Parallel Query Processing in Shared Disk Database Systems

Erhard Rahm

University of Kaiserslautern, Germany
E-Mail: rahm@informatik.uni-kl.de

Abstract

System developments and research on parallel query processing have concentrated either on "Shared Everything" or "Shared Nothing" architectures so far. While there are several commercial DBMS based on the "Shared Disk" alternative, this architecture has received very little attention with respect to parallel query processing. This paper is intended as a first step to close this gap. A detailed comparison between Shared Disk and Shared Nothing reveals many potential benefits for Shared Disk with respect to parallel query processing. In particular, Shared Disk supports more flexible control over the communication overhead for intra-transaction parallelism, and a higher potential for dynamic load balancing and efficient processing of mixed OLTP/ query workloads. We also outline necessary extensions for transaction management (concurrency/coherency control, logging/recovery) to support intra-transaction parallelism in the Shared Disk environment.

1 Introduction

Parallel database systems are the key to high performance transaction and database processing [DG92, Va93a]. These systems utilize the capacity of multiple locally coupled processing nodes. Typically, fast and inexpensive microprocessors are used as processors to achieve high cost-effectiveness compared to mainframe-based configurations. Parallel database systems aim at providing both high throughput for on-line transaction processing (OLTP) as well as short response times for complex ad-hoc queries. Efficient query processing increasingly gains importance due to the wide-spread use of powerful query languages and user tools. Next-generation database applications for engineering, VLSI design or multi-media support will lead to substantially increased query complexity. Since these complex queries typically access large amounts of data or/and perform extensive computations, in general the response time goal can only be achieved by employing parallel query processing strategies [Pi90]. Furthermore, performance should scale with the number of nodes: adding processing nodes ideally improves throughput for OLTP or response times for complex queries linearly.

Parallel database systems are typically based on one of three general architectures termed "Shared Everything" (or "Shared Memory"), database sharing ("Shared Disk") and database partitioning ("Shared Nothing") [St86, Bh88, DG92]. *Shared Everything* (SE) refers to the use of multiprocessors for database processing. In this case, we have a tightly coupled system where all processors share a common main memory as well as peripneral devices (terminals, disks). The shared memory supports efficient cooperation and synchronization between processors. Furthermore, load balancing is comparatively easy to achieve by providing common job queues in shared memory. These advantages are especially valuable for parallel processing leading to increased communication and load balancing requirements to start/terminate and coordinate multiple subtransactions per transaction. Multiprocessors are already utilized by several commercial DBMS for parallelizing certain operations (e.g., index construction/maintenance or sorting) [Ha90, Da92]. Furthermore, several research prototypes use the SE approach for a more general form of intra-transaction parallelism, e.g., XPRS [SKPO88] and Volcano [Gr90].

However, SE has significant limitations with respect to meeting high performance and availability requirements. In particular, the shared memory can become a performance bottleneck thereby limiting the scalability of the architecture. Consequently, the number of processors is quite low in current SE systems (≤ 30). In addition, there are significant availability problems since the shared memory reduces failure isolation between processors, and since there is only a single copy of system software like the operating system or the DBMS [Ki84]. These problems can be overcome by Shared Nothing (SN) and Shared Disk (SD) systems which are typically based on a loose coupling of processors. In loosely coupled systems, each processor is autonomous, i.e., it runs a separate copy of the operating system, the DBMS and other software, and there is no shared memory [Ki84]. Inter-processor communication takes place by means of message passing. Loose coupling can be used for interconnecting uniprocessors or multiprocessors. We use the term processing node (or node) to refer to either a uniprocessor or a multiprocessor as part of a loosely coupled system.

SN and SD differ in the way the external storage devices (usually disks) are allocated to the processing nodes. In SN systems [St86, ÖV91, DG92], external storage devices and thus the data are partitioned among all nodes. A node can directly access only data of the local database partition; if remote data needs to be accessed a distributed transaction execution becomes necessary. For this purpose, the DBMS has to support construction of distributed query execution plans as well as a distributed commit protocol. In SD systems [Ra86, Yu87, MN91], on the other hand, each node can directly access all disks holding the common database. As a result, no distributed transaction execution is necessary as for SN. Inter-node communication is required for concurrency control in order to synchronize the nodes' accesses to the shared database. Furthermore, coherency control is needed since database pages are buffered in the main memory of every node. These page copies remain cached beyond the end of the accessing transaction making the pages susceptible to invalidation by other nodes.

For both system architectures, SN and SD, there are several commercially available DBMS or research prototypes. Currently however, intra-transaction parallelism is only supported by SN systems[1], e.g., in products like Tandem NonStop SQL [Ta89] and Teradata's

DBC/1012 [Ne86] and prototypes such as Arbre [LY89], Bubba [Bo90], EDS [Sk92], Gamma [De90], and Prisma [Ap92]. With the exception of Tandem, these systems represent back-end systems (database machines) dedicated to database processing[2]. According to [DG92], the commercial SN systems are very successful and have demonstrated both linear throughput scaleup and response time speedup in certain benchmarks [Ta88, EGKS90]. The SD approach is supported by several products including IBM's IMS, DEC's Rdb and CODASYL DBMSs, and Oracle's Parallel Server[3]. Oracle's SD system has reached the highest transaction rates in the TPC-A and TPC-B benchmarks [Gr91]. In 1991, more than 1000 tpsB were already achieved on a nCUBE-2 system with 64 nodes [Or91]. Recently, more than 1 KtpsA was achieved on a Sequent configuration of 2 nodes with 23 processors each. In addition to nCUBE and Sequent, other microprocessor-based "cluster" architectures also support the shared-disk paradigm (Pyramid, Encore, etc.).

Despite the significance of SD for high performance database processing, this approach has found almost no attention in the research literature with respect to intra-transaction parallelism. Fortunately, many of the techniques developed for parallel query processing in SN or SE systems can be utilized for SD as well. However, as we will see the fact that each node can directly access all data gives SD a considerably higher flexibility for parallel query processing compared to SN.

Since SN is currently the major approach for parallel query processing, we discuss the SD approach by comparing it with SN. For this purpose, we first compare some general features of both architectures with respect to database processing (section 2). This discussion reconsiders some of the arguments that have been made to promote SN as "the" approach for parallel query processing. This is particularly strange since the SD approach was excluded from consideration without even looking at its potential with respect to intra-transaction parallelism (e.g., in [DG92]). In section 3, we extend our comparison by focussing on parallel query processing for both architectures. The comparison is not intended to show that SD is "better" than SN, but to illustrate that there are major advantages for SD which make this approach an interesting target area for further research on parallel query processing. In particular, major problems of the SN approach with respect to intra-transaction parallelism (e.g., physical database design, support for mixed workloads) are likely to be easier solved for SD. In section 4, we discuss extensions for transaction management that are to be supported by SD systems for intra-transaction parallelism.

1. Oracle is currently working on intra-query parallelism for its SD system [Li93].
2. The back-end approach makes it easier to utilize microprocessors for database processing and may support a more efficient communication than general-purpose systems. Furthermore, the back-end systems may be connected to a variety of front-end systems like workstations or mainframe hosts. On the other hand, a high communication overhead is introduced if front-end applications have to send every database operation to the back-end for processing or if large result sets need to be returned by the back-end. For OLTP applications, transaction programs may be kept in the back-end system (as "stored procedure") to reduce the communication requirements.
3. In [DG92], the Oracle approach has wrongly been classified as a Shared Nothing system.

2 SN vs. SD revisited

Several papers have compared SD and SN (and SE) with each-other [Tr83, Sh86, St86, Bh88, Pi90, DG92, Va93a, Ra93b], partially coming to different assessments. One advantage of SN is that interconnecting a large number of nodes is less expensive than for SD since every disk needs only be connected to one node. However, the cost of interconnecting nodes and disks is also comparatively low in SD architectures such as Ncube. These architectures are based on microprocessors and do not directly attach a disk drive to all nodes, but achieve the same connectivity by an interconnection network (e.g., hybercube) where I/O requests and pages may be transferred through intermediate nodes. Such a message-based I/O interface between processing nodes and I/O nodes (disk controllers) is also used in the DEC VaxClusters [KLS86]. This approach does not imply any inherent limit on the number of processing nodes. For instance, the new nCUBE-3 system has an architectural maximum of 65 336 processors [Gi93].

