Note that, since the refresh transaction is sent to site D sometime after T_3 commits at site M, at step 3 when it reads the value of x at site D, it reads the old value and does not see the value of its own $Write$ that just precedes $Read$. ♦

Because of these problems, there are not too many proposals for full transparency in lazy replication algorithms. A notable exception is that by Bernstein et al. [2006] that considers the single master case and provides a method for validity testing by the master site, at commit point, similar to optimistic concurrency control. The fundamental idea is the following. Consider a transaction T that writes a data item x. At commit time of transaction T, the master generates a timestamp for it and uses this timestamp to set a timestamp for the master copy of x (x_M) that records the timestamp of the last transaction that updated it ($last\_modified(x_M)$). This is appended to refresh transactions as well. When refresh transactions are received at slaves they also set their copies to this same value, i.e., $last\_modified(x_i) \leftarrow last\_modified(x_M)$. The timestamp generation for T at the master follows the following rule:

> The timestamp for transaction T should be greater than all previously issued timestamps and should be less than the $last\_modified$ timestamps of the data items it has accessed. If such a timestamp cannot be generated, then T is aborted.[3]

This test ensures that read operations read correct values. For example, in Example 13.4, master site M would not be able to assign an appropriate timestamp to transaction T_1 when it commits, since the $last\_modified(x_M)$ would reflect the update performed by T_2. Therefore, T_1 would be aborted.

Although this algorithm handles the first problem we discussed above, it does not automatically handle the problem of a transaction not seeing its own writes (what we referred to as transaction inversion earlier). To address this issue, it has been suggested that a list be maintained of all the updates that a transaction performs and this list is consulted when a *Read* is executed. However, since only the master knows the updates, the list has to be maintained at the master and all the *Read*s (as well as *Write*s) have to be executed at the master.

13.3.4 Lazy Distributed Protocols

Lazy distributed replication protocols are the most complex ones owing to the fact that updates can occur on any replica and they are propagated to the other replicas lazily (Figure 13.6).

The operation of the protocol at the site where the transaction is submitted is straightforward: both *Read* and *Write* operations are executed on the local copy, and the transaction commits locally. Sometime after the commit, the updates are propagated to the other sites by means of refresh transactions.

[3] The original proposal handles a wide range of freshness constraints, as we discussed earlier; therefore, the rule is specified more generically. However, since our discussion primarily focuses on 1SR behavior, this (more strict) recasting of the rule is appropriate.

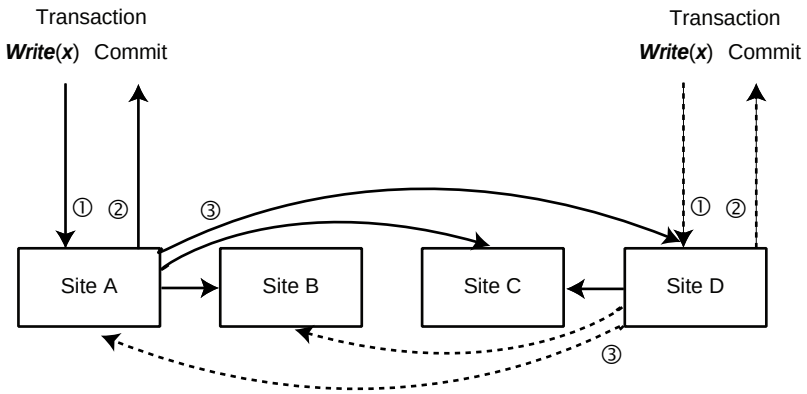

Fig. 13.6 Lazy Distributed Replication Protocol Actions. (1) Two updates are applied on two local replicas; (2) Transaction commit makes the updates permanent; (3) The updates are independently propagated to the other replicas.

The complications arise in processing these updates at the other sites. When the refresh transactions arrive at a site, they need to be locally scheduled, which is done by the local concurrency control mechanism. The proper serialization of these refresh transactions can be achieved using the techniques discussed in previous sections. However, multiple transactions can update different copies of the same data item concurrently at different sites, and these updates may conflict with each other. These changes need to be reconciled, and this complicates the ordering of refresh transactions. Based on the results of reconciliation, the order of execution of the refresh transactions is determined and updates are applied at each site.

