Chapter 8
Parallel Database Systems

Many data-intensive applications require support for very large databases (e.g., hundreds of terabytes or exabytes). Supporting very large databases efficiently for either OLTP or OLAP can be addressed by combining parallel computing and distributed database management.

A parallel computer, or multiprocessor, is a form of distributed system made of a number of nodes (processors, memories, and disks) connected by a very fast network within one or more cabinets in the same room. There are two kinds of multiprocessors depending on how these nodes are coupled: tightly coupled and loosely coupled. Tightly coupled multiprocessors contain multiple processors that are connected at the bus level with a shared-memory. Mainframe computers, supercomputers, and the modern multicore processors all use tight-coupling to boost performance. Loosely coupled multiprocessors, now referred to as computer clusters, or clusters for short, are based on multiple commodity computers interconnected via a high-speed network. The main idea is to build a powerful computer out of many small nodes, each with a very good cost/performance ratio, at a much lower cost than equivalent mainframe or supercomputers. In its cheapest form, the interconnect can be a local network. However, there are now fast standard interconnects for clusters (e.g., Infiniband and Myrinet) that provide high bandwidth (e.g., 100 Gigabits/sec) with low latency for message traffic.

As already discussed in previous chapters, data distribution can be exploited to increase performance (through parallelism) and availability (through replication). This principle can be used to implement *parallel database systems*, i.e., database systems on parallel computers. Parallel database systems can exploit the parallelism in data management in order to deliver high-performance and high-availability database servers. Thus, they can support very large databases with very high loads.

Most of the research on parallel database systems has been done in the context of the relational model because it provides a good basis for parallel data processing. In

The original version of this chapter was revised. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-26253-2_13

M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*,
https://doi.org/10.1007/978-3-030-26253-2_8

this chapter, we present the parallel database system approach as a solution to high-performance and high-availability data management. We discuss the advantages and disadvantages of the various parallel system architectures and we present the generic implementation techniques.

Implementation of parallel database systems naturally relies on distributed database techniques. However, the critical issues are data placement, parallel query processing, and load balancing because the number of nodes may be much higher than the number of sites in a distributed DBMS. Furthermore, a parallel computer typically provides reliable, fast communication that can be exploited to efficiently implement distributed transaction management and replication. Therefore, although the basic principles are the same as in distributed DBMS, the techniques for parallel database systems are fairly different.

This chapter is organized as follows: In Sect. 8.1, we clarify the objectives of parallel database systems. In Sect. 8.2, we discuss architectures, in particular, shared-memory, shared-disk, and shared-nothing. Then, we present the techniques for data placement in Sect. 8.3, query processing in Sect. 8.4, load balancing in Sect. 8.5, and fault-tolerance in Sect. 8.6. In Sect. 8.7, we present the use of parallel data management techniques in database clusters, an important type of parallel database system.

8.1 Objectives

Parallel processing exploits multiprocessor computers to run application programs by using several processors cooperatively, in order to improve performance. Its prominent use has long been in scientific computing to improve the response time of numerical applications. The developments in both general-purpose parallel computers using standard microprocessors and parallel programming techniques have enabled parallel processing to break into the data processing field.

Parallel database systems combine database management and parallel processing to increase performance and availability. Note that performance was also the objective of *database machines* in the 1980s. The problem faced by conventional database management has long been known as "I/O bottleneck," induced by high disk access time with respect to main memory access time (typically hundreds of thousands times faster). Initially, database machine designers tackled this problem through special-purpose hardware, e.g., by introducing data filtering devices within the disk heads. However, this approach failed because of poor cost/performance compared to the software solution, which can easily benefit from hardware progress in silicon technology. The idea of pushing database functions closer to disk has received renewed interest with the introduction of general-purpose microprocessors in disk controllers, thus leading to intelligent disks. For instance, basic functions that require costly sequential scan, e.g., select operations on tables with fuzzy predicates, can be more efficiently performed at the disk level since they avoid overloading the DBMS memory with irrelevant disk blocks. However, exploiting intelligent disks requires adapting the DBMS, in particular, the query processor to decide whether to use the disk functions. Since there is no standard intelligent disk technology, adapting to different intelligent disk technologies hurts DBMS portability.

An important result, however, is in the general solution to the I/O bottleneck. We can summarize this solution as *increasing the I/O bandwidth through parallelism.* For instance, if we store a database of size D on a single disk with throughput T, the system throughput is bounded by T. On the contrary, if we partition the database across n disks, each with capacity D/n and throughput T' (hopefully equivalent to T), we get an ideal throughput of $n * T'$ that can be better consumed by multiple processors (ideally n). Note that the main memory database system solution, which tries to maintain the database in main memory, is complementary rather than alternative. In particular, the "memory access bottleneck" in main memory systems can also be tackled using parallelism in a similar way. Therefore, parallel database system designers have strived to develop software-oriented solutions in order to exploit parallel computers.

A parallel database system can be loosely defined as a DBMS implemented on a parallel computer. This definition includes many alternatives ranging from the straightforward porting of an existing DBMS, which may require only rewriting the operating system interface routines, to a sophisticated combination of parallel processing and database system functions into a new hardware/software architecture. As always, we have the traditional trade-off between portability (to several platforms) and efficiency. The sophisticated approach is better able to fully exploit the opportunities offered by a multiprocessor at the expense of portability. Interestingly, this gives different advantages to computer manufacturers and software vendors. It is therefore important to characterize the main points in the space of alternative parallel system architectures. In order to do so, we will make precise the parallel database system solution and the necessary functions. This will be useful in comparing the parallel database system architectures.

The objectives of parallel database systems are similar to those of distributed DBMSs (performance, availability, extensibility), but have somewhat different focus due to the tighter coupling of computing/storage nodes. We highlight these below.

1. **High performance.** This can be obtained through several complementary solutions: parallel data management, query optimization, and load balancing. Parallelism can be used to increase throughput and decrease transaction response times. However, decreasing the response time of a complex query through large-scale parallelism may well increase its total time (by additional communication) and hurt throughput as a side-effect. Therefore, it is crucial to optimize and parallelize queries in order to minimize the overhead of parallelism, e.g., by constraining the degree of parallelism for the query. *Load balancing* is the ability of the system to divide a given workload equally among all processors. Depending on the parallel system architecture, it can be achieved statically by appropriate physical database design or dynamically at runtime.
2. **High availability.** Because a parallel database system consists of many redundant components, it can well increase data availability and fault-tolerance. In a highly parallel system with many nodes, the probability of a node failure at

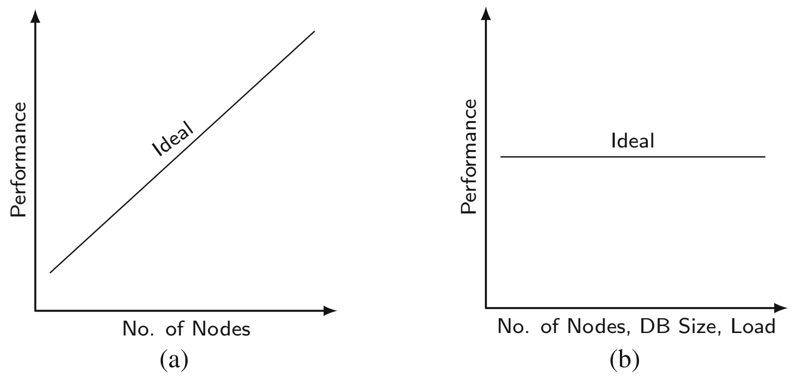


Fig. 8.1 Extensibility metrics. (**a**) Linear speed-up. (**b**) Linear scale-up

any time can be relatively high. Replicating data at several nodes is useful to support *failover*, a fault-tolerance technique that enables automatic redirection of transactions from a failed node to another node that stores a copy of the data. This provides uninterrupted service to users.

3. **Extensibility.** In a parallel system, accommodating increasing database sizes or increasing performance demands (e.g., throughput) should be easier. Extensibility is the ability to expand the system smoothly by adding processing and storage power to the system. Ideally, the parallel database system should demonstrate two extensibility advantages: *linear speed-up* and *linear scale-up* (see Fig. 8.1). Linear speed-up refers to a linear increase in performance for a constant database size and load while the number of nodes (i.e., processing and storage power) is increased linearly. Linear scale-up refers to a sustained performance for a linear increase in both database size, load and number of nodes. Furthermore, extending the system should require minimal reorganization of the existing database.

The increasing use of clusters in large-scale applications, e.g., web data management, has led to the use of the term *scale-out* versus *scale-up*. Figure 8.2 shows a cluster with 4 servers, each with a number of processing nodes ("Ps"). In this context, scale-up (also called vertical scaling) refers to adding more nodes to a server and thus gets limited by the maximum size of the server. Scale-out (also called horizontal scaling) refers to adding more servers, called "scale-out servers" in a loosely coupled fashion, to scale almost infinitely.

8.2 Parallel Architectures

A parallel database system represents a compromise in design choices in order to provide the aforementioned advantages with a good cost/performance. One guiding design decision is the way the main hardware elements, i.e., processors,

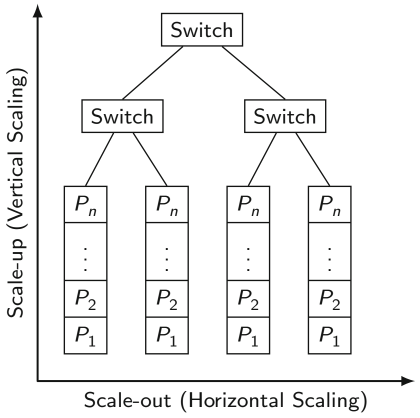


Fig. 8.2 Scale-up versus scale-out

main memory, and disks, are connected through some interconnection network. In this section, we present the architectural aspects of parallel database systems. In particular, we present and compare the three basic parallel architectures: *shared-memory*, *shared-disk*, and *shared-nothing*. Shared-memory is used in tightly coupled multiprocessors, while shared-nothing and shared-disk are used in clusters. When describing these architectures, we focus on the four main hardware elements: interconnect, processors (P), main memory modules (M), and disks. For simplicity, we ignore other elements such as processor cache, processor cores, and I/O bus.

8.2.1 General Architecture

Assuming a client/server architecture, the functions supported by a parallel database system can be divided into three subsystems much like in a typical DBMS. The differences, though, have to do with implementation of these functions, which must now deal with parallelism, data partitioning and replication, and distributed transactions. Depending on the architecture, a processor node can support all (or a subset) of these subsystems. Figure 8.3 shows the architecture using these subsystems, which is based on the architecture of Fig. 1.11 with the addition of a client manager.

1. **Client manager.** It provides support for client interactions with the parallel database system. In particular, it manages the connections and disconnections between the client processes, which run on different servers, e.g., application servers, and the query processors. Therefore, it initiates client queries (which may be transactions) at some query processors, which then become responsible for interacting directly with the clients and perform query processing and transaction management. The client manager also performs load balancing, using

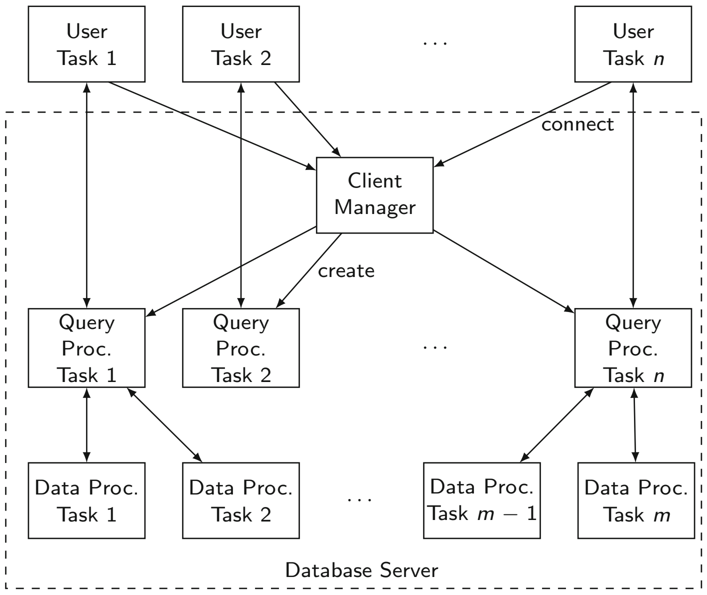


Fig. 8.3 General architecture of a parallel database system

a catalog that maintains information on processor nodes' load and precompiled queries (including data location). This allows triggering precompiled query executions at query processors that are located close to the data that is accessed. The client manager is a lightweight process, and thus not a bottleneck. However, for fault-tolerance, it can be replicated at several nodes.

2. **Query processor.** It receives and manages client queries, such as compile query, execute query, and start transaction. It uses the database directory that holds all metainformation about data, queries, and transactions. The directory itself should be managed as a database, which can be replicated at all query processor nodes. Depending on the request, it activates the various compilation phases, including semantic data control and query optimization and parallelization, triggers and monitors query execution using the data processors, and returns the results as well as error codes to the client. It may also trigger transaction validation at the data processors.
3. **Data processor.** It manages the database's data and system data (system log, etc.) and provides all the low-level functions needed to execute queries in parallel, i.e., database operator execution, parallel transaction support, cache management, etc.

