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 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

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

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

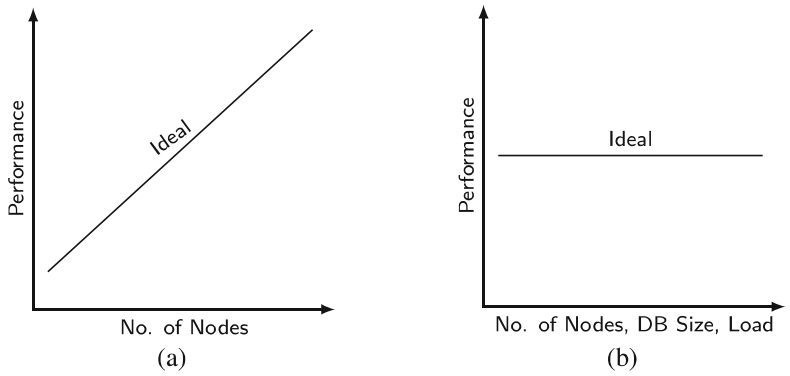

Fig. 8.1 Extensibility metrics. (**a**) Linear speed-up. (**b**) Linear scale-up

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,

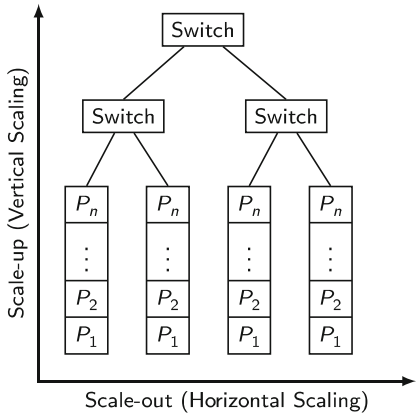

Fig. 8.2 Scale-up versus scale-out

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

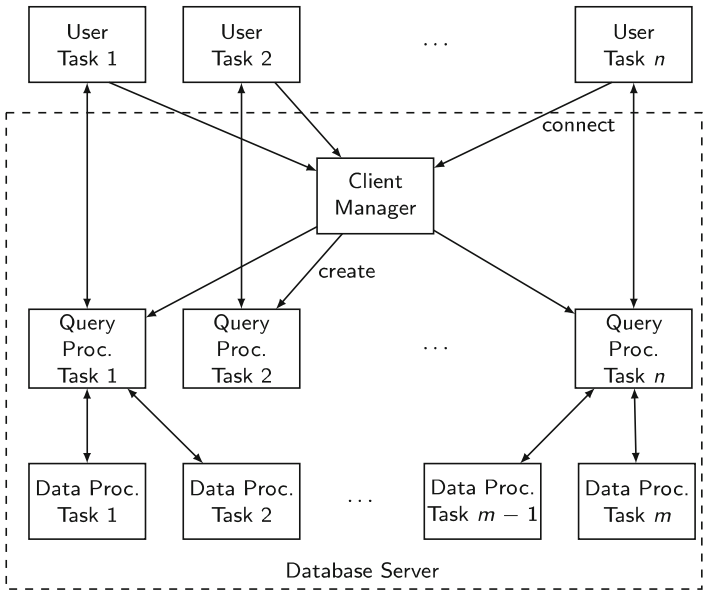

Fig. 8.3 General architecture of a parallel database system

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

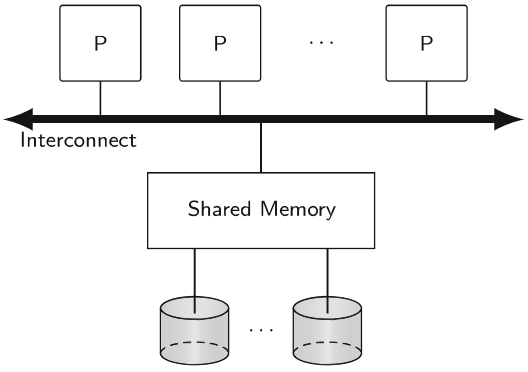

Fig. 8.4 Shared-memory

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.

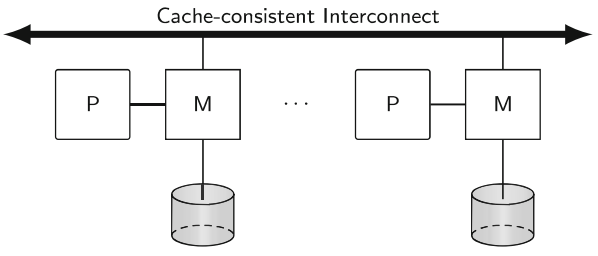

Fig. 8.5 Cache coherent non-uniform memory architecture (CC-NUMA)

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

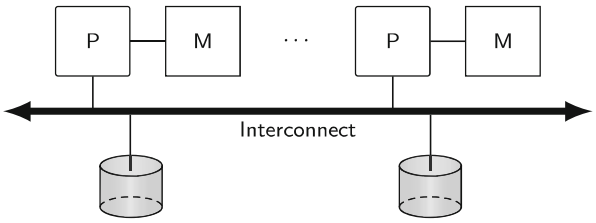

Fig. 8.6 Shared-disk architecture

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

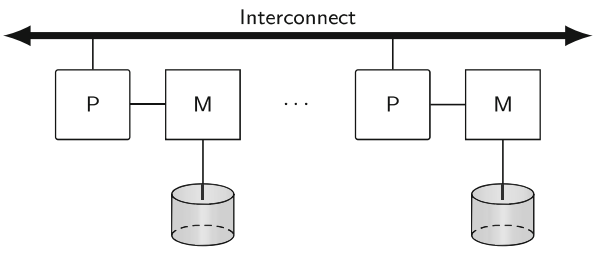

Fig. 8.7 Shared-nothing architecture

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

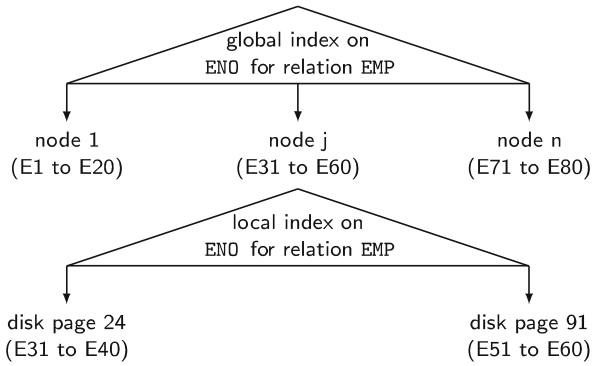

Fig. 8.8 Example of global and local indexes

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.