The critical issue here is reconciliation. One can design a general purpose reconciliation algorithm based on heuristics. For example, updates can be applied in timestamp order (i.e., those with later timestamps will always win) or one can give preference to updates that originate at certain sites (perhaps there are more important sites). However, these are ad hoc methods and reconciliation is really dependent upon application semantics. Furthermore, whatever reconciliation technique is used, some of the updates are lost. Note that timestamp-based ordering will only work if timestamps are based on local clocks that are synchronized. As we discussed earlier, this is hard to achieve in large-scale distributed systems. Simple timestamp-based approach, which concatenates a site number and local clock, gives arbitrary preference between transactions that may have no real basis in application logic. The reason timestamps work well in concurrency control and not in this case is because in concurrency control we are only interested in determining *some* order; here we are interested in determining a *particular* order that is consistent with application semantics.

13.4 Group Communication

As discussed in the previous section, the overhead of replication protocols can be high – particularly in terms of message overhead. A very simple cost model for the replication algorithms is as follows. If there are n replicas and each transaction consists of m update operations, then each transaction issues $n * m$ messages (if multicast communication is possible, m messages would be sufficient). If the system wishes to maintain a throughput of k transactions-per-second, this results in $k * n * m$ messages per second (or $k * m$ in the case of multicasting). One can add sophistication to this cost function by considering the execution time of each operation (perhaps based on system load) to get a cost function in terms of time. The problem with many of the replication protocols discussed above (in particular the distributed ones) is that their message overhead is high.

A critical issue in efficient implementation of these protocols is to reduce the message overhead. Solutions have been proposed that use group communication protocols [Chockler et al., 2001] together with non-traditional techniques for processing local transactions [Stanoi et al., 1998; Kemme and Alonso, 2000a,b; Patiño-Martínez et al., 2000; Jiménez-Peris et al., 2002]. These solutions introduce two modifications: they do not employ 2PC at commit time, but rely on the underlying group communication protocols to ensure agreement, and they use deferred update propagation rather than synchronous.

Let us first review the group communication idea. A group communication system enables a node to multicast a message to all nodes of a group with a delivery guarantee, i.e., the message is eventually delivered to all nodes. Furthermore, it can provide multicast primitives with different delivery orders only one of which is important for our discussion: total order. In total ordered multicast, all messages sent by different nodes are delivered in the same total order at all nodes. This is important in understanding the following discussion.

We will demonstrate the use of group communication by considering two protocols. The first one is an alternative eager distributed protocol [Kemme and Alonso, 2000a], while the second one is a lazy centralized protocol [Pacitti et al., 1999].

The group communication-based eager distributed protocol due to Kemme and Alonso [2000a] uses a local processing strategy where $Write$ operations are carried out on local shadow copies where the transaction is submitted and utilizes total ordered group communication to multicast the set of write operations of the transaction to all the other replica sites. Total ordered communication guarantees that all sites receive the write operations in exactly the same order, thereby ensuring identical serialization order at every site. For simplicity of exposition, in the following discussion, we assume that the database is fully replicated and that each site implements a 2PL concurrency control algorithm.

The protocol executes a transaction T_i in four steps (local concurrency control actions are not indicated):

I. **Local processing phase.** A $Read_i(x)$ operation is performed at the site where it is submitted (this is the master site for this transaction). A $Write_i(x)$ op-

eration is also performed at the master site, but on a shadow copy (see the previous chapter for a discussion of shadow paging).

II. **Communication phase.** If T_i consists only of *Read* operations, then it can be committed at the master site. If it involves $Write$ operations (i.e., if it is an update transaction), then the TM at T_i's master site (i.e., the site where T_i is submitted) assembles the writes into one *write message* WM_i[4] and multicasts it to all the replica sites (including itself) using total ordered group communication.

III. **Lock phase.** When WM_i is delivered at a site S_j, it requests all locks in WM_i in an atomic step. This can be done by acquiring a latch (lighter form of a lock) on the lock table that is kept until all the locks are granted or requests are enqueued. The following actions are performed:

1. For each $Write(x)$ in WM_i (let x_j refer to the copy of x that exists at site S_j), the following are performed:

(a) If there are no other transactions that have locked x_j, then the write lock on x_j is granted.

(b) Otherwise a conflict test is performed:

- If there is a local transaction T_k that has already locked x_j, but is in its local read or communication phases, then T_k is aborted. Furthermore, if T_k is in its communication phase, a final decision message *Abort* is multicast to all the sites. At this stage, read/write conflicts are detected and local read transactions are simply aborted. Note that only local read operations obtain locks during the local execution phase, since local writes are only executed on shadow copies. Therefore, there is no need to check for write/write conflicts at this stage.
- Otherwise, $W_i(x_j)$ lock request is put on queue for x_j.