8.2.2 *Shared-Memory*

In the shared-memory approach, any processor has access to any memory module or disk unit through an interconnect. All the processors are under the control of a single operating system.

One major advantage is simplicity of the programming model based on shared virtual memory. Since metainformation (directory) and control information (e.g., lock tables) can be shared by all processors, writing database software is not very different than for single processor computers. In particular, interquery parallelism comes for free. Intraquery parallelism requires some parallelization but remains rather simple. Load balancing is also easy since it can be achieved at runtime using the shared-memory by allocating each new task to the least busy processor.

Depending on whether physical memory is shared, two approaches are possible: Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA), which we present below.

8.2.2.1 Uniform Memory Access (UMA)

With UMA, the physical memory is shared by all processors, so access to memory is in constant time (see Fig. 8.4). Thus, it has also been called *symmetric multiprocessor (SMP)*. Common network topologies to interconnect processors include bus, crossbar, and mesh.

The first SMPs appeared in the 1960s for mainframe computers and had a few processors. In the 1980s, there were larger SMP machines with tens of processors. However, they suffered from high cost and limited scalability. High cost was incurred by the interconnect that requires fairly complex hardware because of the need to link each processor to each memory module or disk. With faster and faster processors (even with larger caches), conflicting accesses to the shared-memory

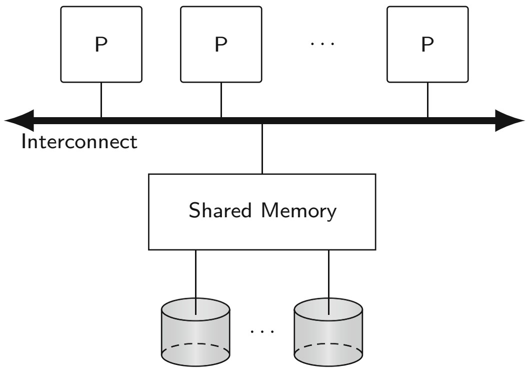


Fig. 8.4 Shared-memory

increase rapidly and degrade performance. Therefore, scalability has been limited to less than ten processors. Finally, since the memory space is shared by all processors, a memory fault may affect most processors, thereby hurting data availability.

Multicore processors are also based on SMP, with multiple processing cores and shared-memory on a single chip. Compared to the previous multichip SMP designs, they improve the performance of cache operations, require much less printed circuit board space, and consume less energy. Therefore, the current trend in multicore processor development is towards an ever increasing number of cores, as processors with hundreds of cores become feasible.

Examples of SMP parallel database systems include XPRS, DBS3, and Volcano.

8.2.2.2 Non-Uniform Memory Access (NUMA)

The objective of NUMA is to provide a shared-memory programming model and all its benefits, in a scalable architecture with distributed memory. Each processor has its own local memory module, which it can access efficiently. The term NUMA reflects the fact that accesses to the (virtually) shared-memory have a different cost depending on whether the physical memory is local or remote to the processor.

The oldest class of NUMA systems is Cache Coherent NUMA (CC-NUMA) multiprocessors (see Fig. 8.5). Since different processors can access the same data in a conflicting update mode, global cache consistency protocols are needed. In order to make remote memory access efficient, one solution is to have cache consistency done in hardware through a special consistent cache interconnect. Because shared-memory and cache consistency are supported by hardware, remote memory access is very efficient, only several times (typically up to 3 times) the cost of local access.

A more recent approach to NUMA is to exploit the Remote Direct Memory Access (RDMA) capability that is now provided by low latency cluster interconnects such as Infiniband and Myrinet. RDMA is implemented in the network card hardware and provides zero-copy networking, which allows a cluster node to directly access the memory of another node without any copying between operating system buffers. This yields typical remote memory access at latencies of the order of 10 times a local memory access. However, there is still room for improvement.

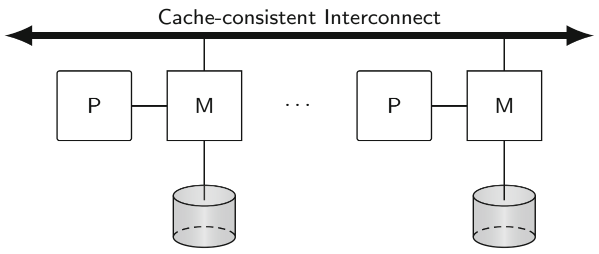


Fig. 8.5 Cache coherent non-uniform memory architecture (CC-NUMA)

For instance, the tighter integration of remote memory control into the node's local coherence hierarchy yields remote access at latencies that are within 4 times a local access. Thus, RDMA can be exploited to improve the performance of parallel database operations. However, it requires new algorithms that are NUMA aware in order to deal with the remote memory access bottleneck. The basic approach is to maximize local memory access by careful scheduling of DBMS tasks close to the data and to interleave computation and network communication.

Modern multiprocessors use a hierarchical architecture that mixes NUMA and UMA, i.e., a NUMA multiprocessor where each processor is a multicore processor. In turn, each NUMA multiprocessor can be used as a node in a cluster.

8.2.3 Shared-Disk

In a shared-disk cluster (see Fig. 8.6), any processor has access to any disk unit through the interconnect but exclusive (nonshared) access to its main memory. Each processor–memory node, which can be a shared-memory node is under the control of its own copy of the operating system. Then, each processor can access database pages on the shared-disk and cache them into its own memory. Since different processors can access the same page in conflicting update modes, global cache consistency is needed. This is typically achieved using a distributed lock manager that can be implemented using the techniques described in Chap. 5. The first parallel DBMS that used shared-disk is Oracle with an efficient implementation of a distributed lock manager for cache consistency. It has evolved to the Oracle Exadata database machine. Other major DBMS vendors such as IBM, Microsoft, and Sybase also provide shared-disk implementations, typically for OLTP workloads.

Shared-disk requires disks to be globally accessible by the cluster nodes. There are two main technologies to share disks in a cluster: network-attached storage (NAS) and storage-area network (SAN). A NAS is a dedicated device to shared-disks over a network (usually TCP/IP) using a distributed file system protocol such as Network File System (NFS). NAS is well-suited for low throughput applications such as data backup and archiving from PC's hard disks. However, it is relatively slow and not appropriate for database management as it quickly

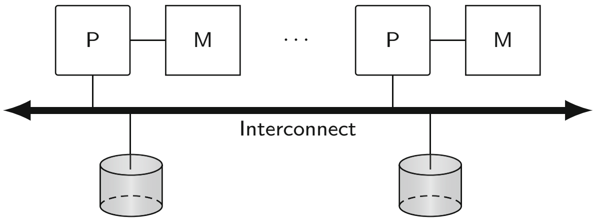


Fig. 8.6 Shared-disk architecture

becomes a bottleneck with many nodes. A storage-area network (SAN) provides similar functionality but with a lower level interface. For efficiency, it uses a block-based protocol, thus making it easier to manage cache consistency (at the block level). As a result, SAN provides high data throughput and can scale up to large numbers of nodes.

Shared-disk has three main advantages: simple and cheap administration, high availability, and good load balance. Database administrators do not need to deal with complex data partitioning, and the failure of a node only affects its cached data while the data on disk is still available to the other nodes. Furthermore, load balancing is easy as any request can be processed by any processor–memory node. The main disadvantages are cost (because of SAN) and limited scalability, which is caused by the potential bottleneck and overhead of cache coherence protocols for very large databases. A solution is to rely on data partitioning as in shared-nothing, at the expense of more complex administration.

8.2.4 Shared-Nothing

In a shared-nothing cluster (see Fig. 8.7), each processor has exclusive access to its main memory and disk, using Directly Attached Storage (DAS).

Each processor–memory–disk node is under the control of its own copy of the operating system. Shared-nothing clusters are widely used in practice, typically using NUMA nodes, because they can provide the best cost/performance ratio and scale up to very large configurations (thousands of nodes).

Each node can be viewed as a local site (with its own database and software) in a distributed DBMS. Therefore, most solutions designed for those systems such as database fragmentation, distributed transaction management, and distributed query processing may be reused. Using a fast interconnect, it is possible to accommodate large numbers of nodes. As opposed to SMP, this architecture is often called Massively Parallel Processor (MPP).

By favoring the smooth incremental growth of the system by the addition of new nodes, shared-nothing provides extensibility and scalability. However, it requires

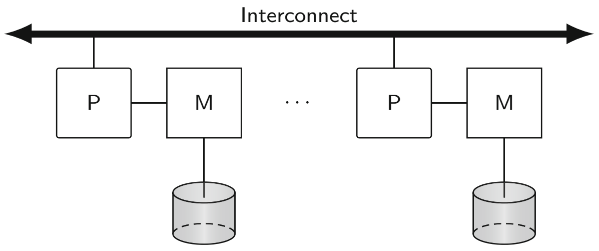


Fig. 8.7 Shared-nothing architecture

careful partitioning of the data on multiple disks. Furthermore, the addition of new nodes in the system presumably requires reorganizing and repartitioning the database to deal with the load balancing issues. Finally, node fault-tolerance is difficult (requires replication) as a failed node will make its data on disk unavailable.

Many parallel database system prototypes have adopted the shared-nothing architecture, e.g., Bubba, Gamma, Grace, and Prisma/DB. The first major parallel DBMS product was Teradata's database machine. Other major DBMS companies such as IBM, Microsoft, and Sybase and vendors of column-store DBMS such as MonetDB and Vertica provide shared-nothing implementations for high-end OLAP applications. Finally, NoSQL DBMSs and big data systems typically use shared-nothing.

Note that it is possible to have a hybrid architecture, where part of the cluster is shared-nothing, e.g., for OLAP workloads, and part is shared-disk, e.g., for OLTP workloads. For instance, Teradata supports the concept of *clique*, i.e., a set of nodes that share a common set of disks, to its shared-nothing architecture to improve availability.

8.3 Data Placement

In the rest of this chapter, we consider a shared-nothing architecture because it is the most general case and its implementation techniques also apply, sometimes in a simplified form, to the other architectures. Data placement in a parallel database system exhibits similarities with data fragmentation in distributed databases (see Chap. 2). An obvious similarity is that fragmentation can be used to increase parallelism. As noted in Chap. 2, parallel DBMSs mostly use horizontal partitioning, although vertical fragmentation can also be used to increase parallelism and load balancing much as in distributed databases and has been employed in column-store DBMSs, such as MonetDB or Vertica. Another similarity with distributed databases is that since data is much larger than programs, execution should occur, as much as possible, where the data resides. As noted in Chap. 2, there are two important differences with the distributed database approach. First, there is no need to maximize local processing (at each node) since users are not associated with particular nodes. Second, load balancing is much more difficult to achieve in the presence of a large number of nodes. The main problem is to avoid resource contention, which may result in the entire system thrashing (e.g., one node ends up doing all the work, while the others remain idle). Since programs are executed where the data resides, data placement is critical for performance.

The most common data partitioning strategies that are used in parallel DBMSs are the round-robin, hashing, and range-partitioning approaches discussed in Sect. 2.1.1. Data partitioning must scale with the increase in database size and load. Thus, the degree of partitioning, i.e., the number of nodes over which a relation is partitioned, should be a function of the size and access frequency of the relation. Therefore, increasing the degree of partitioning may result in placement reorganization. For example, a relation initially placed across eight nodes may have its cardinality doubled by subsequent insertions, in which case it should be placed across 16 nodes.

In a highly parallel system with data partitioning, periodic reorganizations for load balancing are essential and should be frequent unless the workload is fairly static and experiences only a few updates. Such reorganizations should remain transparent to compiled queries that run on the database server. In particular, queries should not be recompiled because of reorganization and should remain independent of data location, which may change rapidly. Such independence can be achieved if the runtime system supports associative access to distributed data. This is different from a distributed DBMS, where associative access is achieved at compile time by the query processor using the data directory.

One solution to associative access is to have a global index mechanism replicated on each node. The global index indicates the placement of a relation onto a set of nodes. Conceptually, the global index is a two-level index with a major clustering on the relation name and a minor clustering on some attribute of the relation. This global index supports variable partitioning, where each relation has a different degree of partitioning. The index structure can be based on hashing or on a B-tree like organization. In both cases, exact-match queries can be processed efficiently with a single node access. However, with hashing, range queries are processed by accessing all the nodes that contain data from the queried relation. Using a B-tree index (usually much larger than a hash index) enables more efficient processing of range queries, where only the nodes containing data in the specified range are accessed.

Example 8.1 Figure 8.8 provides an example of a global index and a local index for relation EMP(ENO, ENAME, TITLE) of the engineering database example we have been using in this book.