Another advantage of SN is its broad applicability to different environments. In particular, the nodes of a SN system may be locally or geographically distributed, and the DBMS instances may be identical or not (heterogeneous and federated database systems are based on SN). On the other hand, SD assumes a homogenous architecture and typically requires close proximity of processing nodes and storage devices due to the attachment of the disk drives to all nodes. However, these are no significant disadvantages with respect to parallel database processing. High performance, high availability and scalability require a close cooperation between all nodes/DBMS which cannot be provided by heterogeneous/federated database systems. Similarly, locally distributed systems provide high communication bandwidth and low latencies and are therefore preferable to geographically distributed systems for high performance database processing. Furthermore, dynamic load balancing is easier achieved in a locally distributed system since global state information (e.g., on CPU utilization) can be maintained with smaller cost and higher accuracy.

A major problem of SN is finding a "good" fragmentation and allocation of the database. The database allocations tend to be static due to the high overhead for physically redistributing large amounts of data. It has a profound impact on performance since it largely determines where database operations have to be processed thus affecting both communication overhead and node utilization. Since different transaction and query types have different data distribution requirements, the database allocation must inevitably constitute a compromise for an expected workload profile. The chosen allocation may however easily lead to suboptimal performance and load imbalances due to short-term workload fluctuations or other deviations in the actual from the expected workload profile. Variations in the number of nodes (node failure, addition of new node) require a reallocation of the database. SD avoids these problems since there is no need to physically partition the database among nodes. In particular, this promises a higher potential for load balancing since each node can process any database operation. We will further discuss the role of the database allocation in the next section.

In [DG92], it is argued that coherency control may limit the scalability of SD compared to SN. In view of the scalability problems of tightly coupled systems caused by cache coherency protocols, this is conceivable. However, for workloads for which SN systems could demonstrate scalability so far, namely read-dominated workloads or perfectly "partitionable" loads like debit-credit, SD has no problems with coherency control. It is fair to say that SN systems depend even more than SD on the partitionability of the workload and database due to the large performance impact of the database allocation for SN [YD91]. SD can utilize the partitionability of a workload to limit the number of buffer invalidations by employing an "affinity-based" workload allocation which assigns transactions referencing/updating the same database portions to the same node [Ra92b]. This is a dynamic form of workload/data allocation which can more easily be adapted than the database allocation for SN. SD allows a more efficient coherency control than in other environments since transactions constitute comparatively large execution units reducing the frequency of invalidations and data transfers between nodes. Furthermore, coherency control can closely be integrated with concurrency control limiting the number of extra messages to a large extent; some coherency control protocols do not require any additional messages [Ra91c].

In [DG92] it is also argued that SN would scale better than SD because interference is minimized. In particular, SN would only move small "questions" and (filtered) answers through the network, while SD would require moving large quantities of data. This argument apparently assumes conventional (non-parallel) database processing because efficient intra-query parallelism for SN requires redistribution of large amounts of data between nodes (see section 3). Furthermore, SN is highly susceptible to processor interference since questions on a particular database object can only be processed by the owning node, even if it is already overutilized. The performance results published for SN so far were typically based on many best-case conditions regarding workload profile, database allocation and load balancing. In particular, almost all performance studies on the use of intra-transaction parallelism assumed single-user mode implying minimial processor interference. Under more realistic conditions (multi-user mode, mixed workloads), SN was shown to exhibit much poorer performance [MR92].

3 SN vs. SD for parallel query processing

The comparisons of the different architectures made so far in the literature did not consider intra-transaction parallelism in most cases. Even in papers coping with parallel database processing [Pi90, DG92], no special attention was paid to parallel query processing for SD. In this section, we will show that SD offers significant advantages for parallel query processing compared to SN. The comparison considers differences and commonalities with respect to database allocation and the processing of scan and join operations. Furthermore, we discuss processing of mixed OLTP/query workloads. At first however, we introduce some terminology on parallel transaction processing.

3.1 Parallel transaction processing terminology

As indicated in Fig. 1, there are different forms of parallel transaction processing to consider. First of all, one can distinguish between inter- and intra-transaction parallelism. *Inter-transaction parallelism* refers to the concurrent execution of multiple independent transactions on the same database. This kind of parallelism is already present in centralized DBMS (multi-user mode), e.g., in order to overlap I/O delays to achieve acceptable system throughput. Of course, inter-transaction parallelism must also be supported in parallel database systems (in combination with intra-transaction parallelism). Otherwise, the throughput requirements for OLTP could not be met and a very poor cost-effectiveness would be obtained (the processing capacity cannot effectively be utilized in single-user mode).

To improve the response time of complex transactions, intra-transaction parallelism is needed either in the form of inter-query or intra-query parallelism. *Inter-query (operation) parallelism* refers to the concurrent execution of different database operations (e.g., SQL statements) of the same transaction. The degree of parallelism obtainable by inter-query parallelism is limited, however, by the number of database operations per transaction as well as by precedence constraints between these operations. Currently, commercial DBMS do not support this kind of parallelism because the programmer would have to specify the query dependencies using adequate language features.

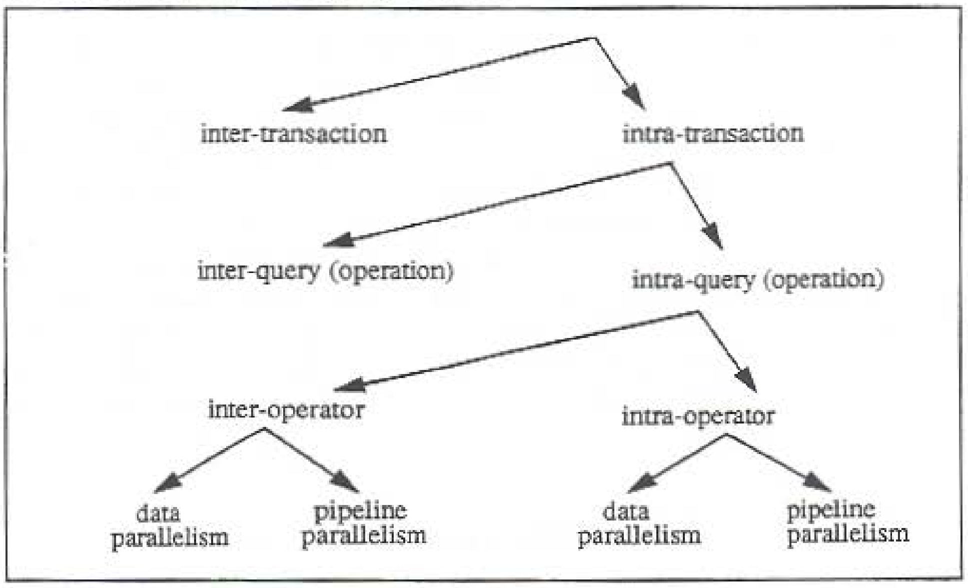


Fig. 1: Classification of parallel transaction processing forms

Intra-query parallelism aims at parallelizing individual database operations. This type of parallelism has been made possible by relational DBMS with their set-oriented operations. It is achieved transparently for database users and application programmers by the DBMS's query optimizer. For every query, the query optimizer generates a parallel execution plan specifying the base operators (scan, join, etc.) to be used for processing the query. Parallelism may be employed between different operators (inter-operator parallelism) or within operators (intra-operator parallelism). Similar to inter-query parallelism, the achievable degree of parallelism with inter-operator parallelism is limited by the number of operators and precedence constraints between operators in the execution plan. Most parallel database systems currently support intra-transaction parallelism only in the form of intra-operator parallelism.

Research papers have studied inter-operator parallelism primarily for optimizing multi-way join operations [SD90, LST91, CYW92].