2. If T_i is a local transaction (recall that the message is also sent to the site where T_i originates, in which case $j = i$), then the site can commit the transaction, so it multicasts a *Commit* message. Note that the commit message is sent as soon as the locks are requested and not after writes; thus this is not a 2PC execution.

IV. **Write phase.** When a site is able to obtain the write lock, it applies the corresponding update (for the master site, this means that the shadow copy is made the valid version). The site where T_i is submitted can commit and release all the locks. Other sites have to wait for the decision message and terminate accordingly.

[4] What is being sent are the updated data items (i.e., state transfer).

Note that in this protocol, the important thing is to ensure that the lock phases of the concurrent transactions are executed in the same order at each site; that is what total ordered multicasting achieves. Also note that there is no ordering requirement on the decision messages (step III.2) and these may be delivered in any order, even before the delivery of the corresponding WM. If this happens, then the sites that receive the decision message before WM simply register the decision, but do not take any action. When WM message arrives, they can execute the lock and write phases and terminate the transaction according to the previously delivered decision message.

This protocol is significantly better, in terms of performance, than the naive one discussed in Section 13.3.2. For each transaction, the master site sends two messages: one when it sends the WM and the second one when it communicates the decision. Thus, if we wish to maintain a system throughput of k transactions-per-second, the total number of messages is $2k$ rather than $k * m$, as is the case with the naive protocol (assuming multicast in both cases). Furthermore, system performance is improved by the use of deferred eager propagation since synchronization among replica sites for all $Write$ operations is done once at the end rather than throughout the transaction execution.

The second example of the use of group communication that we will discuss is in the context of lazy centralized algorithms. Recall that an important issue in this case is to ensure that the refresh transactions are ordered the same way at all the involved slaves so that the database states converge. If totally ordered multicasting is available, the refresh transactions sent by different master sites would be delivered in the same order at all the slaves. However, total order multicast has high messaging overhead which may limit its scalability. It is possible to relax the ordering requirement of the communication system and let the replication protocol take responsibility for ordering the execution of refresh transactions. We will demonstrate this alternative by means of a proposal due to Pacitti et al. [1999]. The protocol assumes FIFO ordered multicast communication with a bounded delay for communication (call it Max), and assumes that the clocks are loosely synchronized so that they may only be out of sync by up to ε. It further assumes that there is an appropriate transaction management functionality at each site. The result of the replication protocol at each slave is to maintain a "running queue" that holds an ordered list of refresh transactions, which is the input to the transaction manager for local execution. Thus, the protocol ensures that the orders in the running queues at each slave site where a set of refresh transactions run are the same.

At each slave site, a "pending queue" is maintained for each master site of this slave (i.e., if the slave site has replicas of x and y whose master sites are $Site_1$ and $Site_2$, respectively, then there are two pending queues, q_1 and q_2, corresponding to master sites $Site_1$ and $Site_2$, respectively). When a refresh transaction RT_i^k is created at a master site $Site_k$, it is assigned a timestamp $ts(RT_i)$ that corresponds to the real time value at the commit time of the corresponding update transaction T_i. When RT_i arrives at a slave, it is put on queue q_k. At each message arrival the top elements of all pending queues are scanned and the one with the lowest timestamp is chosen as the new RT (new_RT) to be handled. If the new_RT has changed since the last cycle (i.e., a new RT arrived with a lower timestamp than what was chosen in the

previous cycle), then the one with the lower timestamp becomes the *new_RT* and is considered for scheduling.

When a refresh transaction is chosen as the *new_RT*, it is not immediately put on the "running queue" for the transaction manager; the scheduling of a refresh transaction takes into account the maximum delay and the possible drift in local clocks. This is done to ensure that any refresh transaction that may be delayed has a chance of reaching the slave. The time when an RT_i is put into the "running queue" at a slave site is $delivery\_time = ts(new\_RT) + Max + \varepsilon$. Since the communication system guarantees an upper bound of Max for message delivery and since the maximum drift in local clocks (that determine timestamps) is ε, a refresh transaction cannot be delayed by more than the $delivery\_time$ before reaching all of the intended slaves. Thus, the protocol guarantees that a refresh transaction is scheduled for execution at a slave when the following hold: (1) all the write operations of the corresponding update transaction are performed at the master, (2) according to the order determined by the timestamp of the refresh transaction (which reflects the commit order of the update transaction), and (3) at the earliest at real time equivalent to its $delivery\_time$. This ensures that the updates on secondary copies at the slave sites follow the same chronological order in which their primary copies were updated and this order will be the same at all of the involved slaves, assuming that the underlying communication infrastructure can guarantee Max and ε. This is an example of a lazy algorithm that ensures 1SR global history, but weak mutual consistency, allowing the replica values to diverge by up to a predetermined time period.