Suppose that we want to locate the elements in relation EMP with ENO value "E50." The first-level index maps the name EMP onto the index on attribute ENO

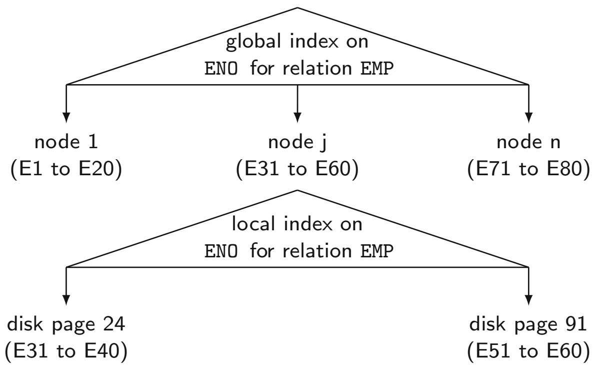


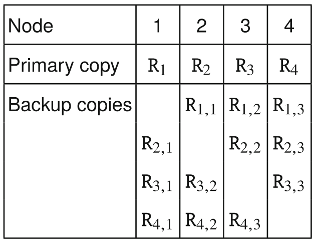
Fig. 8.8 Example of global and local indexes

for relation EMP. Then, the second-level index further maps the cluster value "E50" onto node number j. A local index within each node is also necessary to map a relation onto a set of disk pages within the node. The local index has two levels, with a major clustering on relation name and a minor clustering on some attribute. The minor clustering attribute for the local index is the *same* as that for the global index. Thus, *associative routing* is improved from one node to another based on (relation name, cluster value). This local index further maps the cluster value "E5" onto page number 91. ♦

A serious problem in data placement is dealing with skewed data distributions that may lead to nonuniform partitioning and hurt load balancing. A solution is to treat nonuniform partitions appropriately, e.g., by further fragmenting large partitions. This is easy with range partitioning, since a partition can be split as a B-tree leaf, with some local index reorganization. With hashing, the solution is to use a different hash function on a different attribute, The separation between logical and physical nodes is useful here since a logical node may correspond to several physical nodes.

A final complicating factor for data placement is data replication for high availability, which we discussed at length in Chap. 6. In parallel DBMSs, simpler approaches might be adopted, such as the *mirrored disks* architecture where two copies of the same data are maintained: a primary and a backup copy. However, in case of a node failure, the load of the node with the copy may double, thereby hurting load balance. To avoid this problem, several high-availability data replication strategies have been proposed for parallel database systems. An interesting solution is Teradata's interleaved partitioning that further partitions the backup copy on a number of nodes. Figure 8.9 illustrates the interleaved partitioning of relation R over four nodes, where each primary copy of a partition, e.g., R_1, is further divided into three partitions, e.g., $R_{1,1}$, $R_{1,2}$, and $R_{1,3}$, each at a different backup node. In failure mode, the load of the primary copy gets balanced among the backup copy nodes. But if two nodes fail, then the relation cannot be accessed, thereby hurting availability. Reconstructing the primary copy from its separate backup copies may be costly. In normal mode, maintaining copy consistency may also be costly.

An alternative solution is Gamma's *chained partitioning* , which stores the primary and backup copy on two adjacent nodes (Fig. 8.10). The main idea is that



Node	1	2	3	4
Primary copy	R_1	R_2	R_3	R_4
Backup copies		$R_{1,1}$	$R_{1,2}$	$R_{1,3}$
	$R_{2,1}$		$R_{2,2}$	$R_{2,3}$
	$R_{3,1}$	$R_{3,2}$		$R_{3,3}$
	$R_{4,1}$	$R_{4,2}$	$R_{4,3}$	

Fig. 8.9 Example of interleaved partitioning

Node	1	2	3	4
Primary copy	R_1	R_2	R_3	R_4
Backup copy	R_4	R_1	R_2	R_3

Fig. 8.10 Example of chained partitioning

the probability that two adjacent nodes fail is much lower than the probability that any two nodes fail. In failure mode, the load of the failed node and the backup nodes is balanced among all remaining nodes by using both primary and backup copy nodes. In addition, maintaining copy consistency is cheaper. An open issue is how to perform data placement taking into account data replication. Similar to the fragment allocation in distributed databases, this should be considered an optimization problem.

8.4 Parallel Query Processing

The objective of parallel query processing is to transform queries into execution plans that can be efficiently executed in parallel. This is achieved by exploiting parallel data placement and the various forms of parallelism offered by high-level queries. In this section, we first introduce the basic parallel algorithms for data processing. Then, we discuss parallel query optimization.

8.4.1 Parallel Algorithms for Data Processing

Partitioned data placement is the basis for the parallel execution of database queries. Given a partitioned data placement, an important issue is the design of parallel algorithms for efficient processing of database operators (i.e., relational algebra operators) and database queries that combine multiple operators. This issue is difficult because a good trade-off between parallelism and communication cost must be reached since increasing parallelism involves more communication among processors.

Parallel algorithms for relational algebra operators are the building blocks necessary for parallel query processing. The objective of these algorithms is to maximize the degree of parallelism. However, according to Amdahl's law, only part of an algorithm can be parallelized. Let seq be the ratio of the sequential part of a program (a value between 0 and 1), i.e., which cannot be parallelized, and let p be the number of processors. The maximum speed-up that can be achieved is given by the following formula:

$$MaxSpeedup(seq, p) = \frac{1}{seq + \left(\frac{1-seq}{p}\right)}$$

For instance, with $seq = 0$ (the entire program is parallel) and $p = 4$, we obtain the ideal speed-up, i.e., 4. But with $seq = 0.3$, the speed-up goes down to 2.1. And even if we double the number of processors, i.e., $p = 8$, the speed-up increases only slightly to 2.5. Thus, when designing parallel algorithms for data processing, it is important to minimize the sequential part of an algorithm and to maximize the parallel part, by exploiting intraoperator parallelism.

The processing of the select operator in a partitioned data placement context is identical to that in a fragmented distributed database. Depending on the select predicate, the operator may be executed at a single node (in the case of an exact-match predicate) or, in the case of arbitrarily complex predicates, at all the nodes over which the relation is partitioned. If the global index is organized as a B-tree-like structure (see Fig. 8.8), a select operator with a range predicate may be executed only by the nodes that store relevant data. In the rest of this section, we focus on the parallel processing of the two major operators used in database queries, i.e., sort and join.

8.4.1.1 Parallel Sort Algorithms

Sorting relations is necessary for queries that require an ordered result or involve aggregation and grouping. And it is hard to do efficiently as any item needs to be compared with every other item. One of the fastest single processor sort algorithms is *quicksort* but it is highly sequential and thus, according to Amdahl's law, inappropriate for parallel adaptation. Several other centralized sort algorithms can be made parallel. One of the most popular algorithms is the parallel merge sort algorithm, because it is easy to implement and does not have strong requirements on the parallel system architecture. Thus, it has been used in both shared-disk and shared-nothing clusters. It can also be adapted to take advantage of multicore processors.

We briefly review the b-way merge sort algorithm. Let us consider a set of n elements to be sorted. A run is defined as an ordered sequence of elements; thus, the set to be sorted contains n runs of one element. The method consists of iteratively merging b runs of K elements into a sorted run of $K * b$ elements, starting with $K = 1$. For pass i, each set of b runs of b^{i-1} elements is merged into a sorted run of b^i elements. Starting from $i = 1$, the number of passes necessary to sort n elements is $log_b n$.

We now describe the application of this method in a shared-nothing cluster. We assume the popular master–worker model for executing parallel tasks, with one master node coordinating the activities of the worker nodes, by sending them tasks and data and receiving back notifications of tasks done.

Let us suppose we have to sort a relation of p disk pages partitioned over n nodes. Each node has a local memory of $b+1$ pages, where b pages are used as input pages and 1 is used as an output page. The algorithm proceeds in two stages. In the first stage, each node locally sorts its fragment, e.g., using quicksort if the node is single processor or a parallel b-way merge sort if the node is a multicore processor. This stage is called the optimal stage since all nodes are fully busy. It generates n runs of p/n pages, and if n equals b, one node can merge them in a single pass. However, n can be very much greater than b, in which case the solution is for the master node to arrange the worker nodes as a tree of order b during the last stage, called the postoptimal stage. The number of necessary nodes is divided by b at each pass. At the last pass, one node merges the entire relation. The number of passes for the postoptimal stage is $log_b p$. This stage degrades the degree of parallelism.

8.4.1.2 Parallel Join Algorithms

Assuming two arbitrary partitioned relations, there are three basic parallel algorithms to join them: the parallel merge sort join algorithm , the parallel nested loop (PNL) algorithm, and the parallel hash join (PHJ) algorithm. These algorithms are variations of their centralized counterpart. The parallel merge sort join algorithm simply sorts both relations on the join attribute using a parallel merge sort and joins them using a merge like operation done by a single node. Although the last operation is sequential, the result joined relation is sorted on the join attribute, which can be useful for the next operation.

The other two algorithms are fully parallel. We describe them in more details using a pseudoconcurrent programming language with three main constructs: parallel-do, send, and receive. Parallel-do specifies that the following block of actions is executed in parallel. For example,

```
for i from 1 to n in parallel-do action  A
```

indicates that action A is to be executed by n nodes in parallel. The send and receive constructs are basic data communication primitives: send sends data from one node to one or more nodes, while receive gets the content of the data sent at a particular node. In what follows we consider the join of two relations R and S that are partitioned over m and n nodes, respectively. For the sake of simplicity, we assume that the m nodes are distinct from the n nodes. A node at which a fragment of R (respectively, S) resides is called an R-node (respectively, S-node).

Parallel Nested Loop Join Algorithm

The parallel nested loop algorithm is simple and general. It implements the fragment-and-replicate method described in Sect. 4.5.1. It basically composes the

Algorithm 8.1: Parallel Nested Loop (PNL)

```
Input: $R_1, R_2, \ldots, R_m$: fragments of relation R
$S_1, S_2, \ldots, S_n$: fragments of relation S ;
$JP$: join predicate
Output: $T_1, T_2, \ldots, T_n$: result fragments
begin
    for i from 1 to m in parallel do                {send R entirely to each S-node}
        send $R_i$ to each node containing a fragment of S
    end for
    for j from 1 to n in parallel do                {perform the join at each S-node}
        $R \leftarrow \bigcup_{i=1}^{m} R_i$;        {$R_i$ from R-nodes; R is fully replicated at S-nodes}
        $T_j \leftarrow R \bowtie_{JP} S_j$
    end for
end
```

Cartesian product of relationsR and S in parallel. Therefore, arbitrarily complex join predicates, not only equijoin, may be supported.

The algorithm performs two nested loops. One relation is chosen as the inner relation, to be accessed in the inner loop, and the other relation as the outer relation, to be accessed in the outer loop. This choice depends on a cost function with two main parameters: relation sizes, which impacts communication cost, and presence of indexes on join attributes, which impacts local join processing cost.

This algorithm is described in Algorithm 8.1, where the join result is produced at the S-nodes, i.e., S is chosen as inner relation. The algorithm proceeds in two phases.

In the first phase, each fragment of R is sent and replicated at each node that contains a fragment of S (there are n such nodes). This phase is done in parallel by m nodes; thus, $(m * n)$ messages are necessary.

In the second phase, each S-node j receives relation R entirely, and locally joins R with fragment S_j. This phase is done in parallel by n nodes. The local join can be done as in a centralized DBMS. Depending on the local join algorithm, join processing may or may not start as soon as data is received. If a nested loop join algorithm, possibly with an index on the join attribute of S , is used, join processing can be done in a pipelined fashion as soon as a tuple of R arrives. If, on the other hand, a sort-merge join algorithm is used, all the data must have been received before the join of the sorted relations begins.

To summarize, the parallel nested loop algorithm can be viewed as replacing the operator $R \bowtie S$ by $\cup_{i=1}^{n}(R \bowtie S_i)$.

Example 8.2 Figure 8.11 shows the application of the parallel nested loop algorithm with $m = n = 2$. ♦

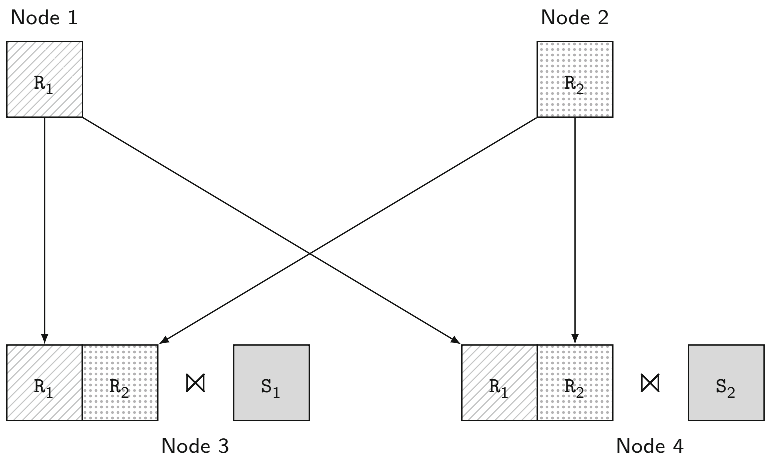


Fig. 8.11 Example of parallel nested loop

Parallel Hash Join Algorithm

The parallel hash join algorithm shown in Algorithm 8.2 applies only in the case of equijoin and does not require any particular partitioning of the operand relations. It has been first proposed for the Grace database machine, and is known as the Grace hash join.

The basic idea is to partition relations R and S into the same number p of mutually exclusive sets (fragments) $R_1, R_2, \ldots, R_p$, and $S_1, S_2, \ldots, S_p$, such that

$$R \bowtie S = \bigcup_{i=1}^{p} (R_i \bowtie S_i)$$

The partitioning of R and S is based on the same hash function applied to the join attribute. Each individual join ($R_i \bowtie S_i$) is done in parallel, and the join result is produced at p nodes. These p nodes may actually be selected at runtime based on the load of the system.