As indicated in Fig. 1, inter- and intra-operator parallelism may use either data or pipeline parallelism. *Data parallelism* is based on a partitioning of the data so that different operators or suboperators can concurrently process disjoint data partitions. For instance, scan operators on different relations can be executed in parallel (inter-operator parallelism); a scan on a single relation can be parallelized by partitioning the relation and the scan operator (intra-operator parallelism). *Pipeline parallelism* uses an overlapped (pipelined) execution of different operators or suboperators to reduce processing time. In this case, the results of one (sub)operator are incrementally sent to the next (sub)operator in the execution plan (dataflow principle). Compared to data parallelism the benefits of pipeline parallelism are limited since relational pipelines are rarely long (≤ 10 operators) [DG92]. Furthermore, there may be large differences in the execution cost of different operators. In addition, some operators cannot be used for pipelining as they do not provide their output until they have processed all input data (e.g., sort or duplicate elimination). Also, the communication cost for pipelining can be substantial since many small messages may have to be exchanged between two operators instead of transferring the result in a single message. For these reasons, we mainly focus on data parallelism in this paper. There are no significant differences between SN and SD with respect to pipeline parallelism which primarily works on derived data that can dynamically be redistributed among nodes for both architectures.

Data parallelism requires both I/O parallelism and processing parallelism. *I/O parallelism* means that the data to be processed by a database operation is declustered across multiple disks so that I/O time is not limited by the low bandwidth of a single disk. *Processing parallelism* requires that the input data of a database operation can be processed by multiple CPUs to avoid that execution time is limited by the capacity of a single processor. For SN and SD, the database allocation to disk directly determines the maximal degree of I/O parallelism per relation. Since the data allocation on disk is expensive to change, the (maximal) degree of I/O parallelism is a rather static parameter. For SN, the degree of processing parallelism is also largely dependent on the static data allocation to disk since each disk is exclusively assigned to one processing node. As we will see below, this results in a reduced flexibility for dynamically varying the degree of processing parallelism compared to SD where each node can access any disk.

3.2 Database allocation

Shared Nothing

Declustering in SN systems is typically based on a horizontal fragmentation and allocation of relations. Fragmentation may be defined by a simple round robin scheme or, more commonly, by hash or range partitioning on a *partitioning attribute* [DG92]. Data allocation incorporates determination of the *degree of declustering* D and mapping of the fragments to D disks (processing nodes). Indices (B+ trees, in general) are also partitioned with the relation so that at each of the D nodes there is a (sub-)index for the local tuples.

Determination of an appropriate database allocation means finding a compromise with respect to contradicting subgoals: support for a high degree of intra-transaction parallelism, low communication overhead, and effective load balancing. For instance, a high degree of declustering supports intra-transaction parallelism and load balancing, but at the expense of a high communication overhead for starting and terminating suboperations. A small degree of declustering, on the other hand, reduces communication overhead and may be sufficient to meet the response time requirements on small relations or for selective queries (e.g., index scan). Furthermore, it supports effective inter-transaction parallelism (high OLTP throughput).

Similar trade-offs are posed by the various fragmentation alternatives [DG92]. Round robin is the simplest approach and promises an optimal load balancing since each fragment contains the same number of tuples. However, it requires a high communication overhead because queries cannot be limited to a subset of the partitions since database allocation is not based on attribut values. Range partitioning permits exact-match and range queries on the partitioning attribute to be localized to the nodes holding relevant data thereby reducing communication overhead. With hash partitioning only exact-match queries (but not range queries) on the partitioning attribute can be restricted to the minimal number of nodes. For range queries, hash partitioning may achieve better load balancing than range partitioning since it randomizes logically related data across multiple nodes rather than cluster it [DG92]. Heuristics for calculating a database allocation for a known workload profile are proposed in [CABK88, GD90].

Shared Disk

For SD, only a database allocation to disk needs to be determined as already for centralized DBMS. The declustering of relations across multiple disks can be defined similarly as for SN, i.e., either based on round robin, hash or range partitioning. The round robin approach may even be implemented outside the DBMS, e.g., by the operating system's file manager or, in the case of disk arrays [PGK88], by the disk subsystem[4]. In this case, the DBMS optimizer would still have to know the degree of declustering to allocate resources (CPUs, memory) for parallel query processing. Note that centralized DBMS and SN systems typically are unable to utilize the I/O bandwidth provided by disk arrays. This is because disk arrays can deliver I/O bandwidths in excess of 100 MB/s, while it is estimated that a single CPU can process relational data merely at a rate of merely 0.1 MB/s per MIPS [GHW90]. For SD the CPU bottleneck is avoided if multiple processing nodes share a disk array and if the disk array is able to split the output of a single read request among multiple nodes (memories). Hence, SD is better positioned than SN or SE to utilize disk arrays for parallel query processing. Of course, SD can also use conventional disk farms instead of disk arrays.

However, hash and range partitioning offer similar advantages over round robin than for SN. In particular, processing may be limited to a subset of the disks thereby reducing the work for a relation scan. Furthermore, disk contention is reduced thus cutting I/O delays for concurrent transactions (better I/O rates). Note however, that the degree of declustering does not

4. File blocks rather than records would then constitute the units of declustering.

directly influence the communication overhead as for SN, since the degree of processing parallelism can be chosen independently for SD (see below). As a result, the "optimal" degree of declustering for a given relation and workload profile may be different for SD than for SN. The use of disk arrays is more complicated in the case of a DBMS-controlled data allocation based on attribute values, in particular if the disk array should still provide high fault tolerance by automatically maintaining parity information or data copies. This requires a suitable coordination between DBMS, file system and disk array controller and constitutes a general (open) problem of disk arrays to be used for database processing.

SD does not require creation and maintenance of multiple independent indices (trees) for one logical index of a declustered relation. Still, a physical declustering of large indices should be possible for load balancing reasons. Such a declustering could be achieved transparently for the DBMS, e.g., by the file system. Furthermore, in contrast to SN there may be a different degree of declustering for relations and indices, and indices may be allocated to a disjoint set of disks. To reduce disk contention between different nodes of a SD system, frequently accessed indices (as well as relations) may be allocated to disks with a shared disk cache or to shared non-volatile semiconductor memory such as solid-state disk or extended memory [Ra91b, Ra92a, Ra93a].

SN proposals for data replication such as interleaved declustering or chained declustering have been shown to be applicable for media recovery in disk arrays as well [CK89, GLM92]. Similarly, these schemes can be applied to the SD environment. Furthermore, proposals for finding a data allocation for disk arrays [WZS91] can be adapted for SD.

While we focus on relational databases in this paper, it is to be noted that database partitioning will become more difficult for next-generation applications. For instance, large multimedia objects can be stored in a single tuple ("long field") so that they would be assigned to a single node in SN systems. Hence, parallelism cannot be utilized for SN to process such large objects. For SD, on the other hand, the object could physically be declustered across multiple disks so that at least I/O parallelism could be utilized to reduce I/O time. Similarly, complex objects for engineering or office automation applications are typically large and consist of many inter-connected and heterogeneous tuples. Partitioning these objects among multiple nodes is very difficult and would introduce a high communication overhead for object processing. Even partitioning at the object level is difficult due to subobjects that are shared by multiple complex objects. Hence, SD is better able to support intra-transaction parallelism on complex-object and object-oriented databases [HSS89].

3.3 Scan

Scan is the simplest and most common relational operator. It produces a row-and-column subset of a relation by applying a selection predicate and filtering away attributes not requested by the query. If predicate evaluation cannot be supported by an index, a complete *relation scan* is necessary where each tuple of the relation must be read and processed. An *index scan* accesses tuples via an index and restricts processing to a subset of the tuples; in the extreme case, no tuple or only one tuple needs to be accessed (e.g., exact-match query on unique attribute).

Shared Nothing

For SN, parallelizing a scan operation is straight-forward and determined by the database allocation. As discussed above, exact-match queries on the partitioning attribute are not parallelized for hash partitioning or range partitioning, in general, but are sent to the node where the corresponding tuple(s) can be found. Similarly, range queries on the partitioning attribute are localized to a subset of nodes for range partitioning. However, all other scan queries must be processed by all nodes holding fragments of the respective relation. Each node performs the scan on its local partition and sends back the result tuples to the query's home node, i.e. the node where the query has been initiated. At this node, a final sorting or duplicate elimination may be performed before the result is returned to the application/user.