13.5 Replication and Failures

Up to this point, we have focused on replication protocols in the absence of any failures. What happens to mutual consistency concerns if there are system failures? The handling of failures differs between eager replication and lazy replication approaches.

13.5.1 Failures and Lazy Replication

Let us first consider how lazy replication techniques deal with failures. This case is relatively easy since these protocols allow divergence between the master copies and the replicas. Consequently, when communication failures make one or more sites unreachable (the latter due to network partitioning), the sites that are available can simply continue processing. Even in the case of network partitioning, one can allow operations to proceed in multiple partitions independently and then worry about the convergence of the database states upon repair using the conflict resolution techniques discussed in Section 13.3.4. Before the merge, databases at multiple partitions diverge, but they are reconciled at merge time.

13.5.2 Failures and Eager Replication

Let us now focus on eager replication, which is considerably more involved. As we noted earlier, all eager techniques implement some sort of ROWA protocol, ensuring that, when the update transaction commits, all of the replicas have the same value. ROWA family of protocols is attractive and elegant. However, as we saw during the discussion of commit protocols, it has one significant drawback. Even if one of the replicas is unavailable, then the update transaction cannot be terminated. So, ROWA fails in meeting one of the fundamental goals of replication, namely providing higher availability.

An alternative to ROWA which attempts to address the low availability problem is the Read-One/Write-All Available (ROWA-A) protocol. The general idea is that the write commands are executed on all the available copies and the transaction terminates. The copies that were unavailable at the time will have to "catch up" when they become available.

There have been various versions of this protocol [Helal et al., 1997], two of which will be discussed here. The first one is known as the *available copies protocol* [Bernstein and Goodman, 1984; Bernstein et al., 1987]. The coordinator of an update transaction T_i (i.e., the master where the transaction is executing) sends each $W_i(x)$ to all the slave sites where replicas of x reside, and waits for confirmation of execution (or rejection). If it times out before it gets acknowledgement from all the sites, it considers those which have not replied as unavailable and continues with the update on the available sites. The unavailable slave sites update their databases to the latest state when they recover. Note, however, that these sites may not even be aware of the existence of T_i and the update to x that T_i has made if they had become unavailable before T_i started.

There are two complications that need to be addressed. The first one is the possibility that the sites that the coordinator thought were unavailable were in fact up and running and may have already updated x but their acknowledgement may not have reached the coordinator before its timer ran out. Second, some of these sites may have been unavailable when T_i started and may have recovered since then and have started executing transactions. Therefore, the coordinator undertakes a validation procedure before committing T_i:

1. The coordinator checks to see if all the sites it thought were unavailable are still unavailable. It does this by sending an inquiry message to every one of these sites. Those that are available reply. If the coordinator gets a reply from one of these sites, it aborts T_i since it does not know the state that the previously unavailable site is in: it could have been that the site was available all along and had performed the original $W_i(x)$ but its acknowledgement was delayed (in which case everything is fine), or it could be that it was indeed unavailable when T_i started but became available later on and perhaps even executed $W_j(x)$ on behalf of another transaction T_j. In the latter case, continuing with T_i would make the execution schedule non-serializable.

2. If the coordinator of T does not get any response from any of the sites that it thought were unavailable, then it checks to make sure that all the sites that were available when $W_i(x)$ executed are still available. If they are, then T can proceed to commit. Naturally, this second step can be integrated into a commit protocol.