The algorithm can be divided into two main phases, a *build* phase and a *probe* phase. The build phase hashes R used as inner relation, on the join attribute, sends it to the target p nodes that build a hash table for the incoming tuples. The probe phase sends S, the outer relation, associatively to the target p nodes that probe the hash table for each incoming tuple. Thus, as soon as the hash tables have been built for R the S tuples can be sent and processed in pipeline by probing the hash tables.

Example 8.3 Figure 8.12 shows the application of the parallel hash join algorithm with $m = n = 2$. We assume that the result is produced at nodes 1 and 2. Therefore, an arrow from node 1 to node 1 or node 2 to node 2 indicates a local transfer. ♦

Algorithm 8.2: Parallel Hash Join (PHJ)

```
Input: $R_1, R_2, \ldots, R_m$: fragments of relation $R$ ;
$S_1, S_2, \ldots, S_n$: fragments of relation $S$ ;
$JP$: join predicate $R.A = S.B$ ;
$h$: hash function that returns an element of $[1, p]$
Output: $T_1, T_2, \ldots, T_p$: result fragments
begin
    {Build phase}
    for i from 1 to m in parallel do
        $R_i^j \leftarrow$ apply $h(A)$ to $R_i$ $(j = 1, \ldots, p)$;          {hash R on A)}
        send $R_i^j$ to node $j$
    end for
    for j from 1 to p in parallel do
        $R_j \leftarrow \bigcup_{i=1}^{m} R_j^i$          {receive $R_j$ fragments from R-nodes}
        build local hash table for $R_j$
    end for
    {Probe phase}
    for i from 1 to n in parallel do
        $S_i^j \leftarrow$ apply $h(B)$ to $S_i$ $(j = 1, \ldots, p)$;          {hash S on B)}
        send $S_i^j$ to node $j$
    end for
    for j from 1 to p in parallel do
        $S_j \leftarrow \bigcup_{i=1}^{n} S_j^i$;          {receive $S_j$ fragments from S-nodes}
        $T_j \leftarrow R_j \bowtie_{JP} S_j$          {probe $S_j$ for each tuple of $R_j$}
    end for
end
```

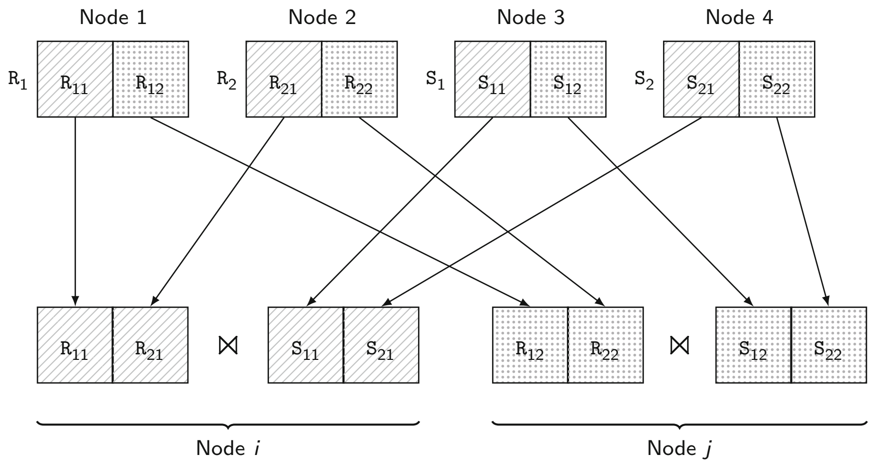


Fig. 8.12 Example of parallel hash join

The parallel hash join algorithm is usually much more efficient than the parallel nested loop join algorithm, since it requires less data transfer and less local join processing in the probe phase. Furthermore, one relation, say R may already be partitioned by hashing on the join attribute. In this case, no build phase is needed and the S fragments are simply sent associatively to corresponding R nodes. It is also generally more efficient than the parallel sort-merge join algorithm. However, this later algorithm is still useful as it produces a result relation sorted on the join attribute.

The problem with the parallel hash join algorithm and its many variants is that the data distribution on the join attribute may be skewed, thus leading to load unbalancing. We discuss solutions to this problem in Sect. 8.5.2.

Variants

The basic parallel join algorithms have been used in many variants, in particular to deal with adaptive query processing or exploit main memories and multicore processors. We discuss these extensions below.

When considering adaptive query processing (see Sect. 4.6), the challenge is to dynamically order pipelined join operators at runtime, while tuples from different relations are flowing in. Ideally, when a tuple of a relation participating in a join arrives, it should be sent to a join operator to be processed on the fly. However, most join algorithms cannot process some incoming tuples on the fly because they are asymmetric with respect to the way inner and outer tuples are processed. Consider PHJ, for instance: the inner relation is fully read during the build phase to construct a hash table, whereas tuples in the outer relation can be pipelined during the probe phase. Thus, an incoming inner tuple cannot be processed on the fly as it must be stored in the hash table and the processing will be possible only when the entire hash table is built. Similarly, the nested loop join algorithm is asymmetric as only the inner relation must be read entirely for each tuple of the outer relation. Join algorithms with some kind of asymmetry offer little opportunity for alternating input relations between inner and outer roles. Thus, to relax the order in which join inputs are consumed, symmetric join algorithms are needed, whereby the role played by the relations in a join may change without producing incorrect results.

The earlier example of symmetric join algorithm is the symmetric hash join, which uses two hash tables, one for each input relation. The traditional build and probe phases of the basic hash join algorithm are simply interleaved. When a tuple arrives, it is used to probe the hash table corresponding to the other relation and find matching tuples. Then, it is inserted in its corresponding hash table so that tuples of the other relation arriving later can be joined. Thus, each arriving tuple can be processed on the fly. Another popular symmetric join algorithm is the ripple join, which is a generalization of the nested loop join algorithm where the roles of inner and outer relation continually alternate during query execution. The main idea is to keep the probing state of each input relation, with a pointer that indicates the last tuple used to probe the other relation. At each toggling point, a change of roles between inner and outer relations occurs. At this point, the new outer relation starts to probe the inner input from its pointer position onwards, to a specified number of tuples. The inner relation, in turn, is scanned from its first tuple to its pointer position minus 1. The number of tuples processed at each stage in the outer relation gives the toggling rate and can be adaptively monitored.

Exploiting processors' main memories is also important for the performance of parallel join algorithms. The hybrid hash join algorithm improves on the Grace hash join by exploiting the available memory to hold an entire partition (called partition 0) during partitioning, thus avoiding disk accesses. Another variation is to modify the built phase so that the resulting hash tables fit into the processor's main memory. This improves performance significantly as the number of cache misses while probing the hash table is reduced. The same idea is used in the radix hash join algorithm for multicore processors, where access to a core's memory is much faster than access to the remote shared-memory. A multipass partitioning scheme is used to divide both input relations into disjoint partitions based on the join attribute, so they fit into the cores' memories. Then, hash tables are built over each partition of the inner relation and probed using the data from the corresponding partition of the outer relation. The parallel merge sort join, which is generally considered inferior to the parallel hash join can also be optimized for multicore processors.

8.4.2 Parallel Query Optimization

Parallel query optimization exhibits similarities with distributed query processing. However, it focuses much more on taking advantage of both intraoperator parallelism (using the algorithms described above) and interoperator parallelism. As any query optimizer, a parallel query optimizer has three components: a search space, a cost model, and a search strategy. In this section, we describe the parallel techniques for these components.

8.4.2.1 Search Space

Execution plans are abstracted by means of operator trees, which define the order in which the operators are executed. Operator trees are enriched with *annotations*, which indicate additional execution aspects, such as the algorithm of each operator. In a parallel DBMS, an important execution aspect to be reflected by annotations is the fact that two subsequent operators can be executed in *pipeline*. In this case, the second operator can start before the first one is completed. In other words, the second operator starts *consuming* tuples as soon as the first one *produces* them. Pipelined executions do not require temporary relations to be materialized, i.e., a tree node corresponding to an operator executed in pipeline is not *stored*.

Some operators and some algorithms require that one operand be stored. For example, in PHJ (Algorithm 8.2), in the build phase, a hash table is constructed in

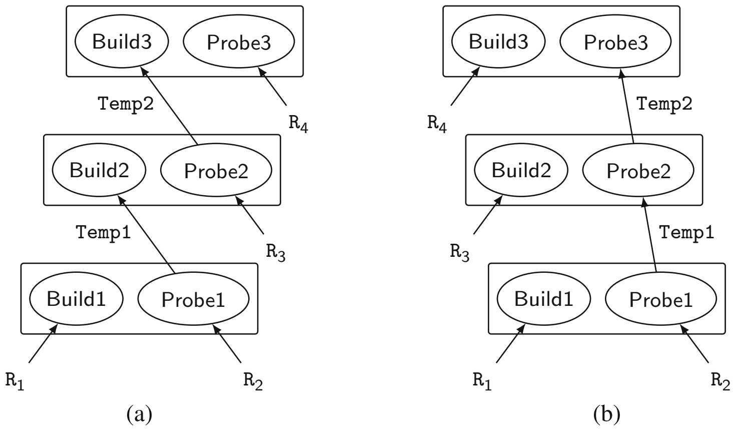


Fig. 8.13 Two hash join trees with a different scheduling. (**a**) No pipeline. (**b**) Pipeline of R_2, Temp1, and Temp2

parallel on the join attribute of the smallest relation. In the probe phase, the largest relation is sequentially scanned and the hash table is consulted for each of its tuples. Therefore, pipeline and stored annotations constrain the *scheduling* of execution plans by splitting an operator tree into nonoverlapping subtrees, corresponding to execution phases. Pipelined operators are executed in the same phase, usually called *pipeline chain*, whereas a storing indication establishes the boundary between one phase and a subsequent phase.

Example 8.4 Figure 8.13 shows two execution trees, one with no pipeline (Fig. 8.13a) and one with pipeline (Fig. 8.13b). In Fig. 8.13a, the temporary relation Temp1 must be completely produced and the hash table in Build2 must be built before Probe2 can start consuming R_3. The same is true for Temp2, Build3, and Probe3. Thus, the tree is executed in four consecutive phases: (1) build R_1's hash table, (2) probe it with R_2 and build Temp1's hash table, (3) probe it with R_3 and build Temp2's hash table, (4) probe it with R_3 and produce the result. Figure 8.13b shows a pipeline execution. The tree can be executed in two phases if enough memory is available to build the hash tables: (1) build the tables for R_1 R_3 and R_4, (2) execute Probe1, Probe2, and Probe3 in pipeline. ♦

The set of nodes where a relation is stored is called its *home*. The *home of an operator* is the set of nodes where it is executed and it must be the home of its operands in order for the operator to access its operand. For binary operators such as join, this might imply repartitioning one of the operands. The optimizer might even sometimes find that repartitioning both the operands is of interest. Operator trees bear execution annotations to indicate repartitioning.

Figure 8.14 shows four operator trees that represent execution plans for a three-way join. Operator trees may be *linear*, i.e., at least one operand of each join node is a base relation or *bushy*. It is convenient to represent pipelined relations as the

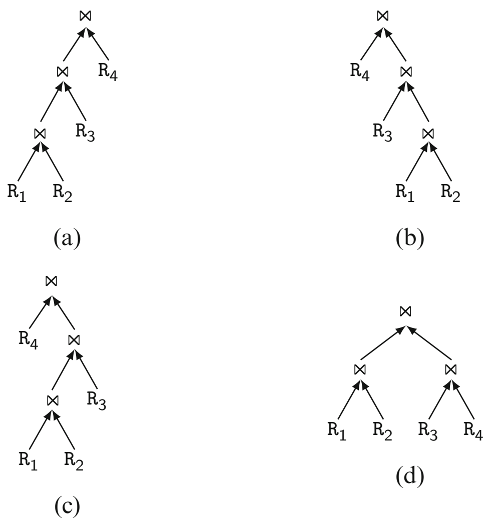


Fig. 8.14 Execution plans as operator trees. (**a**) Left deep. (**b**) Right deep. (**c**) Zigzag. (**d**) Bushy

right-hand side input of an operator. Thus, right-deep trees express full pipelining, while left-deep trees express full materialization of all intermediate results. Thus, assuming enough memory to hold the left-hand side relations, long right-deep trees are more efficient then corresponding left-deep trees. In a left-deep tree such as that of Fig. 8.14a, only the last operator can consume its right input relation in pipeline provided that the left input relation can be entirely stored in main memory.

Parallel tree formats other than left or right deep are also interesting. For example, bushy trees (Fig. 8.14d) are the only ones to allow independent parallelism and some pipeline parallelism. Independent parallelism is useful when the relations are partitioned on disjoint homes. Suppose that the relations in Fig. 8.14d are partitioned such that R_1 and R_2 have the same home h_1 and R_3 and R_4 have the same home h_2 that is different than h_1. Then, the two joins of the base relations could be independently executed in parallel by the set of nodes that constitutes h_1 and h_2.

When pipeline parallelism is beneficial, *zigzag trees*, which are intermediate formats between left-deep and right-deep trees, can sometimes outperform right-deep trees due to a better use of main memory. A reasonable heuristic is to favor right-deep or zigzag trees when relations are partially fragmented on disjoint homes and intermediate relations are rather large. In this case, bushy trees will usually need more phases and take longer to execute. On the contrary, when intermediate relations are small, pipelining is not very efficient because it is difficult to balance the load between the pipeline stages.

