across disjoint sets of PE. To support a static load balancing for scan operations, each PE is assigned the same number of tuples. As a result the larger relation *B* is declustered across 80% of the PE, while the remaining 20% of the PE hold tuples of relation *A*.

The duration of an I/O operation is composed of the controller service time, disk access time and transmission time. For all sequential I/Os, in particular relation scans, clustered index scans and scans on temporary files (partitions), prefetching is utilized by the disk controllers to improve I/O performance. The disk access time for prefetching consists of a base access time per I/O (15 ms) plus an additional delay per page (1 ms). For a prefetching of 4 pages, the average disk access time is 19 ms. The parameter settings for the communication network have been chosen according to the EDS prototype [Sk92].

To capture the behavior of OLTP-style transactions, we provide a workload similar to the debit-credit benchmark. Each OLTP transaction performs four non-clustered index selects on arbitrary input relations and updates the corresponding tuples.

* As pointed out in [En94], most decision support queries are joins between a larger and a smaller relation.

Configuration	settings	Database/Queries	settings
number of PE (#PE)	10, 20, 40, 60, 80	**relation A:**	(100 MB)
CPU speed per PE	20 MIPS	#tuples	250.000
		tuple size	400 B
avg. no. of instructions:		blocking factor	20
initiate a query/transaction	25000	index type	clustered B^+-tree
terminate a query/transaction	25000	storage allocation	disk
I/O	3000	alloaction to PE	partial declustering (20% of #PE)
send message	5000		
receive message	10000	**relation B:**	(400 MB)
copy 8 KB message	5000	#tuples	1.000.000
read a tuple from memory page	500	tuple size	400 B
hash a tuple	500	blocking factor	20
insert a tuple into hash table	100	index type	clustered B^+-tree
write a tuple into output buffer	100	storage allocation	disk
probe hash table	200	allocation to PE	partial declustering (80% of #PE)
buffer manager:		**intermediate results:**	
page size	8 KB	storage allocation	buffer / disk
buffer size	50 pages (0.4 MB) (varied)		
disk devices:		**join queries:**	
number of disk servers per PE	10 (varied)	access method	via clustered index
controller service time	1 ms (per page)	scan selectivity	varied
transmission time per page	0.4 ms	no. of result tuples	100 % of the inner relation
avg. disk access time	15 ms	fudge factor hash table:	1.05
prefetching delay per page	1 ms	arrival rate	single-user, multi-user (varied)
disk cache	200 pages	query placement	random (uniformly over all PE)
prefetching size	4 pages	join parallelism	static/ dynamic
		selection of join processors	random/ dynamic

Fig. 4: System configuration, database and query profile

6.2 Multi-user experiments with homogeneous workloads

The homogeneous workload consists of a single (join) query type. Inter-query parallelism is used to execute multiple queries at a time. Since we want to support not only short response times but also good throughput, we increase the query arrival rate proportionally with the number of PE. We first present multi-user results for isolated load balancing strategies using a static degree of intra-query parallelism. Afterwards we analyze the effectiveness of isolated and integrated strategies that dynamically determine the number of join processors. Next, an experiment with a pronounced disk and memory bottleneck is described. Finally, we study the influence of the join complexity on the effectiveness of dynamic load balancing.

Isolated load balancing strategies with static degree of join parallelism

Fig. 5 shows the multi-user response times for static degrees of parallelism and three different allocation strategies. For comparison purposes, the single-user results obtained with $p_{su\text{-}opt}$ join processors are also shown. For the assumed join query, 3 join processors are sufficient in single-user mode to avoid temporary file I/O, i.e. $p_{su\text{-}noIO} = 3$. The single-user optimum is substantially higher ($p_{su\text{-}opt} = 30$). The system size is varied between 10 and 80 PE; the arrival rate is 0.25 queries per second (QPS) per PE.

For this workload, for up to 40 PE the system is only lightly loaded. Hence, using $p_{su\text{-}opt}$ join processors provides the best multi-user performance with response times not much higher than in single-user mode. In this range, restricting join processing to $p_{su\text{-}noIO}$ processors achieves suboptimal performance since CPU parallelism is not fully exploited. Furthermore, choosing only $p_{su\text{-}noIO}$ join processors is not sufficient to avoid temporary file I/O in multi-user mode because the available memory per processor is smaller than in single-user mode.

With a growing number of processors, performance is increasingly dominated by CPU bottlenecks due to higher arrival rates and increased overhead for the dynamic redistribution of both join inputs*. The redistribution overhead is particularly high for the strategies employing $p_{su\text{-}opt}$ (30) join processors causing substantial response time deteriorations due to CPU contention (more than 80% CPU utilization on an 80 PE system). On the other hand, using $p_{su\text{-}noIO}$ join processors results in a significantly lower CPU utilization (approx. 50% for 80 PE). However, this is achieved at the expense of increased I/O delays and higher disk utilization since 3 join processors are not sufficient any more to avoid temporary file I/O. Still, the best static strategy using $p_{su\text{-}noIO}$ processors (in combination with LUM) outperforms the strategies using $p_{su\text{-}opt}$ processors for more than 60 PE.