The second ROWA-A variant that we will discuss is the distributed ROWA-A protocol. In this case, each site S maintains a set, V_S, of sites that it believes to be available; this is the "view" that S has of the system configuration. In particular, when a transaction T_i is submitted, its coordinator's view reflects all the sites that the coordinator knows to be available (let us denote this as $V_C(T_i)$ for simplicity). A $R_i(x)$ is performed on any replica in $V_C(T_i)$ and a $W_i(x)$ updates all copies in $V_C(T_i)$. The coordinator checks its view at the end of T_i, and if the view has changed since T_i's start, then T_i is aborted. To modify V, a special atomic transaction is run at all sites, ensuring that no concurrent views are generated. This can be achieved by assigning timestamps to each V when it is generated and ensuring that a site only accepts a new view if its version number is greater than the version number of that site's current view.

The ROWA-A class of protocols are more resilient to failures, including network partitioning, than the simple ROWA protocol.

Another class of eager replication protocols are those based on voting. The fundamental characteristics of voting were presented in the previous chapter when we discussed network partitioning in non-replicated databases. The general ideas hold in the replicated case. Fundamentally, each read and write operation has to obtain a sufficient number of votes to be able to commit. These protocols can be pessimistic or optimistic. In what follows we discuss only pessimistic protocols. An optimistic version compensates transactions to recover if the commit decision cannot be confirmed at completion [Davidson, 1984]. This version is suitable wherever compensating transactions are acceptable (see Chapter 10).

The initial voting algorithm was proposed by Thomas [1979] and an early suggestion to use quorum-based voting for replica control is due to Gifford [1979]. Thomas's algorithm works on fully replicated databases and assigns an equal vote to each site. For any operation of a transaction to execute, it must collect affirmative votes from a majority of the sites. Gifford's algorithm, on the other hand, works with partially replicated databases (as well as with fully replicated ones) and assigns a vote to each copy of a replicated data item. Each operation then has to obtain a *read quorum* (V_r) or a *write quorum* (V_w) to read or write a data item, respectively. If a given data item has a total of V votes, the quorums have to obey the following rules:

1. $V_r + V_w > V$
2. $V_w > V/2$

As the reader may recall from the preceding chapter, the first rule ensures that a data item is not read and written by two transactions concurrently (avoiding the read-write conflict). The second rule, on the other hand, ensures that two write operations

from two transactions cannot occur concurrently on the same data item (avoiding write-write conflict). Thus the two rules ensure that serializability and one-copy equivalence are maintained.

In the case of network partitioning, the quorum-based protocols work well since they basically determine which transactions are going to terminate based on the votes that they can obtain. The vote allocation and threshold rules given above ensure that two transactions that are initiated in two different partitions and access the same data cannot terminate at the same time.

The difficulty with this version of the protocol is that transactions are required to obtain a quorum even to read data. This significantly and unnecessarily slows down read access to the database. We describe below another quorum-based voting protocol that overcomes this serious performance drawback [Abbadi et al., 1985].

The protocol makes certain assumptions about the underlying communication layer and the occurrence of failures. The assumption about failures is that they are "clean." This means two things:

1. Failures that change the network's topology are detected by all sites instantaneously.
2. Each site has a view of the network consisting of all the sites with which it can communicate.

Based on the presence of a communication network that can ensure these two conditions, the replica control protocol is a simple implementation of the ROWA-A principle. When the replica control protocol attempts to read or write a data item, it first checks if a majority of the sites are in the same partition as the site at which the protocol is running. If so, it implements the ROWA rule within that partition: it reads any copy of the data item and writes all copies that are in that partition.

Notice that the read or the write operation will execute in only one partition. Therefore, this is a pessimistic protocol that guarantees one-copy serializability, *but only within that partition*. When the partitioning is repaired, the database is recovered by propagating the results of the update to the other partitions.

A fundamental question with respect to implementation of this protocol is whether or not the failure assumptions are realistic. Unfortunately, they may not be, since most network failures are not "clean." There is a time delay between the occurrence of a failure and its detection by a site. Because of this delay, it is possible for one site to think that it is in one partition when in fact subsequent failures have placed it in another partition. Furthermore, this delay may be different for various sites. Thus two sites that were in the same partition but are now in different partitions may proceed for a while under the assumption that they are still in the same partition. The violations of these two failure assumptions have significant negative consequences on the replica control protocol and its ability to maintain one-copy serializability.

The suggested solution is to build on top of the physical communication layer another layer of abstraction which hides the "unclean" failure characteristics of the physical communication layer and presents to the replica control protocol a communication service that has "clean" failure properties. This new layer of abstraction

provides *virtual partitions* within which the replica control protocol operates. A virtual partition is a group of sites that have agreed on a common view of who is in that partition. Sites join and depart from virtual partitions under the control of this new communication layer, which ensures that the clean failure assumptions hold.