The description shows that for all scan operations the degree of processing parallelism is statically determined by the database allocation. In particular, the degree of processing parallelism corresponds to the degree of declustering except for certain queries on the partitioning attribute. This approach has the obvious disadvantage that it does not allow dynamic load balancing, i.e., varying the number of processing nodes and selecting the scan nodes according to the current system state. Furthermore, for selective queries supported by an index it is generally inefficient to involve all D nodes holding a fragment of the relation due to an unfavorable ratio between communication overhead and useful work per node. The latter disadvantage can be reduced by a multidimensional range partitioning approach [GD90, GDQ92]. In this case, fragmentation is defined on multiple partitioning attributes so that queries on each of these attributes can be limited to a subset of the fragments/nodes (Fig. 2). While this approach can reduce communication overhead for certain queries compared to one-dimensional range partitioning, the degree of processing parallelism and thus the communication overhead are still statically determined by the database allocation.

		Salary				
		< 20 K	< 30 K	< 45 K	< 70 K	≥70 K
	A-E	1	2	3	4	5
	F-J	6	7	8	9	10
Name	K-O	11	12	13	14	15
	P-S	16	17	18	19	20
	T-Z	21	22	23	24	25

A two-dimensional range partitioning on the name and salary attributes is used for the employee relation. This allows exact-match and range queries on each of the two attributes be limited to 5 nodes if each fragment is assigned to a different node. A one-dimensional range partitioning could restrict queries on one of the two attributes to 1 node, but would involve all 25 nodes for queries on the other (resulting in an average of 13 nodes per query if both query types occur with the same frequency). Shared Disk permits both query types to be processed by a single processor thus incurring minimal communication overhead.

Fig. 2: Example of multi-dimensional range partitioning

Shared Disk

In SD systems each node has access to the entire database on disk. Hence, scan operations on a relation can be performed by any number of nodes. For example, index scans on any attribute may be performed by a single processor thereby minimizing communication overhead. This would especially be appropriate for exact-match and selective range queries, and supports high OLTP throughput. For relation scans, on the other hand, a high degree of processing parallelism can be employed to utilize intra-query parallelism to reduce response time and to achieve load balancing. Not only the degree of processing parallelism can be chosen based on a query's resource requirements, but also which processors should perform the scan operations. Furthermore, both scheduling decisions can be drawn according to the current system state. For instance, a scan may be allocated to a set of processors with low CPU utilization in order to avoid interference with concurrent transactions on other nodes. Furthermore, even multiple (independent) scans on the same relation may concurrently be executed on different nodes, e.g., if they access disjoint portions of the relation so that disk contention can be avoided. The latter feature is helpful for inter-transaction as well as inter-operator parallelism.
The ability to dynamically determine the degree of scan parallelism and the scan processors represents a key advantage of SD compared to SN. It is critical for a successful use of intra-transaction parallelism in multi-user mode where the current load situation is constantly changing. This is because the optimal degree of intra-query parallelism (yielding the best response time) strongly depends on the system state and is generally the lower the higher the system is utilized [MR93].

For SN, the chosen degree of declustering has a dominant effect on scan performance since it determines the degree of processing parallelism in many cases. For SD, on the other hand, disk declustering is only needed for I/O parallelism, in particular to provide sufficient I/O bandwidth for parallel processing. The degree of declustering merely determines the maximal degree of processing parallelism for SD, if no more than one node is accessing a disk for a given query to limit disk contention[5]. Since the degree of declustering has no direct impact on communication overhead for SD, it may be chosen higher than for SN to improve load balancing and to further extend flexibility with respect to choosing the appropriate degree of scan parallelism.

The result of a scan operation may be too large to be kept in main memory at the query's home node so that it must be stored in a temporary file on disk. In the case of SN, a high overhead is necessary to send the local scan results to the query's home node, to write out the data to disk and to read it in later to perform some postprocessing and return it to the application. SD can avoid the communication overhead for the data transfers since each scan processor can directly write its local scan result to a temporary file on the shared disks (or in shared semiconductor memory). After the temporary file is written, the query's home node is informed that the file can be read from external storage. To simplify the implemen-

5. For multiprocessor nodes with P processors per node, the maximal degree of processing parallelism would be P•D (D = degree of declustering).

Shared Disk

In SD systems each node has access to the entire database on disk. Hence, scan operations on a relation can be performed by any number of nodes. For example, index scans on any attribute may be performed by a single processor thereby minimizing communication overhead. This would especially be appropriate for exact-match and selective range queries, and supports high OLTP throughput. For relation scans, on the other hand, a high degree of processing parallelism can be employed to utilize intra-query parallelism to reduce response time and to achieve load balancing. Not only the degree of processing parallelism can be chosen based on a query's resource requirements, but also which processors should perform the scan operations. Furthermore, both scheduling decisions can be drawn according to the current system state. For instance, a scan may be allocated to a set of processors with low CPU utilization in order to avoid interference with concurrent transactions on other nodes. Furthermore, even multiple (independent) scans on the same relation may concurrently be executed on different nodes, e.g., if they access disjoint portions of the relation so that disk contention can be avoided. The latter feature is helpful for inter-transaction as well as inter-operator parallelism.
The ability to dynamically determine the degree of scan parallelism and the scan processors represents a key advantage of SD compared to SN. It is critical for a successful use of intra-transaction parallelism in multi-user mode where the current load situation is constantly changing. This is because the optimal degree of intra-query parallelism (yielding the best response time) strongly depends on the system state and is generally the lower the higher the system is utilized [MR93].

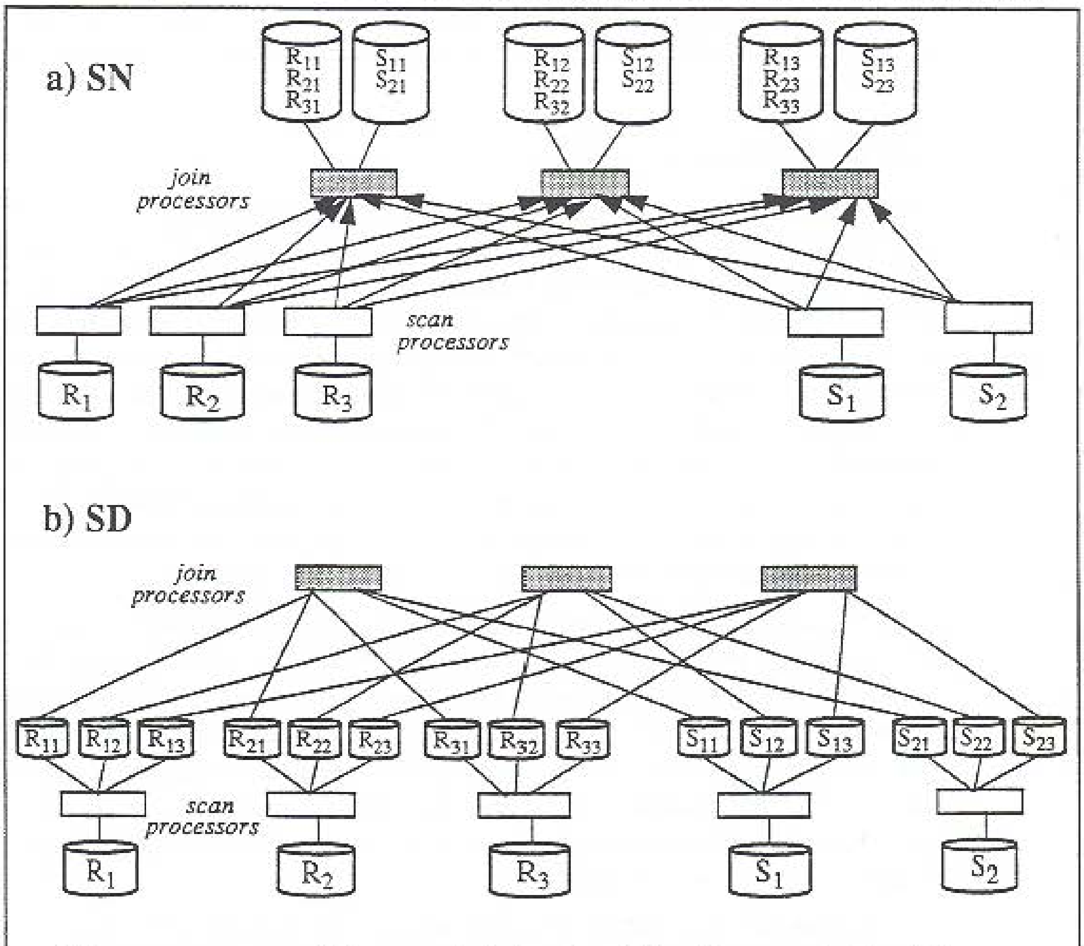
For SN, the chosen degree of declustering has a dominant effect on scan performance since it determines the degree of processing parallelism in many cases. For SD, on the other hand, disk declustering is only needed for I/O parallelism, in particular to provide sufficient I/O bandwidth for parallel processing. The degree of declustering merely determines the maximal degree of processing parallelism for SD, if no more than one node is accessing a disk for a given query to limit disk contention[5]. Since the degree of declustering has no direct impact on communication overhead for SD, it may be chosen higher than for SN to improve load balancing and to further extend flexibility with respect to choosing the appropriate degree of scan parallelism.