The load balancing strategy for selecting the join processors also has a profound impact on the response time results, in particular for higher utilization levels (number of PE). RANDOM exhibits the worst performance in all cases despite the fact that a homogeneous workload is relatively favorable for such a strategy. Still, the CPU and memory utilization of the individual processors varied substantially, in particular with only 3 ($p_{su\text{-}noIO}$) join processors per query. Since this strategy suffered from memory and I/O bottlenecks for a higher number of PE, the LUM policy was much more efficient than the LUC alternative for selecting the join processors. In case of $p_{su\text{-}opt}$ join processors memory contention was not a problem. Instead, CPU was the bottleneck for a higher number of PE. Therefore, the LUC policy was (slightly) more efficient than LUM for the case of 30 ($p_{su\text{-}opt}$) join processors. However, there is no significant difference between the LUM and the LUC policy, since CPU utilization and memory utilization were closely correlated for the homogeneous workload and 30 join processors per query.

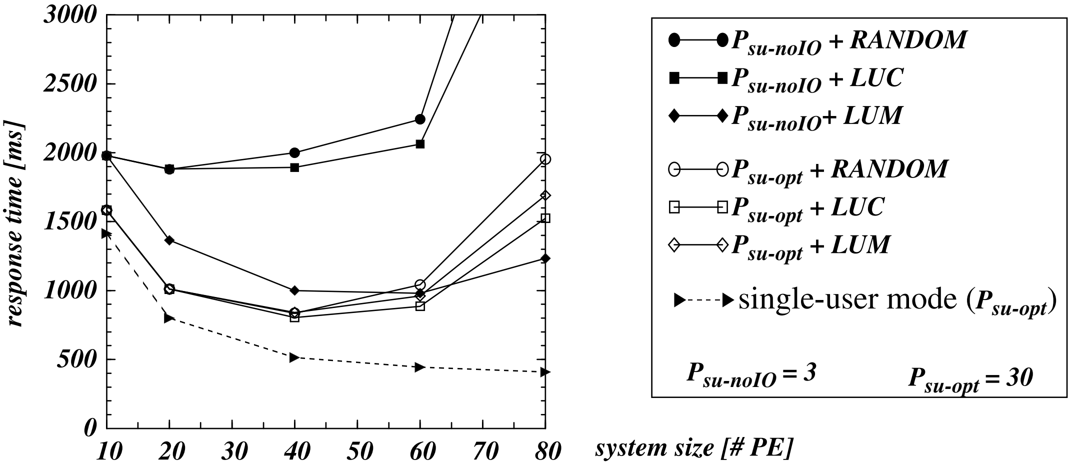

Fig. 5: Multi-user results with constant degree of parallelism and random/dynamic workload allocation (multi-user join 0.25 QPS/PE; 1% scan selectivity)

---

* The redistribution overhead per query increases with the number of nodes since the two relations are declustered across 80% and 20% of all processors, respectively.

Isolated/integrated load balancing strategies with dynamic degree of parallelism

As the discussed results have shown, statically determining the degree of join parallelism is not appropriate for multi-user mode due to changing levels of resource utilization. Therefore, we focus now on the results obtained for a dynamic calculation of the number of join processors (Fig. 6). We consider two isolated approaches based on the DJP strategy for dynamically determining the degree of join parallelism according to the current CPU utilization and using a RANDOM- or LUM-based selection of join processors. In addition, results for the three integrated approaches from Section 5 are shown.

Interestingly, Fig. 6 shows that the worst performance is achieved for the two integrated load balancing strategies MIN_IO and MIN_IO_SUOPT, in particular for a higher number of processors. Both strategies do not provide optimal performance, since they do not consider the current CPU utilization but merely try to avoid temporary file I/O. However, for this purpose an increasing number of join processors became necessary for larger system sizes leading to an even higher CPU contention (> 85% CPU utilization) than with a static degree of $p_{su\text{-}opt}$ join processors. For instance, more than 40 join processors were necessary for a system of 80 PE to avoid temporary I/O. MIN_IO is superior to MIN_IO_SUOPT for larger configurations since the latter strategy generally chooses a higher number of join processors. For smaller configurations (lower CPU utilization), on the other hand, selecting the minimal number of join processors avoiding temporary file I/O (MIN_IO) is slightly less efficient since CPU parallelism is not fully utilized.