The advantage of this protocol is its simplicity. It does not incur any overhead to maintain a quorum for read accesses. Thus the reads can proceed as fast as they would in a non-partitioned network. Furthermore, it is general enough so that the replica control protocol does not need to differentiate between site failures and network partitions.

Given alternative methods for achieving fault-tolerance in the case of replicated databases, a natural question is what the relative advantages of these methods are. There have been a number of studies that analyze these techniques, each with varying assumptions. A comprehensive study suggests that ROWA-A implementations achieve better scalability and availability than quorum techniques [Jiménez-Peris et al., 2003].

13.6 Replication Mediator Service

The replication protocols we have covered so far are suitable for tightly integrated distributed database systems where we can insert the protocols into each component DBMS. In multidatabase systems, replication support has to be supported outside the DBMSs by mediators. In this section we discuss how to provide replication support at the mediator level by means of an example protocol called NODO [Patiño-Martínez et al., 2000].

The NODO (NOn-Disjoint conflict classes and Optimistic multicast) protocol is a hybrid between distributed and primary copy – it permits transactions to be submitted at any site, but it does have the notion of a primary copy for a data item. It uses group communications and optimistic delivery to reduce latency. The optimistic delivery technique delivers a message optimistically as soon as it is received without guaranteeing any order among messages. The message is said to be "opt-delivered". When the total order of the message is established, then the message is to-delivered. Although optimistic delivery does not guarantee any order, most of the time the order will be the same as total ordering. This fact is exploited by NODO to overlap the total ordering of the transaction request with the transaction execution at the master node, thus masking the latency of total ordering. The protocol also executes transactions optimistically (see Section 11.5), and may abort them if necessary.

In the following discussion, we will assume a fully replicated database for simplicity. This allows us to ignore issues such as finding the primary copy site, how to execute a transaction over a set of data items that have different primary copies, etc. In the fully replicated environment, all of the sites in the system form a multicast group.

It is assumed that the data items are grouped into disjoint sets and each set has a primary copy. Each transaction accesses a particular set of items, and, as in all

primary copy techniques, it first executes at the primary copy site, and its writes are then propagated to the slave sites. The transaction is said to be *local* to its primary copy site.

Each set of data items is called a *conflict class*, and the protocol exploits the knowledge of transactions' conflict classes to increase concurrency. Two transactions that access the same conflict class have a high probability of conflict, while two transactions that access different conflict classes can run in parallel. A transaction can access several conflict classes and this must be statically known before execution (e.g., by analyzing the transaction code). Thus, conflict classes are further abstracted into conflict class groups. Each conflict class group has a single primary copy (i.e., the primary copy of one of the individual conflict classes in the group) where all transactions on that conflict class group must be executed. The same individual conflict class can be in different conflict class groups. For instance, if S_i be the primary copy site of $\{C_x, C_y\}$ and S_j be the primary copy site of $\{C_y\}$, transactions T_1 on $\{C_x, C_y\}$ and T_2 on $\{C_y\}$ are executed at S_i and S_j, respectively.

Each transaction is associated with a single conflict class group, and therefore, it has a single primary copy. Each site manages a number of queues for its incoming transactions, one per individual conflict class (not one per conflict class group). The processing of a transaction proceeds in the following way:

1. A transaction is submitted by an application at a site.
2. That site multicasts the transaction to the multicast group (which is the entire set of sites since we are assuming full replication).
3. When the transaction is opt-delivered at a site, it is appended to the queue of all the individual classes included in its conflict class group.
4. At the primary copy site, when the transaction becomes the first in the queue of all the individual conflict classes of its conflict class group, it is optimistically executed.
5. When the transaction is to-delivered at a site, it is checked whether its optimistic ordering was the same as the total ordering. If the optimistic order was wrong, the transaction is reordered in all the queues according to the total order. The primary copy site, in addition, aborts the transaction (if it was already executed) and re-executes it when it again gets to the head of all the relevant queues. If the optimistic ordering was correct, the primary copy site extracts the resulting write set of the transaction and multicasts (without total ordering) it to the multicast group.
6. When the write set is received at the primary copy site (remember that in this case the primary copy site is also in the multicast group, so it receives its own transmission), it commits the transaction. When the write set is received at a slave site and the transaction becomes the first in all the relevant queues, its write set is applied, and then the transaction commits.