The result of a scan operation may be too large to be kept in main memory at the query's home node so that it must be stored in a temporary file on disk. In the case of SN, a high overhead is necessary to send the local scan results to the query's home node, to write out the data to disk and to read it in later to perform some postprocessing and return it to the application. SD can avoid the communication overhead for the data transfers since each scan processor can directly write its local scan result to a temporary file on the shared disks (or in shared semiconductor memory). After the temporary file is written, the query's home node is informed that the file can be read from external storage. To simplify the implemen-

5. For multiprocessor nodes with P processors per node, the maximal degree of processing parallelism would be P•D (D = degree of declustering).

disk contention problems could also be reduced by exchanging the data across shared semiconductor memory supporting both fast access times and high I/O rates.
If the size of the scan output allows a join processing in main memory, the data should also be exchanged across the communication system for SD. This is because such a data transfer is substantially faster than across disk. Furthermore, pipeline parallelism for local join processing can be utilized (e.g., if hash join is used as the local join method).

- Index-supported join queries that only require access to few tuples can be limited to one (or a few) node(s) for SD, while a high communication overhead may be necessary for SN. For instance, join queries for an employee relation fragmented as in Fig. 2 would require processing of all fragments for SN if the join attribute were, say, *serial number* or *department number*.



The scan output on each fragment of relations R and S is split among the three join processors. For SN, the scan output is tranferred to the join processors over the network. The join processors have to store their input fragments in temporary files on disk if they cannot be kept in memory. For SD, the scan output may be exchanged across the shared disks thereby avoiding the communication overhead for data transfers.

Fig. 3: Dynamic data redistribution between scan and join processors

A general SN approach to process θ-joins (non-equi joins) between two relations is to dynamically replicate the smaller relation at all nodes holding a fragment of the larger relation and to perform the θ-joins at the latter nodes [ÖV91]. This approach causes an enormous communication overhead and does not scale well since the communication overhead increases quadratically with the number of nodes (holding fragments of the two relations). SD avoids the communication overhead for data redistribution altogether since each node can directly read all fragments from disk. Furthermore, the number of join processors is not predetermined by the degree of declustering but can dynamically be selected. A high degree of declustering with respect to the disk allocation is favorable to reduce disk contention for SD. Disk contention is also reduced by the use of large main memory caches and disk caches.

3.5 Mixed workloads

Parallel database systems must be able to support both high throughput for OLTP as well as short response times for data-intensive queries of different types. This is difficult to achieve since both workload classes pose partially contradicting requirements. Some of the problems with respect to supporting mixed workloads already occur in centralized (or SE) DBMS. For instance, CPU and memory requirements for query processing may cause increased CPU waits and I/O delays for OLTP. Furthermore, data contention can be significant if queries request long read locks on large amounts of data. The latter problem may be resolved by using a multiversion concurrency control scheme which guarantees that read-only transactions do not cause or suffer from any lock conflicts [HP87, BC92, MPL92]. Resource contention may be controlled by the use of priorities for different transaction types, e.g., for buffer management [JCL90, Br92] and CPU scheduling[7].

In parallel database systems, there are two additional areas where performance problems for mixed workloads may occur: communication overhead and load balancing. Intra-query parallelism inevitably causes increased communication overhead (compared to a sequential execution on one node), leading to higher resource utilization and contention and therefore lower throughput. To limit the communication overhead and resource contention and to effectively utilize the available processing capacity, dynamic load balancing is particularly important for mixed workloads. As the preceding discussions have already shown, SD offers advantages over SN in both areas:

- SN cannot efficiently support both workload types, but requires definition of a (static) database allocation for an "average" transaction profile [GD90]. This inevitably leads to sub-optimal performance for both workload types and does not support dynamic load balancing. In particular, ad-hoc queries have to be restricted to fewer nodes than desirable to limit the communication overhead so that response times may not sufficiently be reduced. On the other hand, OLTP transactions cannot be confined to a single node in many cases thereby causing extra communication overhead and lowering throughput. In both cases, the sub-optimal performance must be accepted even if only one of the two workload types is temporarily active.

7. Much research is still needed for supporting a comprehensive priority-based load control for database applications. In particular, different subsystems like the DBMS, TP-monitor and the operating system must closely coordinate their scheduling decisions, e.g., to avoid anomalies such as the priority-inversion problem [En91].

- In SD systems, declustering of data across multiple disks does not increase the communication overhead for OLTP. In general, OLTP transactions are completely executed on one node to avoid the communication overhead for intra-transaction parallelism and distributed commit[8]. The degree of processing parallelism and thus the communication overhead for ad-hoc queries can be adapted to the current load situation. Furthermore, resource contention for CPU and memory between OLTP transactions and complex queries may largely be avoided by assigning these workload types to disjoint sets of processors which is not possible for SN, in general.

4 SD Transaction Management for Parallel Query Processing

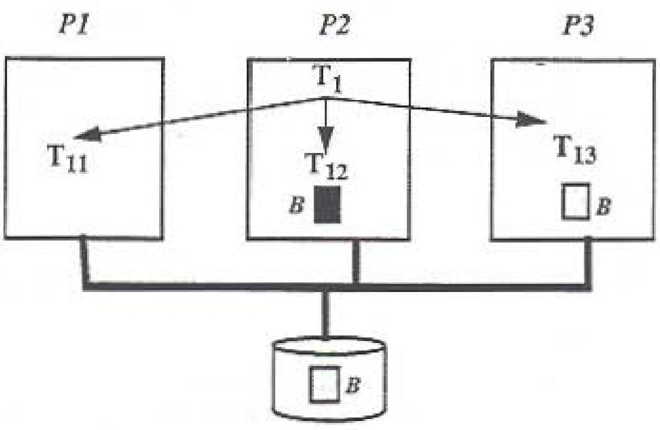
Without intra-transaction parallelism, there is no need for distributed transactions for SD. Each node can perform all database operations since the entire database is directly accessible. In particular, all modifications by a transaction are performed at one node and logged in this node's local log file. Hence, no distributed commit protocol is necessary as for SN to guarantee the ACID properties [HR83]. As mentioned in the introduction, communication is still necessary for global concurrency and coherency control. Furthermore, the local log files have to be merged into a global log to support media and crash recovery [Ra91a].

However, intra-transaction parallelism results in a decomposition of transactions and queries into multiple subtransactions running on different nodes. To guarantee the transaction (ACID) properties in this case, appropriate extensions with respect to transaction management are necessary for SD. While SN techniques for distributed transaction management (e.g., distributed commit protocols) can be used, there are additional problems for SD that require a more complex solution. These problems occur mainly if intra-transaction parallelism for update transaction is to be supported, and will be discussed first. To limit the implementation complexity, it seems desirable to restrict intra-transaction parallelism to read-only transactions (at least in a first step). This case is discussed in subsection 4.2.