Most efficient were the strategies DJP and OPT_IO_DJP that reduce the degree of join parallelism under high CPU load. Both strategies apply at most $p_{su\text{-}opt}$ join processors and reduce the degree of join parallelism with increasing CPU utilization. Therefore, even for 80 PE CPU utilization could be kept below 65% still permitting acceptable response time. While the use of a RANDOM selection of join processors is again worse than a LUM-based selection for the DJP policy, such an approach was still better than the two integrated schemes MIN_IO and MIN_IO_SUOPT. This shows that under high CPU load reducing the degree of join parallelism is more important than minimizing the amount of temporary I/O.

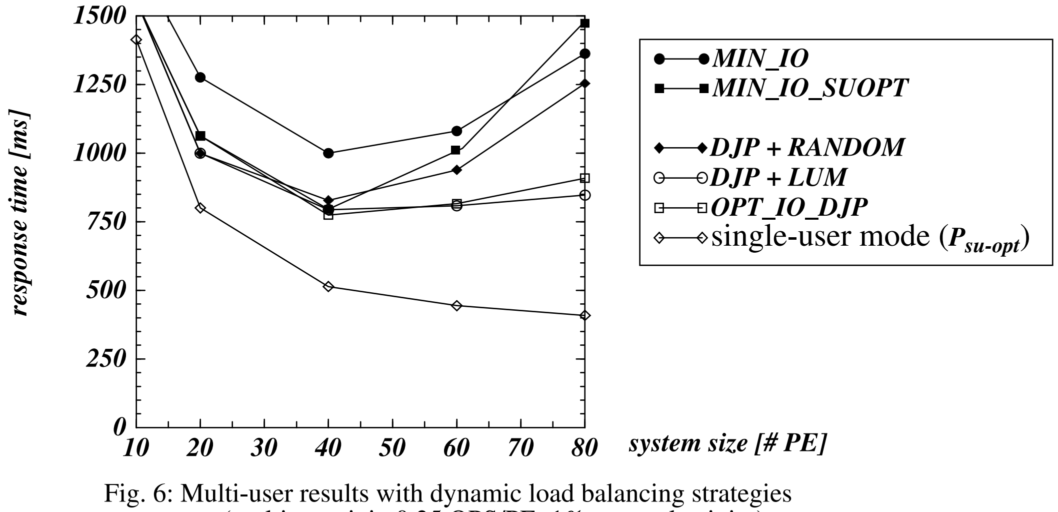

Fig. 6: Multi-user results with dynamic load balancing strategies
(multi-user join 0.25 QPS/PE; 1% scan selectivity)

The two best strategies DJP+LUM and OPT_IO_DJP showed very similar performance characteristics for this experiment. For the experiments with heterogeneous workloads, the differences between these approaches will become more apparent.

Memory/disk bottleneck

In the previous experiment that was largely influenced by CPU contention for larger system sizes, the strategies reducing the degree of parallelism according to the current CPU utilization were most effective. We now focus on a memory-bound environment by reducing the memory size per processor by a factor of 10 and reducing the query arrival rate. Furthermore, we assume only 1 disk per PE for temporary file I/O (instead of 10 disks). For this experiment, we only compare one of the worst strategies of the previous experiment (MIN_IO_SUOPT) with one of the best strategies (DJP + LUM) for both single-user and multi-user mode (Fig. 7).

The assumed workload resulted in a low CPU utilization of under 20%, but caused a high buffer utilization (> 90%). Since there was no CPU bottleneck, the DJP strategy always used $P_{su\text{-}opt}$ join processors. However, this degree of join parallelism was not sufficient in multi-user mode to minimize the number of overflow I/Os causing an increasing degree of memory and disk utilization (> 60%) for growing system sizes. The same effect would have occurred for the OPT_IO_DJP strategy. The MIN_IO_SUOPT approach, on the other hand, was able to minimize the amount of overflow I/O by increasing the number of join processors with the system size. As indicated in Fig. 7, the average degree of join parallelism in multi-user mode was increased to up to 42 for 80 PE as opposed to 33 in single-user mode and 30 for DJP. The corresponding savings in the number of I/Os and the reduced disk contention allowed drastically improved response times compared to the DJP policy.

These experiments illustrate that there is no single policy that performs best under all conditions, but that the load balancing strategy itself should be selected according to the current load and resource situation.