Example 13.6. Let site S_i, respectively S_j, be the master of the conflict class group $\{C_x, C_y\}$, respectively $\{C_x\}$ and $\{C_y\}$. Let transaction T_1 be on $\{C_x, C_y\}$, T_2 on $\{C_y\}$ and T_3 on $\{C_x\}$. Thus, T_1 is local to S_i while T_2 and T_3 are local to S_j. At S_i and S_j, let transaction T_i be the i-th in the total order (i.e., the total order is $T_1 \rightarrow T_2 \rightarrow T_3$). Consider the following state of the queues C_x and C_y at S_i and S_j after the transactions have been opt-delivered.

$$S_i : C_x = [T_1, T_3]; C_y = [T_1, T_2]$$
$$S_j : C_x = [T_3, T_1]; C_y = [T_1, T_2]$$

At S_i T_1 is the first in the queues C_x and C_y and thus it is executed. Similarly, at S_j T_3 is at the head of C_x and thus, executed. When S_i to-delivers T_1, since the optimistic ordering was correct, it extracts T_1's write set and multicasts it. Upon delivering the write set of T_1 at S_i, T_1 is committed. Upon delivering T_1's write set at S_j, it is realized that T_1 was wrongly ordered after T_3, and T_1 is reordered before T_3 and T_3 is aborted since its optimistic ordering was wrong. T_1's write set is then applied and committed. At both S_i and S_j, T_1 is removed from all the queues. Now T_2 and T_3 are first of their queues at S_j, their primary copy site, and both are executed in parallel. Since they are in disjoint conflict class groups, their relative ordering is irrelevant. Now T_2 is to-delivered and since it is optimistic delivery was correct, its write set is extracted and multicast. Upon delivery of the T_2's write set, S_j commits T_2, while S_i applies the write set and commits it. Finally, T_3 is to-delivered and since its execution was performed according to the total order, S_j extracts T_3's write set and multicasts it. Upon delivery of the T_3's writeset, S_j commits T_3. Similarly, S_i applies the write set and commits T_3. The final ordering is $T_1 \rightarrow T_2 \rightarrow T_3$ at both nodes. ♦

Interestingly, there are many cases where, in spite of an ordering mismatch between opt and to-delivery, it is possible to commit transactions consistently by using the optimistic rather than total ordering, thus minimizing the number of aborts due to optimism failures. This fact is exploited by the REORDERING protocol [Patiño-Martínez et al., 2005].

The implementation of the NODO protocol combines concurrency control with group communication primitives and what has been traditionally done inside the DBMS. This solution can be implemented outside a DBMS without a negligible overhead, and thus supports DBMS autonomy Jiménez-Peris et al. [2002]. Similar eager replication protocols have been proposed to support *partial replication*, where copies can be stored at subsets of nodes [Sousa et al., 2001; Serrano et al., 2007]. Unlike full replication, partial replication increases access locality and reduces the number of messages for propagating updates to replicas.

13.7 Conclusion

In this chapter we discussed different approaches to data replication and presented protocols that are appropriate under different circumstances. Each of the alterna-

tive protocols we have discussed have their advantages and disadvantages. Eager centralized protocols are simple to implement, they do not require update coordination across sites, and they are guaranteed to lead to one-copy serializable histories. However, they put a significant load on the master sites, potentially causing them to become bottlenecks. Consequently, they are harder to scale, in particular in the single master site architecture – primary copy versions have better scalability properties since the master responsibilities are somewhat distributed. These protocols result in long response times (the longest among the four alternatives), since the access to any data has to wait until the commit of any transaction that is currently updating it (using 2PC, which is expensive). Furthermore, the local copies are used sparingly, only for read operations. Thus, if the workload is update-intensive, eager centralized protocols are likely to suffer from bad performance.

Eager distributed protocols also guarantee one-copy serializability and provide an elegant symmetric solution where each site performs the same function. However, unless there is communication system support for efficient multicasting, they result in very high number of messages that increase network load and result in high transaction response times. This also constrains their scalability. Furthermore, naive implementations of these protocols will cause significant number of deadlocks since update operations are executed at multiple sites concurrently.

Lazy centralized protocols have very short response times since transactions execute and commit at the master, and do not need to wait for completion at the slave sites. There is also no need to coordinate across sites during the execution of an update transaction, thus reducing the number of messages. On the other hand, mutual consistency (i.e., freshness of data at all copies) is not guaranteed as local copies can be out of date. This means that it is not possible to do a local read and be assured that the most up-to-date copy is read.