4.1 Problems with parallel update transactions

In SN systems, the separation of global transactions into subtransactions is determined by the database allocation. This approach typically ensures that each subtransaction operates on data owned by the respective node. Concurrency control is a local function since each node can process all lock requests on its data. For SD, on the other hand, it cannot generally be excluded that subtransactions of a given transaction reference and modify the same database objects (e.g., index pages) at different nodes. Hence, there is a need for concurrency control between parallel subtransactions. Furthermore, coherency control is also required between parallel subtransactions to avoid access to obsolete data and to propagate updated database objects between subtransactions (Fig. 4).

8. Compared to a clustered data allocation where each relation is stored on a minimal number of disks, the number of disk I/Os for OLTP may be increased, however, if round robin or hash partitioning is used for declustering which distribute logically consecutive records to different disks. More disk I/Os could further increase disk contention between OLTP transactions. However, these problems can be avoided by a range partitioning which preserves clustering properties to a large extent [GHW90].



Subtransactions of transaction T_1 are running in parallel at processing nodes P1, P2 and P3. Subtransaction T_{12} at P2 has modified page B in main memory; the copies of B at P3 and on disk are obsolete. When subtransactions T_{11} and T_{13} want to read page B, a synchronization with T_{12} is necessary as well as a propagation of the new page version from P2 to P1 and P3.

Fig. 4: Concurrency/coherency control problem between subtransactions

The new requirements can be met by supporting a nested transaction model [Mo85, HR87] and by extending the SD concurrency/coherency control schemes for sequential transactions accordingly. Since the applications should remain unchanged compared to sequential transaction processing (unless inter-query parallelism is to be supported), nested transactions are only used internally by the DBMS to structure queries into a hierarchy of subtransactions or subqueries. Subtransactions can be executed concurrently at different nodes. Furthermore, subtransactions may be rolled back without impact on other subtransactions, i.e., the scope of undo recovery can be substantially limited compared to flat transactions. Isolation between subtransactions is achieved by a suitable locking protocol defining the rules for lock processing within a transaction hierarchy. Such a protocol supporting parallel subtransactions, upward and downward inheritance of locks as well as multiple lock granularities has been proposed in [HR87] and can be extended to the SD environment. One difference for SD results from the fact that lock requests may have to be sent to a global lock manager thus incurring communication overhead and delays[9]. Furthermore, coherency control must be incorporated into the concurrency control scheme. Fortunately, this can be accomplished in a similar way than for sequential transactions, e.g., by a so-called on-request invalidation protocol that uses extended global lock information to detect obsolete page copies and to record

9. In lock protocols for SD there is typically a global lock manager per database object [Ra91c]. Instead of using a central lock manager, the responsibility for global lock processing may be spread over all nodes (by partitioning the lock space) to balance the concurrency control overhead. Such an approach also saves the communication overhead for lock requests on "local" objects, i.e., for which the node processing the respective (sub)transaction is responsible. There are several techniques to reduce the number of remote lock requests [Ra91c] which can also be utilized for nested transactions.

where a page was modified most recently [Ra91c]. Such an approach detects obsolete page copies during lock processing so that extra messages for this purpose are avoided.

Although a detailed description of the extended concurrency/coherency control protocol is beyond the scope of this paper, we can illustrate some points by explaining how it would handle the situation shown in Fig. 4. Assume P2 is the global lock manager for page B so that the lock request by subtransaction T_{12} can be processed without communication delay. When T_{11} and T_{12} want to reference page B, they request a read lock at the global lock manager in P1 which is not granted since T_{12} holds a write lock. At the end of subtransaction T_{12}, the write lock is released and inherited to the parent transaction T_1 (as well as the new version of B). The global lock manager for B can now grant the read locks for subtransactions T_{11} and T_{13} (downward inheritance). Furthermore, the global lock manager detects that page B is also to be sent to nodes P1 and P3 (together with the message used for granting the lock). This is because nodes P1 and P3 cannot hold the current version of B since it was modified most recently at P2.

In addition to the extensions needed for concurrency and coherency control, parallel update transactions in SD systems require major changes for logging and recovery. In particular, the updates of a single transaction may now be performed at multiple nodes so that a transaction's log data are spread over multiple local log files. While this is also the case for SN, SN has the advantage that each local log file only contains log data of one database partition thereby supporting media recovery without the need for a global log file. Hence, parallel update transactions for SD would require support of the logging and recovery protocols of both SN (distributed commit, distributed undo and redo recovery) and SD (construction of a global log file).

One possibility to avoid the complexity and overhead of supporting both recovery paradigms is to log all updates of a nested transaction at the transaction's home node. This would require that during the commit of a subtransaction its log data are transferred to the home node where they are written to the local log file. Such an approach would result in a log situation as for SD without intra-transaction parallelism, so that the SN protocols could largely be avoided. However, such a scheme has its own problems. First, the transfers of the log data incur additional communication overhead and delays for subtransactions. Furthermore, the log data of subtransactions may have to be additionally kept in private log files to limit the number of log transfers and to support a local rollback of subtransactions. These private log files can introduce a substantial number of extra I/Os since some log information would have to be written several times.

The use of a global log for media recovery can be avoided by duplicating the database contents on disk (e.g., by using mirrored disks or implementing replication schemes such as interleaved or chained declustering [CK89, HD90]) or by maintaining appropriate parity information as in the case of RAID-5 disk arrays [PGK88]. These schemes support much faster media recovery than with a log-based approach, albeit at a considerable storage and update cost. In addition, despite the high degree of redundancy the probability of data loss (which occurs when multiple disks fail simultaneously) is not negligible for a high number of disks. Hence, a global log file may have to be supported additionally to meet high availability requirements. Furthermore, a global log may already be necessary for crash recovery depending on the update propagation strategy [Ra91a, MN91].

4.2 Parallelization of read-only transactions

Solving the problems introduced by parallel update transactions incurs a high implementation complexity. Fortunately, most of the performance gains of intra-transaction parallelism can already be expected by merely parallelizing read-only transactions since complex transactions/queries are mostly read-only. In this special case, update transactions are completely executed at one node. For typical OLTP transactions, response time limits can be met without intra-transaction parallelism. Furthermore, their sequential execution reduces communication overhead thereby supporting high transaction rates.

By always executing update transactions sequentially at one processing node, the problems mentioned above are avoided to a large extent. In particular, no changes are necessary with respect to logging and recovery compared to conventional SD systems. Furthermore, lock conflicts between concurrent subtransaction of the same transaction are avoided as well as the need for coherency control within transactions.

However, we still need a nested transaction model to propagate locks within the transaction hierarchy (e.g., between main transaction and subtransactions). Furthermore, a hierarchical lock protocol with multiple lock granularities (e.g., relation, index, page, record) is to be supported to keep the concurrency control overhead for complex queries small. Such a hierarchical protocol also permits limiting the communication overhead for global lock requests. For instance, a relation scan locks the entire relation which may be performed with one global lock request (several lock request messages are necessary if the lock responsibility for the relation is partitioned among multiple nodes). Subtransactions performing scan operations on different relation fragments in parallel do not have to request additional locks since the relation lock guarantees that there are no lock conflicts with concurrent update transactions. In terms of the nested transaction model, each subtransaction inherits a read lock from its parent transaction (representing the entire relation scan) on the relation fragment it processes. At the end of a subtransaction, the fragment lock is returned to the parent transaction which can finally release the entire relation lock with a single message.
For index scans, locking an entire relation is generally too restrictive with respect to concurrent update transactions so that page locking may be more appropriate. However, if there is no contention at the relation level when the index scan starts, a relation (and index) lock may be obtained at first to limit the number of global lock requests. These locks can later be de-escalated to page or record locks if concurrent transactions want to update the relation. Such a de-escalation protocol for SD has been proposed in [Jo91] and can also be used for parallel query processing.