With the operator trees above, operators must capture parallelism, which requires repartitioning input relations. This is exemplified in the PHJ algorithm (see Sect. 8.4.1.2), where input relations are partitioned based on the same hash function applied to the join attribute, followed by a parallel join on local partitions.

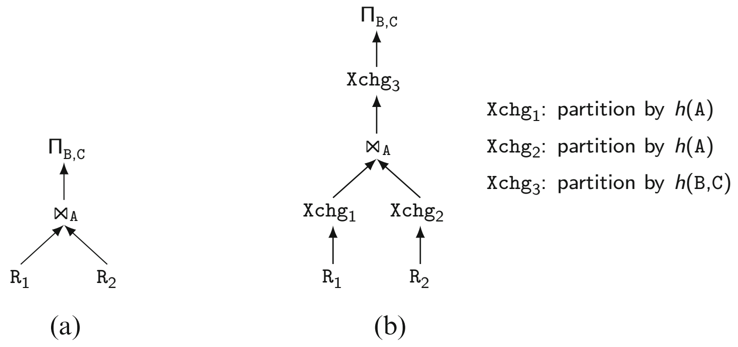


Fig. 8.15 Operator tree with exchange operators. (**a**) Sequential operator tree. (**b**) Parallel operator tree

To ease navigation in the search space by the optimizer, data repartitioning can be encapsulated in an *exchange operator*. Depending on how partitioning is done, we can have different exchange operators such as hashed partitioning, range partitioning, or replicating data to a number of nodes. Examples of uses of exchange operators are:

- Parallel hash join: hashed partitioning of the input relations on join attribute followed by local join;
- Parallel nested loop join: replicating the inner relation on the nodes where the outer relation is partitioned, followed by local join;
- Parallel range sort: range partitioning followed by local sort.

Figure 8.15 shows an example of operator tree with exchange operators. The join operation is done by hashed partitioning of the input relations on A (operators $Xchg_1$ and $Xchg_2$) followed by local join. The project operations are done by duplicate elimination by hashing (operator $Xchg_3$), followed by local project.

8.4.2.2 Cost Model

Recall that the optimizer cost model is responsible for estimating the cost of a given execution plan. It consists of two parts: architecture-dependent and architecture-independent. The architecture-independent part is constituted by the cost functions for operator algorithms, e.g., nested loop for join and sequential access for select. If we ignore concurrency issues, only the cost functions for data repartitioning and memory consumption differ and constitute the architecture-dependent part. Indeed, repartitioning a relation's tuples in a shared-nothing system implies transfers of data across the interconnect, whereas it reduces to hashing in shared-memory systems. Memory consumption in the shared-nothing case is complicated by interoperator parallelism. In shared-memory systems, all operators read and write data through a global memory, and it is easy to test whether there is enough space to execute them in parallel, i.e., the sum of the memory consumption of individual operators is less than the available memory. In shared-nothing, each processor has its own memory, and it becomes important to know which operators are executed in parallel on the same processor. Thus, for simplicity, we can assume that the set of processors (home) assigned to operators do not overlap, i.e., either the intersection of the set of processors is empty or the sets are identical.

The total time of a plan can be computed by a formula that simply adds all CPU, I/O, and communication cost components as in distributed query optimization. The response time is more involved as it must take pipelining into account.

The response time of plan p, scheduled in phases (each denoted by ph), is computed as follows:

$$\begin{aligned} RT(p) = & \sum_{ph \in p} (max_{Op \in ph}(respTime(Op) + pipe\_delay(Op)) \\ & + store\_delay(ph)) \end{aligned}$$

where Op denotes an operator, $respTime(Op)$ is the response time of Op, $pipe\_delay(Op)$ is the waiting period of Op necessary for the producer to deliver the first result tuples (it is equal to 0 if the input relations of Op are stored), $store\_delay(ph)$ is the time necessary to store the output result of phase ph (it is equal to 0 if ph is the last phase, assuming that the results are delivered as soon as they are produced).

To estimate the cost of an execution plan, the cost model uses database statistics and organization information, such as relation cardinalities and partitioning, as with distributed query optimization.

8.4.2.3 Search Strategy

The search strategy does not need to be different from either centralized or distributed query optimization. However, the search space tends to be much larger because there are more parameters that impact parallel execution plans, in particular, pipeline and store annotations. Thus, randomized search strategies such as Iterative Improvement and Simulated Annealing generally outperform traditional deterministic search strategies in parallel query optimization. Another interesting, yet simple approach to reduce the search space is the two phase optimization strategy proposed for XPRS, a shared-memory parallel DBMS. First, at compile time, the optimal query plan based on a centralized cost model is produced. Then, at execution time, runtime parameters such as available buffer size and number of free processors are considered to parallelize the query plan. This approach is shown to almost always produce optimal plans.

8.5 Load Balancing

Good load balancing is crucial for the performance of a parallel system. The response time of a set of parallel operators is that of the longest one. Thus, minimizing the time of the longest one is important for minimizing response time. Balancing the load of different nodes is also essential to maximize throughput. Although the parallel query optimizer incorporates decisions on how to execute a parallel execution plan, load balancing can be hurt by several problems incurring at execution time. Solutions to these problems can be obtained at the intra and interoperator levels. In this section, we discuss these parallel execution problems and their solutions.

8.5.1 Parallel Execution Problems

The principal problems introduced by parallel query execution are initialization, interference, and skew.

Initialization

Before the execution takes place, an initialization step is necessary. This step is generally sequential and includes task (or thread) creation and initialization, communication initialization, etc. The duration of this step is proportional to the degree of parallelism and can actually dominate the execution time of simple queries, e.g., a select query on a single relation. Thus, the degree of parallelism should be fixed according to query complexity.

A formula can be developed to estimate the maximal speed-up reachable during the execution of an operator and to deduce the optimal number of processors. Let us consider the execution of an operator that processes N tuples with n processors. Let c be the average processing time of each tuple and a the initialization time per processor. In the ideal case, the response time of the operator execution is

$$ResponseTime = (a * n) + \frac{c * N}{n}$$

By derivation, we can obtain the optimal number of processors n_{opt} to allocate and the maximal achievable speed-up ($Speed_{max}$).

$$n_{opt} = \sqrt{\frac{c * N}{a}} \qquad Speed_{max} = \frac{n_{opt}}{2}$$

The optimal number of processors (n_{opt}) is independent of n and only depends on the total processing time and initialization time. Thus, maximizing the degree of parallelism for an operator, e.g., using all available processors, can hurt speed-up because of the overhead of initialization.

Interference

A highly parallel execution can be slowed down by *interference*. Interference occurs when several processors simultaneously access the same resource, hardware, or software. A typical example of hardware interference is the contention created on the interconnect of a shared-memory system. When the number of processors is increased, the number of conflicts on the interconnect increases, thus limiting the extensibility of shared-memory systems. A solution to these interferences is to duplicate shared resources. For instance, disk access interference can be eliminated by adding several disks and partitioning the relations.

Software interference occurs when several processors want to access shared data. To prevent incoherence, mutual exclusion variables are used to protect shared data, thus blocking all but one processor that accesses the shared data. This is similar to the locking-based concurrency control algorithms (see Chap. 5). However, shared variables may well become the bottleneck of query execution, creating hot spots. A typical example of software interference is the access of database internal structures such as indexes and buffers. For simplicity, the earlier versions of database systems were protected by a unique mutual exclusion variable, which incurred much overhead.

A general solution to software interference is to partition the shared resource into several independent resources, each protected by a different mutual exclusion variable. Thus, two independent resources can be accessed in parallel, which reduces the probability of interference. To further reduce interference on an independent resource (e.g., an index structure), replication can be used. Thus, access to replicated resources can also be parallelized.

Skew

Load balancing problems can arise with intraoperator parallelism (variation in partition size), namely *data skew*, and interoperator parallelism (variation in the complexity of operators).

The effects of skewed data distribution on a parallel execution can be classified as follows: *Attribute value skew (AVS)* is skew inherent in the data (e.g., there are more citizens in Paris than in Waterloo), while *tuple placement skew (TPS)* is the skew introduced when the data is initially partitioned (e.g., with range partitioning). *Selectivity skew (SS)* is introduced when there is variation in the selectivity of select predicates on each node. *Redistribution skew (RS)* occurs in the redistribution step between two operators. It is similar to TPS. Finally *join product skew (JPS)* occurs because the join selectivity may vary between nodes. Figure 8.16 illustrates this classification on a query over two relations R and S that are poorly partitioned.

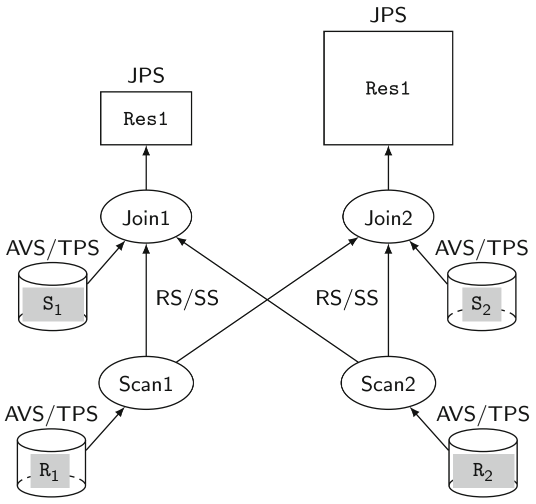


Fig. 8.16 Data skew example

The boxes are proportional to the size of the corresponding partitions. Such poor partitioning stems from either the data (AVS) or the partitioning function (TPS). Thus, the processing times of the two instances Scan1 and Scan2 are not equal. The case of the join operator is worse. First, the number of tuples received is different from one instance to another because of poor redistribution of the partitions of R (RS) or variable selectivity according to the partition of R processed (SS). Finally, the uneven size of S partitions (AVS/TPS) yields different processing times for tuples sent by the scan operator and the result size is different from one partition to the other due to join selectivity (JPS).

8.5.2 Intraoperator Load Balancing

Good intraoperator load balancing depends on the degree of parallelism and the allocation of processors for the operator. For some algorithms, e.g., PHJ, these parameters are not constrained by the placement of the data. Thus, the home of the operator (the set of processors where it is executed) must be carefully decided. The skew problem makes it hard for the parallel query optimizer to make this decision statically (at compile time) as it would require a very accurate and detailed cost model. Therefore, the main solutions rely on adaptive or specialized techniques that can be incorporated in a hybrid query optimizer. We describe below these techniques in the context of parallel join processing, which has received much attention. For simplicity, we assume that each operator is given a home as decided by the query processor (either statically or just before execution).

Adaptive Techniques

The main idea is to statically decide on an initial allocation of the processors to the operator (using a cost model) and, at execution time, adapt to skew using load reallocation. A simple approach to load reallocation is to detect the oversized partitions and partition them again onto several processors (among the processors already allocated to the operation) to increase parallelism. This approach is generalized to allow for more dynamic adjustment of the degree of parallelism. It uses specific *control operators* in the execution plan to detect whether the static estimates for intermediate result sizes will differ from the runtime values. During execution, if the difference between the estimate and the real value is sufficiently high, the control operator performs relation redistribution in order to prevent join product skew and redistribution skew. Adaptive techniques are useful to improve intraoperator load balancing in all kinds of parallel architectures. However, most of the work has been done in the context of shared-nothing where the effects of load unbalance are more severe on performance. DBS3 has pioneered the use of an adaptive technique based on relation partitioning (as in shared-nothing) for shared-memory. By reducing processor interference, this technique yields excellent load balancing for intraoperator parallelism.

Specialized Techniques

Parallel join algorithms can be specialized to deal with skew. One approach is to use multiple join algorithms, each specialized for a different degree of skew, and to determine, at execution time, which algorithm is best. It relies on two main techniques: range partitioning and sampling. Range partitioning is used instead of hash partitioning (in the parallel hash join algorithm) to avoid redistribution skew of the building relation. Thus, processors can get partitions of equal numbers of tuples, corresponding to different ranges of join attribute values. To determine the values that delineate the range values, sampling of the building relation is used to produce a histogram of the join attribute values, i.e., the numbers of tuples for each attribute value. Sampling is also useful to determine which algorithm to use and which relation to use for building or probing. Using these techniques, the parallel hash join algorithm can be adapted to deal with skew as follows:

1. Sample the building relation to determine the partitioning ranges.
2. Redistribute the building relation to the processors using the ranges. Each processor builds a hash table containing the incoming tuples.
3. Redistribute the probing relation using the same ranges to the processors. For each tuple received, each processor probes the hash table to perform the join.

This algorithm can be further improved to deal with high skew using additional techniques and different processor allocation strategies. A similar approach is to modify the join algorithms by inserting a scheduling step that is in charge of redistributing the load at runtime.

8.5.3 Interoperator Load Balancing

In order to obtain good load balancing at the interoperator level, it is necessary to choose, for each operator, how many and which processors to assign for its execution. This should be done taking into account pipeline parallelism, which requires interoperator communication. This is harder to achieve in shared-nothing for the following reasons: First, the degree of parallelism and the allocation of processors to operators, when decided in the parallel optimization phase, are based on a possibly inaccurate cost model. Second, the choice of the degree of parallelism is subject to errors because both processors and operators are discrete entities. Finally, the processors associated with the latest operators in a pipeline chain may remain idle a significant time. This is called the pipeline delay problem.