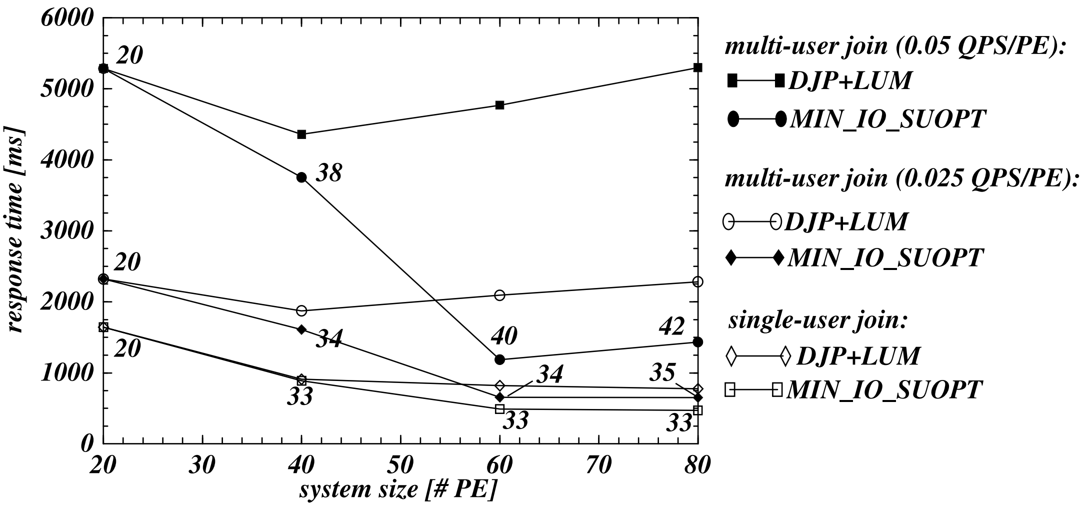

Fig. 7: Dynamic load balancing strategies in a memory-bound environment (1% scan selectivity)

Influence of the join complexity on the effectiveness of dynamic load balancing

To study the influence of the join complexity on the effectiveness of dynamic load balancing we vary the size of the join input by using different scan selectivities. This experiment was performed for a constant system size of 60 PE. Scan selectivity was varied between 0.1 and 5% for both input relations. For each join complexity, the arrival rate was determined individually, so that at least one of the physical resources (CPU, memory or disk) was highly loaded (>75%). Fig. 8 shows the relative response time improvement using dynamic strategies compared to a static degree of join parallelism (MIN(N,$p_{su\text{-}opt}$)) and random selection of join processors.

We observe that the dynamic load balancing schemes outperform the static approach in all cases, but that the relative performance improvements shrink with increasing join complexity. This is largely because we use a constant system size while increasing the join size leading to an increase in the optimal number of join processors. In single-user mode, the optimum $p_{su\text{-}opt}$ increases from 10 for a scan selectivity of 0.1% to 70 (> N) for a selectivity of 5%; the minimal number of nodes needed to avoid overhead I/O, $p_{su\text{-}noIO}$, grows from 1 to 14. In multi-user mode, larger joins also require higher degrees of parallelism not only to reduce the amount of temporary I/O but also to reduce the amount of processing per join processor.

For small joins (scan selectivity 0.1%) avoiding temporary I/O is no problem so that performance is primarily limited by the CPU contention associated with higher degrees of join parallelism (unfavorable ratio between startup/termination cost and actual work). Hence, the best performance is achieved for the strategies using few join processors ($P_{su\text{-}noIO}$+LUM and MIN_IO), while the schemes using $p_{su\text{-}opt}$ join processors (MIN_IO_SUOPT) achieve the lowest response time improvements. For larger joins (5%), on the other hand, startup and termination costs become less relevant and higher degrees of join parallelism are needed to limit temporary I/O and to fully exploit CPU parallelism. The strategy $P_{su\text{-}noIO}$+LUM achieves the worst performance since it utilizes only 14 processors which is not sufficient to

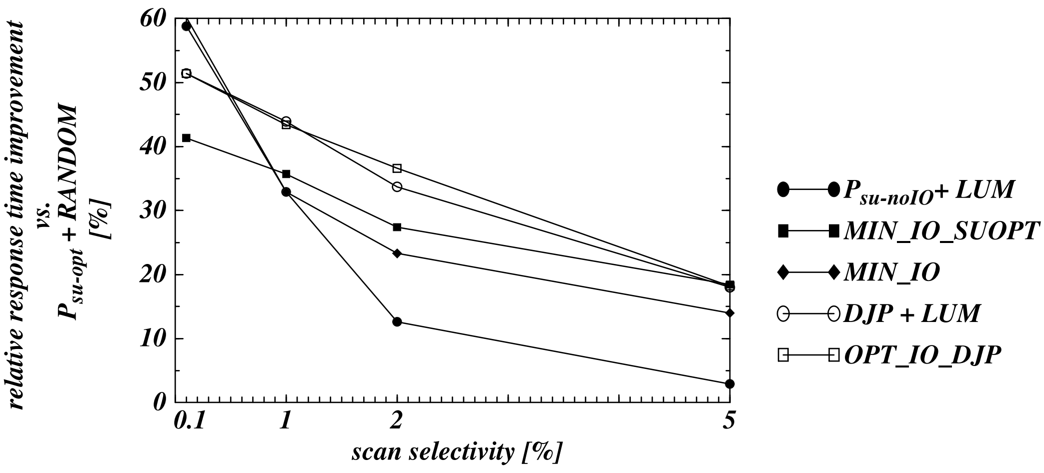

Fig. 8: Influence of the join complexity on the relative response time improvement using dynamic load balancing compared to $P_{su\text{-}opt}$+*RANDOM*