Such a hierachical lock protocol has to be extended to support coherency control. For instance, if an on-request invalidation approach is used to limit the number of extra messages for coherency control, all coherency control information about a relation has to be returned to the respective transaction when a relation lock is granted. This information is then propagated to the individual subtransactions to eliminate all obsolete page versions of the relation that may still reside in main memory at the respective nodes. Furthermore, if the disk copy of updated pages of the relation is not yet updated, subtransactions may have to request the current page copies from the nodes where the most recent modification was performed. There are several alternatives for such an update propagation[10]:

- At the beginning of a subtransaction it requests all updated pages from the nodes where the most recent modification was performed (this information is recorded in the global lock table). This avoids later deactivations for page requests, but bears the risk that the requested pages may have to be buffered for a long time at the subtransaction's node. By the time such a page is to be processed, it may have already been replaced so that it must be read from disk.
- A subtransaction requests a modified page when it is actually to be processed. In this case, the page request results in a synchronous delay similar to a disk I/O. Furthermore, the node where the page was modified may have replaced (written out) the page in the meantime so that it has to be read from disk and the overhead for the page request was unnecessary. The number of these unnecessary page requests could be kept low by periodically broadcasting which pages have been written out.
- A subtransaction requests at its beginning that all updated pages to be processed by it are written out to the database on disk. In this case no page requests are necessary, but the current page versions can be read from disk. However, this approach causes a high number of disk accesses for writing out the modified pages and read them in later. Typically, a disk I/O takes considerably longer than a direct page transfer between nodes.

To limit data contention between read-only and update transactions, support of multi-version concurrency control is also desirable for SD. The implementation of such an approach requires that read-only transaction be provided with a consistent (but potentially outdated) version of the database. This can also be supported by recording in the global lock table where different versions of database objects can be obtained [Ra91c].

5 Conclusions

We have investigated the potential of Shared Disk (SD) database systems for parallel query processing. A detailed comparison with Shared Nothing (SN) systems revealed a number of significant advantages for SD[11]:

- SD supports intra-transaction parallelism with less communication overhead than SN. For SN, the database allocation statically determines the degree of parallelism and thus the communication overhead for scan, the most important relational operator. Even selective index scans may involve a high number of processing nodes for SN since only queries on one (or a few) attribute(s) can generally be restricted to a subset of the nodes holding fragments of the relation. SD allows a dynamic determination of the degree of scan parallelism so that selective queries can be processed with minimal communication overhead while large relation scans can be spread over many nodes to shorten response time.
- The communication overhead for parallel query processing is further reduced for SD by the possibility to exchange large intermediate results across shared storage devices rather than over the network.

10. We assume a NOFORCE strategy for writing modified pages to disk [HR83], since the FORCE alternative is typically unacceptable for high performance unless non-volatile semiconductor memory can be utilized [Ra92a].

11. Given the problems of SN, other researches have also advocated for more flexible alternatives. Valduriez favours a "shared-something" approach, that is a SD system in which each node is itself a multiprocessor [Va93b].

- SD supports a higher potential for dynamic load balancing than SN, in particular with respect to scan operations. Not only the degree of intra-transaction parallelism can be determined dynamically for SD but also which nodes should perform a given operation since each node can directly access all disks.
- SD can more efficiently support mixed OLTP/query workloads than SN and thus the use of intra-transaction parallelism in combination with inter-transaction parallelism. This is mainly because of the aforementioned points. In particular, OLTP transactions can always be executed on a single node to limit communication overhead and support high transaction rates. For complex queries, the use of intra-transaction parallelism to reduce response time can be dynamically adjusted to the current load situation (degree of parallelism, selection of processing nodes).
- SD is better positioned than SN to utilize disk arrays and to support parallel query processing for next-generation database applications (object-oriented DBMS) for which partitioning the database among multiple nodes is very difficult.

Of course, these advantages come not for free but must be achieved by suitable implementation techniques. Fortunately, many of the concepts and algorithms for parallel query processing developed for Shared Everything (multiprocessors) and SN can be adapted to the SD environment. Still, substantially more work is needed in several areas to fully exploit the potential of the SD architecture for parallel query processing. In particular, algorithms for transaction management (concurrency/coherency control, logging/recovery) have to be extended as discussed in section 4 especially if parallel processing of update transactions is to be supported. The potential for dynamic load balancing must be utilized by query processing strategies that base their scheduling decisions on information about the current system state. For this purpose, many strategies are conceivable and more work is needed to find out suitable heuristics that work well while incurring limited overhead.

Furthermore, the alternatives for disk allocation of the database need to be studied in more detail. While we believe that the disk allocation problem is easier solved than the database allocation problem for SN, there are a number of subtle issues to be considered for parallel query processing. In particular, a low disk contention must be supported by the disk allocation, query processing and by other means (use of disk caches, replication of data on disk and in main memory). In the case of disk arrays, the cooperation between disk controllers, file system, and DBMS with respect to parallel query processing and media recovery is another fruitful area for further research (for both SD and SE).

Finally, the performance advantages of SD outlined in this paper are mainly claims for the time being and must be validated by detailed performance studies. We have already conducted a number of simulation experiments on parallel query processing for SN with both synthetical and trace-driven workload models [MR92, MR93, RM93]. Currently, we are implementing a comprehensive simulation model for parallel query processing in SD systems to permit a performance comparison with SN.

6 References

Ap92 Apers, P.M.G. et al.: PRISMA/DB: A Parallel, Main-Memory Relational DBMS. *IEEE Trans. on Knowledge and Data Engineering* 4, 1992

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

Bh88 Bhide, A.: An Analysis of Three Transaction Processing Architectures. *Proc. of the 14th Int. Conf. on Very Large Data Bases*, Long Beach, CA, 339-350, 1988

Bo90 Boral, H. et al.: Prototyping Bubba: A Highly Parallel Database System. *IEEE Trans. on Knowledge and Data Engineering* 2, 1, 4-24, 1990

Br92 Brown, K.P. et al.: Resource Allocation and Scheduling for Mixed Database Workloads. TR 1095, Univ. of Wisconsin, Madison, 1992

CABK88 Copeland, G., Alexander, W., Boughter, E., Keller, T.: Data Placement in Bubba. *Proc. ACM SIGMOD Conf.*, 99-108, 1988

CK89 Copeland, G., Keller, T.: A Comparison of High-Availability Media Recovery Techniques. *Proc. ACM SIGMOD Conf.*, 98-109, 1989

CYW92 Chen, M., Yu, P.S., Wu, K.: Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries. *Proc. 8th IEEE Data Engineering Conf.*, 58-67, 1992

Da92 Davison, W.: Parallel Index Building in Informix OnLine 6.0. *Proc. ACM SIGMOD Conf.*, p. 103, 1992

De90 DeWitt, D.J. et al. 1990. The Gamma Database Machine Project. *IEEE Trans. on Knowledge and Data Engineering* 2 ,1, 44-62, 1990

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

DNSS92 DeWitt, D.J., Naughton, J.F., Schneider, D.A., Seshadri, S.: Practical Skew Handling in Parallel Joins. *Proc. of the 18th Int. Conf. on Very Large Data Bases*, 1992

EGKS90 Englert, S., Gray, J., Kocher, T., Shath, P.: A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scale-Up on Large Databases. *Proc. ACM SIGMETRICS Conf.*, 245-246, 1990

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

GD90 Ghandeharizadeh, S., DeWitt, D.J.: Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines. *Proc. 16th Int. Conf. on Very Large Data Bases*, 481-492, 1990

GDQ92 Ghandeharizadeh, S., DeWitt, D.J., Qureshi, W.: A Performance Analysis of Alternative Multi-Attribute Declustering Strategies. *Proc. ACM SIGMOD Conf.*, 29-38, 1992

GHW90 Gray, J., Horst, B., Walker, M.: Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput. *Proc. 16th Int. Conf. on Very Large Data Bases*, 148-161, 199

Gi93 Gietl, H.: nCUBE3 - The Way to Teraflops. *Proc. 5th Int. PARLE Conf. (Parallel Architectures and Languages Europe)*, Munich, June 1993

GLM92 Golubchik, L., Lui, J.C.S., Muntz, R.R.: Chained Declustering: Load Balancing and Robustness to Skew and Failures. *Proc. 2nd Int. Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, 88-95, IEEE Computer Society Press, 1992

Gr90 Graefe, G.: Encapsulation of Parallelism in the Volcano Query Processing System. *Proc. ACM SIGMOD Conf.*, 102-111, 1990

Gr91 Gray, J. (ed.): *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, 1991

Ha90 Haderle, D.: Parallelism with IBM's Relational Database2 (DB2). *Proc. IEEE Spring Comp Con Conf.*, 488-489, 1990