The main approach in shared-nothing is to determine dynamically (just before the execution) the degree of parallelism and the localization of the processors for each operator. For instance, the rate match algorithm uses a cost model in order to match the rate at which tuples are produced and consumed. It is the basis for choosing the set of processors that will be used for query execution (based on available memory, CPU, and disk utilization). Many other algorithms are possible for the choice of the number and localization of processors, for instance, by maximizing the use of several resources, using statistics on their usage.

In shared-disk and shared-memory, there is more flexibility since all processors have equal access to the disks. Since there is no need for physical relation partitioning, any processor can be allocated to any operator. In particular, a processor can be allocated all the operators in the same pipeline chain, thus, with no interoperator parallelism. However, interoperator parallelism is useful for executing independent pipeline chains. The approach proposed in XPRS for shared-memory allows the parallel execution of independent pipeline chains, called tasks. The main idea is to combine I/O-bound and CPU-bound tasks to increase system resource utilization. Before execution, a task is classified as I/O-bound or CPU-bound using cost model information as follows. Let us suppose that, if executed sequentially, task t generates disk accesses at rate $IO_{rate}(t)$, e.g., in numbers of disk accesses per second. Let us consider a shared-memory system with n processors and a total disk bandwidth of B (numbers of disk accesses per second). Task t is defined as I/O-bound if $IO_{rate}(t) > B/n$ and CPU-bound otherwise. CPU-bound and I/O-bound talks can then be run in parallel at their optimal I/O-CPU balance point. This is accomplished by dynamically adjusting the degree of intraoperator parallelism of the tasks in order to reach maximum resource utilization.

8.5.4 Intraquery Load Balancing

Intraquery load balancing must combine intra and interoperator parallelism. To some extent, given a parallel architecture, the techniques for either intra or interoperator load balancing we just presented can be combined. However, in shared-nothing clusters with shared-memory nodes (or multicore processors), the

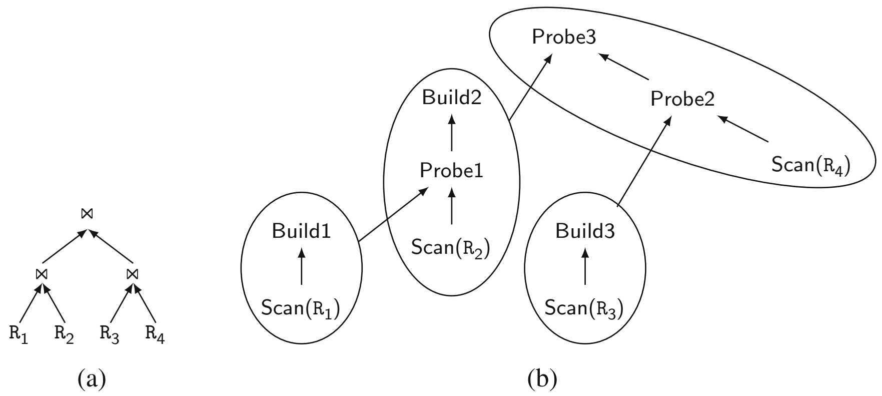


Fig. 8.17 A join tree and associated operator tree. (**a**) Join tree. (**b**) Operator tree (ellipses are pipeline chains)

problems of load balancing are exacerbated because they must be addressed at two levels, locally among the processors or cores of each shared-memory node (SM-node) and globally among all nodes. None of the approaches for intra and interoperator load balancing just discussed can be easily extended to deal with this problem. Load balancing strategies for shared-nothing would experience even more severe problems worsening (e.g., complexity and inaccuracy of the cost model). On the other hand, adapting dynamic solutions developed for shared-memory systems would incur high communication overhead.

A general solution to load balancing is the execution model called *Dynamic Processing (DP)*. The fundamental idea is that the query is decomposed into self-contained units of sequential processing, each of which can be carried out by any processor. Intuitively, a processor can migrate horizontally (intraoperator parallelism) and vertically (interoperator parallelism) along the query operators. This minimizes the communication overhead of internode load balancing by maximizing intra and interoperator load balancing within shared-memory nodes. The input to the execution model is a parallel execution plan as produced by the optimizer, i.e., an operator tree with operator scheduling and allocation of computing resources to operators. The operator scheduling constraints express a partial order among the operators of the query: $Op_1 \prec Op_2$ indicates that operator Op_1 cannot start before operator Op_2.

Example 8.5 Figure 8.17 shows a join tree with four relations R_1, R_2, R_3, and R_4, and the corresponding operator tree with the pipeline chains clearly identified. Assuming that parallel hash join is used, the operator scheduling constraints are between the associated build and probe operators:

Build1 $\prec$ Probe1
Build2 $\prec$ Probe3
Build3 $\prec$ Probe2

There are also scheduling heuristics between operators of different pipeline chains that follow from the scheduling constraints :

Heuristic1: Build1 $\prec$ Scan(R_2) Build3 $\prec$ Scan(R_4), Build2 $\prec$ Scan(R_3)
Heuristic2: Build2 $\prec$ Scan(R_3)

Assuming three SM-nodes i, j, and k with R_1 stored at node i, R_2 and R_3 at node j, and R_4 at node k, we can have the following operator homes:

home (Scan(R_1)) = i
home (Build1, Probe1, Scan(R_2), Scan(R_3)) = j
home (Scan(R_4)) = k
home (Build2, Build3, Probe2, Probe3) = j and k

♦

Given such an operator tree, the problem is to produce an execution that minimizes response time. This can be done by using a dynamic load balancing mechanism at two levels: (i) within an SM-node, load balancing is achieved via fast interprocess communication; (ii) between SM-nodes, more expensive message-passing communication is needed. Thus, the problem is to come up with an execution model so that the use of local load balancing is maximized, while the use of global load balancing (through message passing) is minimized.

We call *activation* the smallest unit of sequential processing that cannot be further partitioned. The main property of the DP model is to allow any processor to process any activation of its SM-node. Thus, there is no static association between threads and operators. This yields good load balancing for both intraoperator and interoperator parallelism within an SM-node, and thus reduces to the minimum the need for global load balancing, i.e., when there is no more work to do in an SM-node.

The DP execution model is based on a few concepts: activations, activation queues, and threads.

Activations

An activation represents a sequential unit of work. Since any activation can be executed by any thread (by any processor), activations must be self-contained and reference all information necessary for their execution: the code to execute and the data to process. Two kinds of activations can be distinguished: trigger activations and data activations. A *trigger activation* is used to start the execution of a leaf operator, i.e., scan. It is represented by an $(Operator, Partition)$ pair that references the scan operator and the base relation partition to scan. A *data activation* describes a tuple produced in pipeline mode. It is represented by an $(Operator, Tuple, Partition)$ triple that references the operator to process. For a build operator, the data activation specifies that the tuple must be inserted in the hash table of the bucket and for a probe operator, that the tuple must be probed with the partition's hash table. Although activations are self-contained, they can only be executed on the SM-node where the associated data (hash tables or base relations) are.

Activation Queues

Moving data activations along pipeline chains is done using *activation queues* associated with operators. If the producer and consumer of an activation are on the same SM-node, then the move is done via shared-memory. Otherwise, it requires message passing. To unify the execution model, queues are used for trigger activations (inputs for scan operators) as well as tuple activations (inputs for build or probe operators). All threads have unrestricted access to all queues located on their SM-node. Managing a small number of queues (e.g., one for each operator) may yield interference. To reduce interference, one queue is associated with each thread working on an operator. Note that a higher number of queues would likely trade interference for queue management overhead. To further reduce interference without increasing the number of queues, each thread is given priority access to a distinct set of queues, called its primary queues. Thus, a thread always tries to first consume activations in its *primary queues*. During execution, operator scheduling constraints may imply that an operator is to be blocked until the end of some other operators (the blocking operators). Therefore, a queue for a blocked operator is also blocked, i.e., its activations cannot be consumed but they can still be produced if the producing operator is not blocked. When all its blocking operators terminate, the blocked queue becomes consumable, i.e., threads can consume its activations. This is illustrated in Fig. 8.18 with an execution snapshot for the operator tree of Fig. 8.17.

Threads

A simple strategy for obtaining good load balancing inside an SM-node is to allocate a number of threads that is much higher than the number of processors and let the operating system do thread scheduling. However, this strategy incurs high numbers of system calls due to thread scheduling and interference. Instead of relying on the operating system for load balancing, it is possible to allocate only one thread per processor per query. This is made possible by the fact that any thread can execute any operator assigned to its SM-node. The advantage of this one thread per processor allocation strategy is to significantly reduce the overhead of interference and synchronization, provided that a thread is never blocked.

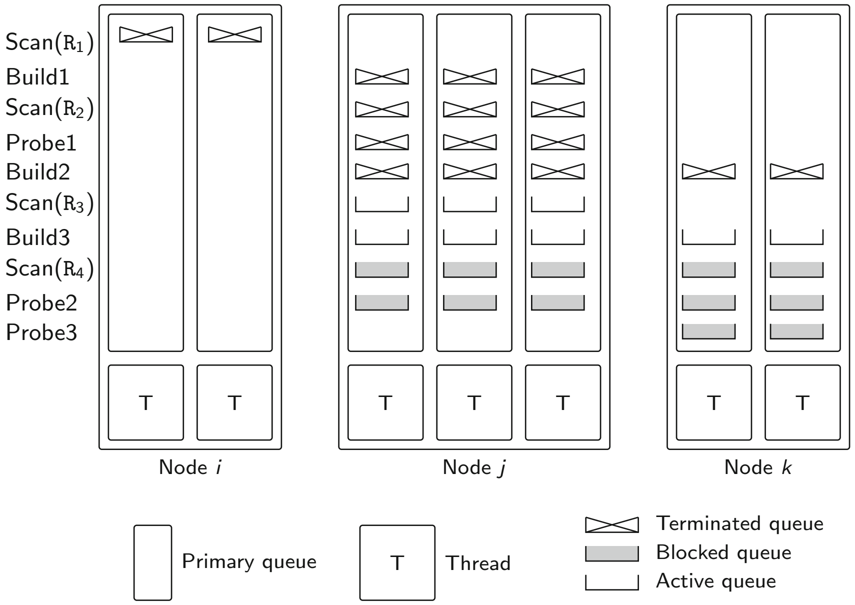


Fig. 8.18 Snapshot of an execution

Load balancing within an SM-node is obtained by allocating all activation queues in a segment of shared-memory and by allowing all threads to consume activations in any queue. To limit thread interference, a thread will consume as much as possible from its set of primary queues before considering the other queues of the SM-node. Therefore, a thread becomes idle only when there is no more activation of any operator, which means that there is no more work to do on its SM-node that is starving.

When an SM-node starves, we share the load of another SM-node by acquiring some of its workload. However, acquiring activations (through message passing) incurs communication overhead. Furthermore, activation acquisition is not sufficient since associated data, i.e., hash tables, must also be acquired. Thus, the benefit of acquiring activations and data should be dynamically estimated.

The amount of load balancing depends on the number of operators that are concurrently executed, which provides opportunities for finding some work to share in case of idle times. Increasing the number of concurrent operators can be done by allowing concurrent execution of several pipeline chains or by using nonblocking hash join algorithms, which allows the concurrent execution of all the operators of the bushy tree. On the other hand, executing more operators concurrently can increase memory consumption. Static operator scheduling as provided by the optimizer should avoid memory overflow and solve this trade-off.

8.6 Fault-Tolerance

In this section, we discuss what happens in the advent of failures. There are several issues raised by failures. The first is how to maintain consistency despite failures. Second, for outstanding transactions, there is the issue of how to perform failover. Third, when a failed replica is reintroduced (following recovery), or a fresh replica is introduced in the system, the current state of the database needs to be recovered. The main concern is how to cope with failures. To start with, failures need to be detected. In group communication based approaches (see Chap. 6), failure detection is provided by the underlying group communication (typically based on some kind of heartbeat mechanism). Membership changes are notified as events.[1] By comparing the new membership with the previous one, it becomes possible to learn which replicas have failed. Group communication also guarantees that all the connected replicas share the same membership notion. For approaches that are not based on group communication failure detection can be either delegated to the underlying communication layer (e.g., TCP/IP) or implemented as an additional component of the replication logic. However, some agreement protocol is needed to ensure that all connected replicas share the same membership notion of which replicas are operational and which ones are not. Otherwise, inconsistencies can arise.

Failures should also be detected at the client side by the client API. Clients typically connect through TCP/IP and can suspect of failed nodes via broken connections. Upon a replica failure, the client API must discover a new replica, reestablish a new connection to it, and, in the simplest case, retransmit the last outstanding transaction to the just connected replica. Since retransmissions are needed, duplicate transactions might be delivered. This requires a duplicate transaction detection and removal mechanism. In most cases, it is sufficient to have a unique client identifier, and a unique transaction identifier per client. The latter is incremented for each new submitted transaction. Thus, the cluster can track whether a client transaction has already been processed and if so, discard it.