avoid temporary I/O in multi-user mode. MIN_IO avoids memory/disk bottlenecks, but also selects too few join processors so that no sufficient level of CPU parallelism is achieved. For large joins, the best performance is provided by the strategies DJP+LUM, OPT_IO_DJP and MIN_IO_SUOPT as they employ almost all processors for join processing. Still they are able to improve response times (by about 18%) compared to the static scheme $P_{su\text{-}opt}$+RANDOM (which uses all processors) because the dynamic strategies avoid join processing at temporarily overloaded nodes.

The experiment confirms the expectation that the potential for dynamic load balancing becomes small as soon as the optimal number of join processors approaches the total number of processors. In addition, the use of a homogeneous workload can be considered as a worst-case assumption for complex queries as it results in a relatively uniform resource allocation even for random selection of the join processors. (Furthermore, the chosen database allocation allowed an equal distribution of the scan work.) In real systems, the workload is expected to consist of transaction and query types with largely different resource requirements thus improving the load balancing potential. Such heterogenous workloads will be considered in the next experiment. Furthermore, the potential for dynamic load balancing increases with the total number of processors, i.e., such schemes are essential for super-servers.

6.3 Multi-user experiments with heterogeneous workloads

We now study the effectiveness of dynamic load balancing for the case of heterogeneous workloads consisting of OLTP transactions and join queries. For OLTP processing, we assume a simple transaction type accessing only one relation (A or B) and that an affinity-based routing [Ra92] can achieve a largely local processing (similar to debit-credit). For the concurrent execution of join queries, we study multi-user join processing.

Fig. 9 shows the average join response times for two mixed workloads differing in whether the OLTP transaction type is accessing relation A (Fig. 9a) or relation B (Fig. 9b). In both cases we use an OLTP transaction rate of 100 TPS (transactions per second) per A (B) node. The OLTP workload causes per A (B) node a CPU, disk, and memory utilization of about 50%, 60%, and 45%, respectively. Join queries arrive at a rate of 0.075 QPS per PE. We consider two static load balancing schemes for join processing with a fixed degree of join parallelism of $P_{su\text{-}opt}$ or $P_{su\text{-}noIO}$ processors that are randomly selected. For $P_{su\text{-}noIO}$ processors we additionally investigate the LUM allocation strategy. Moreover, the two dynamic load balancing strategies DJP + LUM and OPT_IO_DJP are examined.

The results indicate that for mixed workloads dynamic load balancing is indeed even more effective (and needed) than for homogeneous workloads. The differences between static and dynamic approaches are particularly pronounced in the case when the OLTP load is processed on B nodes (Fig. 9b). This is because we have the four-fold OLTP throughput compared to the other configuration resulting in a higher system utilization and longer response times. Static schemes based on RANDOM selection of join processors are particularly unsuited in such a situation as they frequently assign join work on nodes that are highly utilized due to OLTP processing. Using a small static degree of join parallelism ($p_{su\text{-}noIO}$) in combination with a LUM-based selection of join processors is already much better since it largely avoids join processing on nodes with high memory utilization. Still, such semi-static approaches are insufficient since they cause either an unnecessarily high I/O overhead ($P_{su\text{-}noIO}$) or CPU contention ($P_{su\text{-}opt}$).

The dynamic approaches could largely avoid these deficiencies and provided much better performance than the static schemes. In particular, response times could be kept very low for larger system sizes despite the growing query and transaction throughput. This is particularly the case for the integrated policy OPT_IO_DJP. The isolated strategy DJP + LUM, however, suffered from performance problems with a lower number of processors, in particular with OLTP processing on relation A (Fig. 9a). The problem comes from the fact that this strategy only considers CPU utilization for determining the number of join processors p_{mu}, while memory utilization is solely used for selecting the join processors. For smaller system sizes of up to 30 PE when the *average* CPU utilization is comparatively low, the DJP strategy does not reduce the degree of join parallelism but requires join processing on all PE. Hence, join processing also takes place on the processors that are highly utilized due to OLTP processing causing substantial performance degradations. OPT_IO_DJP, on the other hand, uses the current CPU utilization only to determine the maximal number of join processors but selects a smaller degree of parallelism if this allows for reduced I/O requirements according to the current memory utilization. In this way, this strategy was able to avoid join processing on the OLTP nodes permitting substantially better response times. This demon-