HD90 Hsiao, H., DeWitt, D.J.: Chained Declustering: a new Availability Strategy for Multiprocessor Database Machines. *Proc. 6th IEEE Data Engineering Conf.*, 456-465, 1990

HP87 Härder, T., Petry, E.: Evaluation of Multiple Version Scheme for Concurrency Control. *Information Systems* 12 (1), 83-98, 1987

HR83 Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys 15* (4), 287-317, 1983

HR87 Härder, T., Rothermel, K.: Concurrency Control Issues in Nested Transactions. IBM Research Report RJ 5803, San Jose, 1987, to appear in VLDB Journal (1993)

HSS89 Härder, T., Schöning, H., Sikeler, A.: Evaluation of Hardware Architectures for Parallel Execution of Complex Database Operations. *Proc. 3rd Annual Parallel Processing Symp.*, Fullerton, CA, 564-578, 1989

JCL90 Jauhari, R., Carey, M.J., Livny, M.: Priority-Hints: An Algorithm for Priority-Based Buffer Management. *Proc. 16th Int. Conf. on Very Large Data Bases*, 708-721, 1990

Jo91 Joshi, A.M.: Adaptive locking strategies in a multi-node data sharing environment. *Proc. of the 17th Int. Conf. on Very Large Data Bases*, 181-191, 1991

Ki84 Kim, W.: Highly Available Systems for Database Applications. *ACM Computing Surveys* 16 (1), 71-98, 1984

KLS86 Kronenberg, N.P., Levy, H.M., and Strecker, W.D.: VAX clusters: a Closely Coupled Distributed System. *ACM Trans. Computer Syst.* 4 (2), 130-146, 1986

Li93 Linder, B.: Oracle Parallel RDBMS on Massively Parallel Systems. *Proc. PDIS-93*, 67-68, 1993

LST91 Lu, H., Shan, M., Tan, K.: Optimization of Multi-Way Join Queries for Parallel Execution. *Proc. 17th Int. Conf. on Very Large Data Bases*, 549-560,1991

LY89 Lorie, R.A. Young, H.C.: A Low Communication Sort Algorithm for a Parallel Database Machine. *Proc. 15th Int. Conf. on Very Large Data Bases*, 125-134, 1989

Mo85 Moss, J.E.B.: *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985

MR92 Marek, R., Rahm, E.: Performance Evaluation of Parallel Transaktion Processing in Shared Nothing Database Systems. *Proc. 4th Int. PARLE Conf.* (Parallel Architectures and Languages Europe), Springer-Verlag, Lecture Notes in Computer Science 605, 295-310, 1992

MR93 Marek, R., Rahm, E.: On the Performance of Parallel Join Processing in Shared Nothing Database Systems. *Proc. 5th Int. PARLE Conf.* (Parallel Architectures and Languages Europe), Springer-Verlag, Lecture Notes in Computer Science, Munich, June 1993

MLO86 Mohan, C., Lindsay, B., Obermarck, R.: Transaction Management in the R* Distributed Database Management System. *ACM Trans. on Database System 11* (4), 378-396, 1986

MN91 Mohan, C., Narang, I.: Recovery and Coherency-control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment. *Proc. 17th Int. Conf. on Very Large Data Bases*, Barcelona, 193-207, 1991

MPL92 Mohan, C., Pirahesh, H., Lorie, R.: Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions. *Proc. ACM SIGMOD Conf.*, 124-133, 1992

Ne86 Neches, P.M.: The Anatomy of a Database Computer - Revisited. *Proc. IEEE CompCon Spring Conf.*, 374-377, 1986

Or91 TPC Benchmark B - Full disclosure report for the nCUBE 2 scalar supercomputer model nCDB-1000 using Oracle V6.2. Oracle Corporation, part number 3000097-0391, July 1991

ÖV91 Özsu, M.T., Valduriez, P.: *Principles of Distributed Database Systems*. Prentice Hall, 1991

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

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

Ra86 Rahm, E.: Primary Copy Synchronization for DB-Sharing. *Information Systems* 11 (4), 275-286, 1986

Ra91a Rahm, E.: Recovery Concepts for Data Sharing Systems. *Proc. 21st Int. Symp. on Fault-Tolerant Computing* (FTCS-21), Montreal, 368-375, 1991

Ra91b Rahm, E.: Use of Global Extended Memory for Distributed Transaction Processing. *Proc. 4th Int. Workshop on High Performance Transaction Systems*, Asilomar, Sep. 1991

Ra91c	Rahm, E.: Concurrency and Coherency Control in Database Sharing Systems, Techn. Report 3/91, Univ. Kaiserslautern, Dept. of Comp. Science, Dec. 1991
Ra92a	Rahm, E.: Performance Evaluation of Extended Storage Architectures for Transaction Processing. *Proc. ACM SIGMOD Conf.*, San Diego, CA, 308-317, 1992
Ra92b	Rahm, E.: A Framework for Workload Allocation in Distributed Transaction Processing Systems. *Journal of Systems and Software* 18, 171-190, 1992
Ra93a	Rahm, E.: Evaluation of Closely Coupled Systems for High Performance Database Systems. *Proc. 13th Int. Conf. on Distributed Computing Systems*, Pittsburgh,301-310, 1993
Ra93b	Rahm, E.: Empirical Performance Evaluation of Concurrency and Coherency Control for Database Sharing Systems. *ACM Trans. on Database Systems* 18 (2), 333-377, 1993
RM93	Rahm, E., Marek, R.: Analysis of Dynamic Load Balancing Strategies for Parallel Shared Nothing Database Systems, *Proc. 19th Int. Conf. on Very Large Data Bases*, 1993
SD90	Schneider, D.A., DeWitt, D.J.: Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. *Proc. 16th Int. Conf. on Very Large Data Bases*, 469-480, 1990
Sh86	Shoens, K.: Data Sharing vs. Partitioning for Capacity and Availability. *IEEE Database Engineering* 9, 1, 10-16, 1986
Sk92	Skelton, C.J. et al.: EDS: A Parallel Computer System for Advanced Information Processing, *Proc. 4th Int. PARLE Conf.* (Parallel Architectures and Languages Europe), Springer-Verlag, Lecture Notes in Computer Science 605, 3-18, 1992
SKPO88	Stonebraker, M.; Katz, R.; Patterson, D.; Ousterhout, J.: The Design of XPRS. *Proc. 14th Int. Conf. on Very Large Data Bases*, 318-330, 1988
St86	Stonebraker, M.: The Case for Shared Nothing. *IEEE Database Engineering* 9 (1), 4-9, 1986
Ta88	The Tandem Performance Group: A Benchmark of NonStop SQL on the Debit Credit Transaction. *Proc. ACM SIGMOD Conf.*, 337-341, 1988
Ta89	The Tandem Database Group: NonStop SQL, A Distributed, High-Performance, High-Availability Implementation of SQL. In Lecture Notes in Computer Science 359, Springer-Verlag, 60-104, 1989
Tr83	Traiger, I.: Trends in Systems Aspects of Database Management. *Proc. of the British Computer Society* 2nd Int.Conf. on Databases, 1-20, 1983
Va93a	Valduriez, P.: Parallel Database Systems: Open Problems and New Issues. *Distr. and Parallel Databases* 1 (2), 137-165, 1993
Va93b	Valduriez, P.: Parallel Database Systems: The Case for Shared-Something. *Proc. 9th Int. Conf. on Data Engineering*, 460-465, 1993
WZS91	Weikum, G., Zabback, P., Scheuermann, P.: Dynamic File Allocation in Disk Arrays. *Proc. ACM SIGMOD Conf.*, 406-415, 1991
YD91	Yu, P.S., Dan, A.: Comparison on the Impact of Coupling Architectures to the Performance of Transaction Processing Systems. *Proc. 4th Int. Workshop on High Performance Transaction Systems*, 1991
Yu87	Yu, P.S. et al.: On Coupling Multi-systems through Data Sharing. *Proceedings of the IEEE* 75 (5), 573-587, 1987
Ze90	Zeller, H.: Parallel Query Execution in NonStop SQL. *Proc. IEEE Spring CompCon Conf.*, 484-487, 1990