Once a replica failure has been detected, several actions should be taken. These actions are part of the failover process, which must redirect the transactions from a failed node to another replica node, in a way that is as transparent as possible for the clients. Failover highly depends on whether or not the failed replica was a master. If a nonmaster replica fails, no action needs to be taken on the cluster side. Clients with outstanding transactions connect to a new replica node and resubmit the last transactions. However, the interesting question is which consistency definition is provided. Recall from Sect. 6.1 that, in a replicated database, one-copy serializabil-itycan be violated as a result of serializing transactions at different nodes in reverse order. Due to failover, the transactions may also be processed in such a way that one-copy serializability is compromised.

[1]Group communication literature uses the term *view change* to denote the event of a membership change. Here, we will not use the term to avoid confusion with the database *view* concept.

In most replication approaches, failover is handled by aborting all ongoing transactions to prevent these situations. However, this way of handling failures has an impact on clients that must resubmit the aborted transactions. Since clients typically do not have transactional capabilities to undo the results of a conversational interaction, this can be very complex. The concept of *highly available transactions* makes failures totally transparent to clients so they do not observe transaction aborts due to failures.

The actions to be taken in the case of a master replica failure are more involved as a new master should be appointed to take over the failed master. The appointment of a new master should be agreed upon by all the replicas in the cluster. In group-based replication, thanks to the membership change notification, it is enough to apply a deterministic function over the new membership to assign masters (all nodes receive exactly the same list of up and connected nodes).

Another essential aspect of fault-tolerance is recovery after failure. High availability requires to tolerate failures and continue to provide consistent access to data despite failures. However, failures diminish the degree of redundancy in the system, thereby degrading availability and performance. Hence, it is necessary to reintroduce failed or fresh replicas in the system to maintain or improve availability and performance. The main difficulty is that replicas do have state and a failed replica may have missed updates while it was down. Thus, a recovering failed replica needs to receive the lost updates before being able to start processing new transactions. A solution is to stop transaction processing. Thus, a quiescent state is directly attained that can be transferred by any of the working replicas to the recovering one. Once the recovering replica has received all the missed updates, transaction processing can resume and all replicas can process new transactions.

8.7 Database Clusters

A parallel database system typically implements the parallel data management functions in a tightly coupled fashion, with all homogeneous nodes under the full control of the parallel DBMS. A simpler (yet not as efficient) solution is to use a *database cluster*, which is a cluster of autonomous databases, each managed by an off-the-shelf DBMS. A major difference with a parallel DBMS implemented on a cluster is the use of a “black-box” DBMS at each node. Since the DBMS source code is not necessarily available and cannot be changed to be “cluster-aware,” parallel data management capabilities must be implemented via middleware. This approach has been successfully adopted in the MySQL or PostgreSQL clusters.

Much research has been devoted to take full advantage of the cluster environment (with fast, reliable communication) in order to improve performance and availability by exploiting data replication. The main results of this research are new techniques for replication, load balancing, and query processing. In this section, we present these techniques after introducing a database cluster architecture.

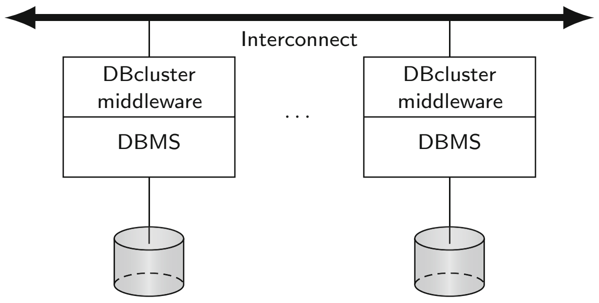


Fig. 8.19 A shared-nothing database cluster

8.7.1 Database Cluster Architecture

Figure 8.19 illustrates a database cluster with a shared-nothing architecture. Parallel data management is done by independent DBMSs orchestrated by a middleware replicated at each node. To improve performance and availability, data can be replicated at different nodes using the local DBMS. Client applications interact with the middleware in a classical way to submit database transactions, i.e., ad hoc queries, transactions, or calls to stored procedures. Some nodes can be specialized as access nodes to receive transactions, in which case they share a global directory service that captures information about users and databases. The general processing of a transaction to a single database is as follows. First, the transaction is authenticated and authorized using the directory. If successful, the transaction is routed to a DBMS at some, possibly different, node to be executed. We will see in Sect. 8.7.4 how this simple model can be extended to deal with parallel query processing, using several nodes to process a single query.

As in a parallel DBMS, the database cluster middleware has several software layers: transaction load balancer, replication manager, query processor, and fault-tolerance manager. The transaction load balancer triggers transaction execution at the best node, using load information obtained from node probes. The "best" node is defined as the one with lightest transaction load. The transaction load balancer also ensures that each transaction execution obeys the ACID properties, and then signals to the DBMS to commit or abort the transaction. The replication manager manages access to replicated data and assures strong consistency in such a way that transactions that update replicated data are executed in the same serial order at each node. The query processor exploits both inter and intraquery parallelism. With interquery parallelism, the query processor routes each submitted query to one node and, after query completion, sends results to the client application. Intraquery parallelism is more involved. As the black-box DBMSs are not cluster-aware, they cannot interact with one another in order to process the same query. Then, it is up to the query processor to control query execution, final result composition, and load balancing. Finally, the fault-tolerance manager provides online recovery and failover.

8.7.2 Replication

As in distributed DBMSs, replication can be used to improve performance and availability. In a database cluster, the fast interconnect and communication system can be exploited to support one-copy serializability while providing scalability (to achieve performance with large numbers of nodes) and autonomy (to exploit black-box DBMS). A cluster provides a stable environment with little evolution of the topology (e.g., as a result of added nodes or communication link failures). Thus, it is easier to support a group communication system that manages reliable communication between groups of nodes. Group communication primitives (see Sect. 6.4) can be used with either eager or lazy replication techniques as a means to attain atomic information dissemination (i.e., instead of the expensive 2PC).

We present now another protocol, called *preventive replication*, which is lazy and provides support for one-copy serializability and scalability. Preventive replication also preserves DBMS autonomy. Instead of using total ordered multicast, it uses FIFO reliable multicast that is simpler and more efficient. The principle is the following. Each incoming transaction T to the system has a chronological timestamp $ts(T) = C$, and is multicast to all other nodes where there is a copy. At each node, a time delay is introduced before starting the execution of T. This delay corresponds to the upper bound of the time needed to multicast a message (a synchronous system with bounded computation and transmission time is assumed). The critical issue is the accurate computation of the upper bounds for messages (i.e., delay). In a cluster system, the upper bound can be computed quite accurately. When the delay expires, all transactions that may have committed before C are guaranteed to be received and executed before T, following the timestamp order (i.e., total order). Hence, this approach prevents conflicts and enforces strong consistency in database clusters. Introducing delay times has also been exploited in several lazy centralized replication protocols for distributed systems. The validation of the preventive replication protocol using experiments with the TPC-C benchmark over a cluster of 64 nodes running the PostgreSQL DBMS have shown excellent scale-up and speed-up.

8.7.3 Load Balancing

In a database cluster, replication offers good load balancing opportunities. With eager or preventive replication (see Sect. 8.7.2), query load balancing is easy to achieve. Since all copies are mutually consistent, any node that stores a copy of the transaction data, e.g., the least loaded node, can be chosen at runtime by a conventional load balancing strategy. Transaction load balancing is also easy in the case of lazy distributed replication since all master nodes need to eventually perform the transaction. However, the total cost of transaction execution at all nodes may be high. By relaxing consistency, lazy replication can better reduce transaction execution cost and thus increase performance of both queries and transactions. Thus, depending on the consistency/performance requirements, eager and lazy replication are both useful in database clusters.

8.7.4 Query Processing

In a database cluster, parallel query processing can be used successfully to yield high performance. Interquery parallelism is naturally obtained as a result of load balancing and replication as discussed in the previous section. Such parallelism is primarily useful to increase the throughput of transaction-oriented applications and, to some extent, to reduce the response time of transactions and queries. For OLAP applications that typically use ad hoc queries, which access large quantities of data, intraquery parallelism is essential to further reduce response time. Intraquery parallelism consists of processing the same query on different partitions of the relations involved in the query.

There are two alternative solutions for partitioning relations in a database cluster: physical and virtual. Physical partitioning defines relation partitions, essentially as horizontal fragments, and allocates them to cluster nodes, possibly with replication. This resembles fragmentation and allocation design in distributed databases (see Chap. 2) except that the objective is to increase intraquery parallelism, not locality of reference. Thus, depending on the query and relation sizes, the degree of partitioning should be much finer. Physical partitioning in database clusters for decision-support can use small grain partitions. Under uniform data distribution, this solution is shown to yield good intraquery parallelism and outperform interquery parallelism. However, physical partitioning is static and thus very sensitive to data skew conditions and the variation of query patterns that may require periodic repartitioning.

Virtual partitioning avoids the problems of static physical partitioning using a dynamic approach and full replication (each relation is replicated at each node). In its simplest form, which we call *simple virtual partitioning (SVP)* , virtual partitions are dynamically produced for each query and intraquery parallelism is obtained by sending subqueries to different virtual partitions. To produce the different subqueries, the database cluster query processor adds predicates to the incoming query in order to restrict access to a subset of a relation, i.e., a virtual partition. It may also do some rewriting to decompose the query into equivalent subqueries followed by a composition query. Then, each DBMS that receives a subquery is forced to process a different subset of data items. Finally, the partitioned result needs to be combined by an aggregate query.

Example 8.6 Let us illustrate SVP with the following query *Q*:

```
SELECT PNO, AVG(DUR)
FROM   WORKS
WHERE  SUM(DUR) > 200
GROUP BY PNO
```

A generic subquery on a virtual partition is obtained by adding to *Q*'s where clause the predicate "**and** PNO >= 'P1' and PNO < 'P2'." By binding ['P1', 'P2'] to *n* subsequent ranges of PNO values, we obtain *n* subqueries, each for a different node on a different virtual partition of WORKS. Thus, the degree of intraquery parallelism is *n*. Furthermore, the **AVG**(DUR) operation must be rewritten as **SUM**(DUR), **COUNT**(DUR) in the subquery. Finally, to obtain the correct result for **AVG**(DUR), the composition query must perform **SUM**(DUR)/**SUM**(**COUNT**(DUR)) over the *n* partial results.

The performance of each subquery's execution depends heavily on the access methods available on the partitioning attribute (PNO). In this example, a clustered index on PNO would be best. Thus, it is important for the query processor to know the access methods available to decide, according to the query, which partitioning attribute to use. ♦

SVP allows great flexibility for node allocation during query processing since any node can be chosen for executing a subquery. However, not all kinds of queries can benefit from SVP and be parallelized. We can classify OLAP queries such that queries of the same class have similar parallelization properties. This classification relies on how the largest relations, called fact tables in a typical OLAP application, are accessed. The rationale is that the virtual partitioning of such relations yields higher intraoperator parallelism. Three main classes are identified:

1. Queries without subqueries that access a fact table.
2. Queries with a subquery that are equivalent to a query of Class 1.
3. Any other queries.

Queries of Class 2 need to be rewritten into queries of Class 1 in order for SVP to apply, while queries of Class 3 cannot benefit from SVP.

SVP has some limitations. First, determining the best virtual partitioning attributes and value ranges can be difficult since assuming uniform value distribution is not realistic. Second, some DBMSs perform full table scans instead of indexed access when retrieving tuples from large intervals of values. This reduces the benefits of parallel disk access since one node could read an entire relation to access a virtual partition. This makes SVP dependent on the underlying DBMS query capabilities. Third, as a query cannot be externally modified while being executed, load balancing is difficult to achieve and depends on the initial partitioning.

Fine-grained virtual partitioning addresses these limitations by using a large number of subqueries instead of one per DBMS. Working with smaller subqueries avoids full table scans and makes query processing less vulnerable to DBMS idiosyncrasies. However, this approach must estimate the partition sizes, using database statistics and query processing time estimates. In practice, these estimates are hard to obtain with black-box DBMSs.

Adaptive virtual partitioning (AVP) solves this problem by dynamically tuning partition sizes, thus without requiring these estimates. AVP runs independently at each participating cluster node, avoiding internode communication (for partition size determination). Initially, each node receives an interval of values to work with. These intervals are determined exactly as for SVP. Then, each node performs the following steps:

1. Start with a very small partition size beginning with the first value of the received interval.
2. Execute a subquery with this interval.
3. Increase the partition size and execute the corresponding subquery while the increase in execution time is proportionally smaller than the increase in partition size.
4. Stop increasing. A stable size has been found.
5. If there is performance degradation, i.e., there were consecutive worse executions, decrease size and go to Step 2.

Starting with a very small partition size avoids full table scans at the very beginning of the process. This also avoids having to know the threshold after which the DBMS does not use clustered indices and starts performing full table scans. When partition size increases, query execution time is monitored allowing determination of the point after which the query processing steps that are data size independent do not influence too much total query execution time. For example, if doubling the partition size yields an execution time that is twice the previous one, this means that such a point has been found. Thus the algorithm stops increasing the size. System performance can deteriorate due to DBMS data cache misses or overall system load increase. It may happen that the size being used is too large and has benefited from previous data cache hits. In this case, it may be better to shrink partition size. That is precisely what step 5 does. It gives a chance to go back and inspect smaller partition sizes. On the other hand, if performance deterioration was due to a casual and temporary increase of system load or data cache misses, keeping a small partition size can lead to poor performance. To avoid such a situation, the algorithm goes back to Step 2 and restarts increasing sizes.