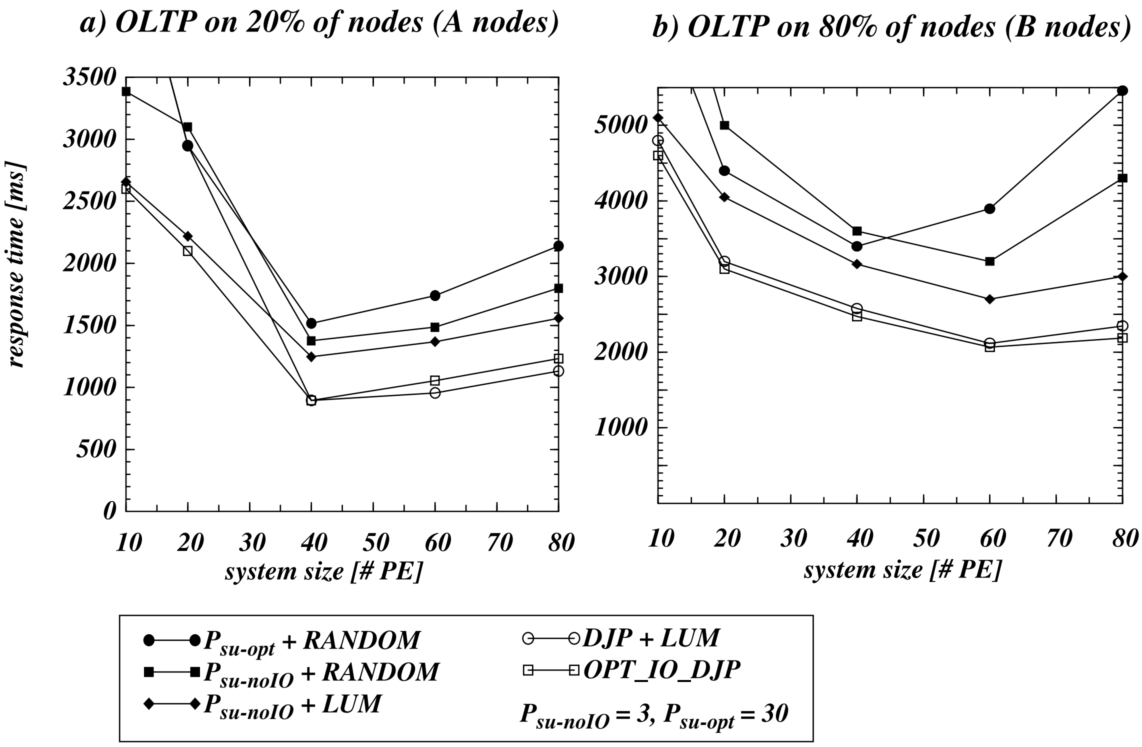

Fig. 9: Static vs. dynamic load balancing for mixed workloads
(multi-user join 0.075 QPS/PE; 5 disks per PE)

strates the importance of determining the number of join processors and selecting the processing nodes in an integrated way.

7 Conclusions

In this paper, we have investigated the problem of dynamic load balancing for parallel Shared Nothing database systems. Such a load balancing is a critical prerequisite for effective utilization of "super servers", in particular to support effective intra-query parallelism in multi-user mode, i.e., in combination with inter-query and inter-transaction parallelism. The major control decisions to draw dynamically include determining the degree of intra-query parallelism and selecting the processors for executing subqueries. We found that these two subproblems need to be solved in an integrated way and that the current system state with respect to multiple resources, in particular CPU, memory and disk, needs to be considered for dynamic load balancing.

We have studied these issues for parallel hash join processing based on a dynamic redistribution of both join inputs among several join processors. While in single-user mode minimizing the amount of I/O to temporary files (due to hash table overflow) is of prime importance, the performance in multi-user mode may be dominated by other factors like the degree of CPU and disk contention. In particular, we observed a basic performance tradeoff with respect to the optimal degree of join parallelism in multi-user mode. Under high CPU utilization we found it necessary to reduce the degree of join parallelism in order to limit CPU contention (communication overhead for startup/termination and data redistribution). Under disk and memory bottlenecks, on the other hand, the degree of join parallelism should be increased in order to reduce the memory and I/O requirements per subquery.

We have investigated the performance of several single- and multi-resource load balancing strategies for homogeneous and heterogeneous (query/OLTP) workloads by means of a detailed simulation model. We considered static and dynamic as well as isolated and integrated policies. Isolated policies determine the degree of join parallelism independently from the policy used for selecting the join processors, while integrated strategy try to address both scheduling problems together. We found that dynamic load balancing schemes clearly outperform static approaches in particular for heterogeneous workloads when the load situation at different processors may vary significantly. However, simple integrated policies considering only the current utilization of a single resource (e.g., memory) are not always better than isolated schemes considering multiple resources. This underlines the need to have a dynamic, integrated and multi-resource load balancing approach. As our results suggest, such an approach should be realized by a family of load balancing strategies so that the most appropriate policy can be selected according to the current system state. For instance, if the system suffers primarily from memory and disk bottlenecks an integrated policy like MIN_IO_SUOPT should be chosen that minimizes the amount of I/O based on the current memory availability. For situations with high CPU contention or with both CPU and memory bottlenecks, an integrated policy like OPT_IO_DJP has proven to be very effective.

While our study focussed on parallel hash join processing, we believe the principles behind our strategies are equally valid for other relational operators that use a dynamic redistribution of their input for parallel execution (e.g., sort). Furthermore, we believe that the proposed strategies are not limited to Shared Nothing but can equally be applied in Shared Disk database systems. Currently, we are studying the performance of different approaches to deal with data skew (in particular, redistribution skew) in multi-user mode. Preliminary results indicate that the overhead of proposed skew handling techniques is a significant problem in multi-user mode. On the other hand, the skew problem may be reduced by dynamic load balancing strategies that do not try to generate equally-sized subjoins but select the join processors dependent on the size of the subjoins (by assigning larger subjoins to less loaded nodes, etc.).

8 References

AG89 Abbott, R., Garcia-Molina, H.: Scheduling Real-Time Transactions with Disk-Resident Data. *Proc 15th VLDB Conf.*, 385-396, 1989

BC92 Bober, P.M., Carey, M.J.: On Mixing Queries and Transactions via Multiversion Locking. *Proc. 8th IEEE Data Engineering Conf.*, 535-545, 1992

BCL93 Brown, K., Carey, M.J., Livny, M.: Managing Memory to Meet Multiclass Workload Response Time Goals. *Proc 19th VLDB Conf.*, 328-341, 1993

BMCL94 Brown, K.P.; Mehta, M.; Carey, M.J.; Livny, M.: Towards Automated Performance Tuning for Complex Workloads. *Proc. 20th VLDB Conf.*, 72-84, 1994

CJL89 Carey, M.J., Jauhari, R., Livny, M.: Priority in DBMS Resource Scheduling. *Proc. 15th VLDB Conf.,* 397-410, 1989

CKL90 Carey, M.J., Krishnamurthi, S., Livny, M.: Load Control for Locking: The 'Half-and-Half' Approach. *Proc. 9th ACM Symp. on Principles of Database Systems*, 72-84, 1990

CM86 Carey, M.J., Muhanna, W.A.: The Performance of Multiversion Concurrency Control Algorithms. *ACM Trans. on Computer Systems* 4 (4), 338-378, 1986

DG92 DeWitt, D.J., Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems. *Comm. ACM* 35 (6), 85-98, 1992

DG94 Davison, D.L.; Graefe, G.: Memory-Contention Responsive Hash Joins. *Proc. 20th VLDB Conf.*, 379-390, 1994.

DNSS92 DeWitt, D.J., Naughton, J.F., Schneider, D.A., Seshadri, S.: Practical Skew Handling in Parallel Joins. *Proc. 18th VLDB Conf.* 1992

En91 Englert, S.: Load Balancing Batch and Interactive Queries in a Highly Parallel Environment. *Proc. IEEE Spring CompCon Conf.*, 110-112, 1991

En94 Englert, S.: NonStop SQL: Scalability and Availability for Decision Support. *Proc. ACM SIGMOD Conf.*, 491, 1994

Gr93 Graefe, G.: Query Evaluation Techniques for Large Databases. *ACM Comput. Surveys* 25 (2), 73-170, 1993

Gr95 Gray, J.: Super-Servers: Commodity Computer Clusters Pose a Software Challenge. *Proc. German Database Conf. BTW*, March 1995

Ho92 Hong, W.: Exploiting Inter-Operation Parallelism in XPRS. *Proc. ACM SIGMOD Conf.*, 19-28, 1992

HS93 Hua, K.A., Su, J.X.W.: Dynamic Load Balancing in Very Large Shared-Nothing Hybercube Database Computers. *IEEE Trans. on Computers* 42 (12), 1425-1439, 1993

HSIT91 Hirano, Y., Satoh, T., Inoue, U., Teranaka, K.: Load Balancing Algorithms for Parallel Database Processing on Shared Memory Multiprocessors. *Proc. 1st Int. Conf. on Parallel and Distributed Information Systems*, 210-217, 1991

JCL90 Jauhari, R., Carey, M.J., Livny, M.: Priority-Hints: An Algorithm for Priority-Based Buffer Management. *Proc. 16th VLDB Conf.*, 708-721, 1990