AVP and other variants of virtual partitioning have several advantages: flexibility for node allocation, high availability because of full replication, and opportunities for dynamic load balancing. But full replication can lead to high cost in disk usage. To support partial replication, hybrid solutions have been proposed to combine physical and virtual partitioning. The hybrid design uses physical partitioning for the largest and most important relations and fully replicates the small tables. Thus, intraquery parallelism can be achieved with lesser disk space requirements. The hybrid solution combines AVP with physical partitioning. It solves the problem of disk usage while keeping the advantages of AVP, i.e., full table scan avoidance and dynamic load balancing.

8.8 Conclusion

Parallel database systems have been exploiting multiprocessor architectures to provide high-performance, high-availability, extensibility, and scalability with a good cost/performance ratio. Furthermore, parallelism is the only viable solution for supporting very large databases and applications within a single system.

Parallel database system architectures can be classified as shared-memory, shared-disk, and shared-nothing. Each architecture has its advantages and limitations. Shared-memory is used in tightly coupled NUMA multiprocessors or multicore processors, and can provide the highest performance because of fast memory access and great load balancing. However, it has limited extensibility and scalability. Shared-disk and shared-nothing are used in computer clusters, typically using multicore processors. With low latency networks (e.g., Infiniband and Myrinet), they can provide high performance and scale up to very large configurations (with thousands of nodes). Furthermore, the RDMA capability of those networks can be exploited to make cost-effective NUMA clusters. Shared-disk is typically used for OLTP workloads as it is simpler and has good load balancing. However, shared-nothing remains the only choice for highly scalable systems, as need in OLAP or big data, with the best cost/performance ratio.

Parallel data management techniques extend distributed database techniques. However, the critical issues for such architectures are data partitioning, replication, parallel query processing, load balancing, and fault-tolerance. The solutions to these issues are more involved than in distributed DBMS because they must scale to high numbers of nodes. Furthermore, recent advances in hardware/software such as low latency interconnect, multicore processor nodes, large main memory, and RDMA provide new opportunities for optimization. In particular, parallel algorithms for the most demanding operators such as join and sort need be made NUMA-aware.

A database cluster is an important kind of parallel database system that uses a black-box DBMS at each node. Much research has been devoted to take full advantage of the cluster stable environment in order to improve performance and availability by exploiting data replication. The main results of this research are new techniques for replication, load balancing, and query processing.

8.9 Bibliographic Notes

The earlier proposal of a database machine dates back to [Canaday et al. 1974], mainly to address the "I/O bottleneck" [Boral and DeWitt 1983], induced by high disk access time with respect to main memory access time. The main idea was to push database functions closer to disk. CAFS-ISP is an early example of hardware-based filtering device [Babb 1979] that was bundled within disk controllers for fast associative search. The introduction of general-purpose microprocessors in disk controllers also led to intelligent disks [Keeton et al. 1998].

The first parallel database system products were Teradata and Tandem NonStopSQL in the early 1980s. Since then, all major DBMS players have delivered a parallel version of their product. Today, the field is still the subject of intensive research to deal with big data and exploit new hardware capabilities, e.g., low latency interconnects, multicore processor nodes, and large main memories.

Comprehensive surveys of parallel database systems are provided in [DeWitt and Gray 1992, Valduriez 1993, Graefe 1993]. Parallel database system architectures are discussed in [Bergsten et al. 1993, Stonebraker 1986, Pirahesh et al. 1990], and compared using a simple simulation model in [Breitbart and Silberschatz 1988]. The first NUMA architectures are described in [Lenoski et al. 1992, Goodman and Woest 1988]. A more recent approach based on Remote Direct Memory Access (RDMA) is discussed in [Novakovic et al. 2014, Leis et al. 2014, Barthels et al. 2015].

Examples of parallel database system prototypes are Bubba [Boral et al. 1990], DBS3 [Bergsten et al. 1991], Gamma [DeWitt et al. 1986], Grace [Fushimi et al. 1986], Prisma/DB [Apers et al. 1992], Volcano [Graefe 1990], and XPRS [Hong 1992].

Data placement, including replication, in a parallel database system is treated in [Livny et al. 1987, Copeland et al. 1988, Hsiao and DeWitt 1991]. A scalable solution is Gamma's *chained partitioning* [Hsiao and DeWitt 1991], which stores the primary and backup copy on two adjacent nodes. Associative access to a partitioned relation using a global index is proposed in [Khoshafian and Valduriez 1987].

Parallel query optimization is treated in [Shekita et al. 1993], [Ziane et al. 1993], and [Lanzelotte et al. 1994]. Our discussion of cost model in Sect. 8.4.2.2 is based on [Lanzelotte et al. 1994]. Randomized search strategies are proposed in [Swami 1989, Ioannidis and Wong 1987]. XPRS uses a two phase optimization strategy [Hong and Stonebraker 1993]. The exchange operator, which is the basis for parallel repartitioning in parallel query processing, was proposed in the context of the Volcano query evaluation system [Graefe 1990].

There is an extensive literature on parallel algorithms for database operators, in particular sort and join. The objective of these algorithms is to maximize the degree of parallelism, following Amdahl's law [Amdahl 1967] that states that only part of an algorithm can be parallelized. The seminal paper by [Bitton et al. 1983] proposes and compares parallel versions of merge sort, nested loop join, and sort-merge join algorithms. Valduriez and Gardarin [1984] propose the use of hashing for parallel join and semijoin algorithms. A survey of parallel sort algorithms can be found in [Bitton et al. 1984]. The specification of two main phases, *build* and *probe*, [DeWitt and Gerber 1985] has been useful to understand parallel hash join algorithms. The Grace hash join [Kitsuregawa et al. 1983], the hybrid hash join algorithm [DeWitt et al. 1984, Shatdal et al. 1994], and the radix hash join [Manegold et al. 2002] have been the basis for many variations in particular to exploit multicore processors and NUMA [Barthels et al. 2015]. Other important join algorithms are the symmetric hash join [Wilschut and Apers 1991] and the Ripple join [Haas and Hellerstein 1999b]. In [Barthels et al. 2015], the authors show that a radix hash join can perform very well in large-scale shared-nothing clusters using RDMA.

The parallel sort-merge join algorithm is gaining renewed interest in the context of multicore and NUMA systems [Albutiu et al. 2012, Pasetto and Akhriev 2011].

Load balancing in parallel database systems has been extensively studied both in the context of shared-memory and shared-disk [Lu et al. 1991, Shekita et al. 1993] and shared-nothing [Kitsuregawa and Ogawa 1990, Walton et al. 1991, DeWitt et al. 1992, Shatdal and Naughton 1993, Rahm and Marek 1995, Mehta and DeWitt 1995, Garofalakis and Ioannidis 1996]. The presentation of the Dynamic Processing execution model in Sect. 8.5 is based on [Bouganim et al. 1996, 1999]. The rate match algorithm is described in [Mehta and DeWitt 1995].

The effects of skewed data distribution on a parallel execution are introduced in [Walton et al. 1991]. A general adaptive approach to dynamically adjust the degree of parallelism using control operators is proposed in [Biscondi et al. 1996]. A good approach to deal with data skew is to use multiple join algorithms, each specialized for a different degree of skew, and to determine, at execution time, which algorithm is best [DeWitt et al. 1992].

The content of Sect. 8.6 on fault-tolerance is based on [Kemme et al. 2001, Jiménez-Peris et al. 2002, Perez-Sorrosal et al. 2006].

The concept of database cluster is defined in [Röhm et al. 2000, 2001]. Several protocols for scalable eager replication in database clusters using group communication are proposed in [Kemme and Alonso 2000b,a, Patiño-Martínez et al. 2000, Jiménez-Peris et al. 2002]. Their scalability has been studied analytically in [Jiménez-Peris et al. 2003]. Partial replication is studied in [Sousa et al. 2001]. The presentation of preventive replication in Sect. 8.7.2 is based on [Pacitti et al. 2005]. Load balancing in database clusters is addressed in [Milán-Franco et al. 2004, Gançarski et al. 2007].

Most of the content of Sect. 8.7.4 is based on the work on adaptive virtual partitioning [Lima et al. 2004] and hybrid partitioning [Furtado et al. 2008]. Physical partitioning in database clusters for decision-support is addressed by [Stöhr et al. 2000], using small grain partitions. Akal et al. [2002] propose a classification of OLAP queries such that queries of the same class have similar parallelization properties.

Exercises

Problem 8.1 (*) Consider a shared-disk cluster and very big relations that need to be partitioned across several disk units. How you would adapt the various partitioning and replication techniques in Sect. 8.3 to take advantage of shared-disk? Discuss the impact on query performance and fault-tolerance.

Problem 8.2 (**) Order-preserving hashing [Knuth 1973] could be used to partition a relation on an attribute A, so that the tuples in any partition $i+1$ have A values higher than those of the tuples in partition i. Propose a parallel sort algorithm that exploits order-preserving hashing. Discuss it advantages and limitations, compared with the b-way merge sort algorithm in Sect. 8.4.1.1.

Problem 8.3 Consider the parallel hash join algorithm in Sect. 8.4.1.2. Explain what the build phase and probe phase are. Is the algorithm symmetric with respect to its input relations?

Problem 8.4 (*) Consider the join of two relations `R` and `S` in a shared-nothing cluster. Assume that `S` is partitioned by hashing on the join attribute. Modify the parallel hash join algorithm in Sect. 8.4.1.2 to take advantage of this case. Discuss the execution cost of this algorithm.

Problem 8.5 (**) Consider a simple cost model to compare the performance of the three basic parallel join algorithms (nested loop join, sort-merge join, and hash join). It is defined in terms of total communication cost (C_{COM}) and processing cost (C_{PRO}). The total cost of each algorithm is therefore

$$Cost(Alg.) = C_{COM}(Alg.) + C_{PRO}(Alg.)$$

For simplicity, C_{COM} does not include control messages, which are necessary to initiate and terminate local tasks. We denote by $msg(\#tup)$ the cost of transferring a message of $\#tup$ tuples from one node to another. Processing costs (that include total I/O and CPU cost) are based on the function $C_{LOC}(m, n)$ that computes the local processing cost for joining two relations with cardinalities m and n. Assume that the local join algorithm is the same for all three parallel join algorithms. Finally, assume that the amount of work done in parallel is uniformly distributed over all nodes allocated to the operator. Give the formulas for the total cost of each algorithm, assuming that the input relations are arbitrary partitioned. Identify the conditions under which an algorithm should be used.

Problem 8.6 Consider the following SQL query:

```
SELECT ENAME, DUR
FROM   EMP, ASG, PROJ
WHERE  EMP.ENO=ASG.ENO
AND    ASG.PNO=PROJ.PNO
AND    RESP="Manager"
AND    PNAME="Instrumentation"
```

Give four possible operator trees: right-deep, left-deep, zigzag, and bushy. For each one, discuss the opportunities for parallelism.

Problem 8.7 Consider a nine way join (ten relations are to be joined), calculate the number of possible right-deep, left-deep, and bushy trees, assuming that each relation can be joined with anyone else. What do you conclude about parallel optimization?

Problem 8.8 (**) Propose a data placement strategy for a NUMA cluster (using RDMA) that maximizes a combination of *intranode* parallelism (intraoperator parallelism within shared-memory nodes) and *internode* parallelism (interoperator parallelism across shared-memory nodes).

Problem 8.9 (**) How should the DP execution model presented in Sect. 8.5.4 be changed to deal with interquery parallelism?

Problem 8.10 (**) Consider a multiuser centralized database system. Describe the main change to allow interquery parallelism from the database system developer and administrator's points of view. What are the implications for the end-user in terms of interface and performance?

Problem 8.11 (*) Consider the database cluster architecture in Fig. 8.19. Assuming that each cluster node can accept incoming transactions, make precise the database cluster middleware box by describing the different software layers, and their components and relationships in terms of data and control flow. What kind of information need be shared between the cluster nodes? how?

Problem 8.12 (**) Discuss the issues of fault-tolerance for the preventive replication protocol (see Sect. 8.7.2).

Problem 8.13 (**) Compare the preventive replication protocol with the eager replication protocol (see Chap. 6) in the context of a database cluster in terms of: replication configurations supported, network requirements, consistency, performance, fault-tolerance.

Problem 8.14 (**) Consider two relations R(A,B,C,D,E) and S(A,F,G,H). Assume there is a clustered index on attribute A for each relation. Assuming a database cluster with full replication, for each of the following queries, determine whether Virtual Partitioning can be used to obtain intraquery parallelism and, if so, write the corresponding subquery and the final result composition query.

(a)
```
SELECT B, COUNT(C)
FROM   R
GROUP BYB
```

(b)
```
SELECT C, SUM(D), AVG(E)
FROM   R
WHERE  B=:v1
GROUP BY C
```

(c)
```
SELECT B, SUM(E)
FROM.  R, S
WHERE. R.A=S.A
GROUP BY B
HAVING COUNT(*) > 50
```

(d)
```
SELECT B, MAX(D)
FROM+.  R, S
WHERE C = (SELECT SUM(G) FROM S WHERE S.A=R.A)
GROUP BY  B
```

(e)
```
SELECT B, MIN(E)
FROM.  R
WHERE  D > (SELECT MAX(H) FROM S WHERE G >= :v1)
GROUP BY B
```