LT92 Lu, H., Tan, K.: Dynamic and Load-Balanced Task-Oriented Database Query Processing in Parallel Systems. *Proc. EDBT*, LNCS 580, 357-372, 1992

Ma95 Marek, R.: A Cost Model for Parallel Query Processing in Shared Nothing DBS (in German). *Proc. German Database Conf. BTW*, March 1995

MD93 Mehta, M., DeWitt, D.J.: Dynamic Memory Allocation for Multiple-Query Workloads. *Proc 19th VLDB Conf.*, 354-367, 1993

ME92 Mishra, P., Eich, M.: Join Processing in Relational Databases. *ACM Comput. Surveys* 24 (1), 63-113, 1992

MR92 Marek, R., Rahm, E.: Performance Evaluation of Parallel Transaction Processing in Shared Nothing Database Systems*, Proc. 4th Int. PARLE Conf.,* LNCS 605, 295-310, 1992

MS91 Murphy, M.; Shan, M.: Execution Plan Balancing. *Proc. 1st Int. Conf. on Parallel and Distributed Information Systems*, 1991

MW92 Mönkeberg, A., Weikum, G.: Performance Evaluation of an Adaptive and Robust Load Control Method for the Avoidance of Data-Contention Thrashing. *Proc. 18th VLDB Conf.*, 432-443, 1992

Om91 Omiecinski, E.: Performance Analysis of a Load-Balancing Hash-Join Algorithm for a Shared-Memory Multiprocessor. *Proc 17th VLDB Conf.*, 375-385, 1991

PCL93 Pang, H., Carey, M.J., Livny, M.: Partially Preemptible Hash Joins. *Proc. ACM SIGMOD Conf.*, 59-68, 1993

PGK88 Patterson, D.A., Gibson, G., Katz, R.H.: A Case for Redundant Arrays of Inexpensive Disks (RAID). *Proc. ACM SIGMOD Conf.***,** 109-116, 1988

PLC92 Pang, H., Livny, M., Carey, M.J.: Transaction Scheduling in Multiclass Real-Time Database Systems. *Proc. 13th Real-Time Systems Symp.*, 1992

Pi90 Pirahesh, H. et al.: Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches. *Proc. 2nd Int.Symposium on Databases in Parallel and Distributed Systems*, IEEE Computer Society Press, 1990

Ra92 Rahm, E.: A Framework for Workload Allocation in Distributed Transaction Processing Systems. *Journal of Systems and Software* 18, 171-190, 1992

RM93 Rahm, E., Marek, R.: Analysis of Dynamic Load Balancing Strategies for Parallel Shared Nothing Database Systems. *Proc 19th VLDB Conf.*, 182-193, 1993

RS95 Rahm, E., Stöhr, T.: Analysis of Parallel Scan Processing in Shared Disk Database Systems. Techn. Report, Univ. of Leipzig, 1995

SD89 Schneider, D.A., DeWitt, D.J.: A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. *Proc. ACM SIGMOD Conf.*, 110-121, 1989

SD90 Schneider, D.A., DeWitt, D.J.: Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. *Proc. 16th VLDB Conf.*, 469-480, 1990

Se93 Selinger, P.: Predictions and Challenges for Database Systems in the Year 2000. *Proc 19th VLDB Conf.*, 667-675, 1993

Sk92 Skelton, C.J. et al.: EDS: A Parallel Computer System for Advanced Information Processing, *Proc. 4th Int. PARLE Conf.*, Springer-Verlag, LNCS 605, 3-18, 1992

Th92 Thomasian, A.: Thrashing in Two-Phase Locking Revisited. *Proc. 8th IEEE Data Engineering Conf.*, 518-526, 1992

Va93 Valduriez, P.: Parallel Database Systems: Open Problems and New Issues. *Distr. and Parallel Databases* 1 (2), 137-165, 1993

WDJ91 Walton, C.B; Dale A.G.; Jenevein, R.M.: A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. *Proc. 17th VLDB Conf.*, 537-548, 1991

WDYT91 Wolf, J.L., Dias, D.M., Yu, P.S., Turek, J.: An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew. *Proc. 7th IEEE Data Engineering Conf.*, 200-209, 1991

WFA92 Wilschut, A.; Flokstra, J.; Apers, P.: Parallelism in a Main-Memory DBMS: The performance of PRISMA/DB. *Proc. 18th Int. Conf. on Very Large Data Bases*, 521-532, 1992

WHMZ94 Weikum, G.; Hasse, C.; Mönkeberg, A.; Zabback, P.: The COMFORT Automatic Tuning Project. *Information Systems* 19(5), 381-432, 1994

ZG90 Zeller, H., Gray, J.: An Adaptive Hash Join Algorithm for Multiuser Environments. *Proc. 16th VLDB Conf.*, 186-197, 1990