Finally, lazy multi-master protocols have the shortest response times and the highest availability. This is because each transaction is executed locally, with no distributed coordination. Only after they commit are the other replicas updated through refresh transactions. However, this is also the shortcoming of these protocols – different replicas can be updated by different transactions, requiring elaborate reconciliation protocols and resulting in lost updates.

Replication has been studied extensively within the distributed computing community as well as the database community. Although there are considerable similarities in the problem definition in the two environments, there are also important differences. Perhaps the two more important differences are the following. Data replication focuses on data, while replication of computation is equally important in distributed computing. In particular, concerns about data replication in mobile environments that involve disconnected operation have received considerable attention. Secondly, database and transaction consistency is of paramount importance in data replication; in distributed computing, consistency concerns are not as high on the list of priorities. Consequently, considerably weaker consistency criteria have been defined.

Replication has been studied within the context of parallel database systems, in particular within parallel database clusters. We discuss these separately in Chapter 14.

13.8 Bibliographic Notes

Replication and replica control protocols have been the subject of significant investigation since early days of distributed database research. This work is summarized very well in [Helal et al., 1997]. Replica control protocols that deal with network partitioning are surveyed in [Davidson et al., 1985].

A landmark paper that defined a framework for various replication algorithms and argued that eager replication is problematic (thus opening up a torrent of activity on lazy techniques) is [Gray et al., 1996]. The characterization that we use in this chapter is based on this framework. A more detailed characterization is given in [Wiesmann et al., 2000]. A recent survey on optimistic (or lazy) replication techniques is [Saito and Shapiro, 2005]. The entire topic is discussed at length in [Kemme et al., 2010]

Freshness, in particular for lazy techniques, have been a topic of some study. Alternative techniques to ensure "better" freshness are discussed in [Pacitti et al., 1998; Pacitti and Simon, 2000; Röhm et al., 2002a; Pape et al., 2004; Akal et al., 2005].

There are many different versions of quorum-based protocols. Some of these are discussed in [Triantafillou and Taylor, 1995; Paris, 1986; Tanenbaum and van Renesse, 1988]. Besides the algorithms we have described here, some notable others are given in [Davidson, 1984; Eager and Sevcik, 1983; Herlihy, 1987; Minoura and Wiederhold, 1982; Skeen and Wright, 1984; Wright, 1983]. These algorithms are generally called *static* since the vote assignments and read/write quorums are fixed a priori. An analysis of one such protocol (such analyses are rare) is given in [Kumar and Segev, 1993]. Examples of *dynamic replication protocols* are in [Jajodia and Mutchler, 1987; Barbara et al., 1986, 1989] among others. It is also possible to change the way data are replicated. Such protocols are called *adaptive* and one example is described in [Wolfson, 1987].

An interesting replication algorithm based on economic models is described in [Sidell et al., 1996].

Exercises

Problem 13.1. For each of the four replication protocols (eager centralized, eager distributed, lazy centralized, lazy distributed), give a scenario/application where the approach is more suitable than the other approaches. Explain why.

Problem 13.2. A company has several geographically distributed warehouses storing and selling products. Consider the following partial database schema:

ITEM(ID, ItemName, Price, ...)

STOCK(ID, Warehouse, Quantity, ...)

CUSTOMER(ID, CustName, Address, CreditAmt, ...)

CLIENT-ORDER(ID, Warehouse, Balance, ...)

ORDER(ID, Warehouse, CustID, Date)

ORDER-LINE(ID, ItemID, Amount, ...)

The database contains relations with product information (ITEM contains the general product information, STOCK contains, for each product and for each warehouse, the number of pieces currently on stock). Furthermore, the database stores information about the clients/customers, e.g., general information about the clients is stored in the CUSTOMER table. The main activities regarding the clients are the rdering of products, the payment of bills and general information requests. There exist several tables to register the orders of a customer. Each order is regustered in the ORDER and ORDER−LINE tables. For each order/purchase, one entry exists in the order table, having an ID, indicating the customer-id, the warehouse at which the order was submitted, the date of the order, etc. A client can have several orders pending at a warehouse. Within each order, several products can be ordered. ORDER−LINE contains an entry for each product of the order, which may include one or more products. CLIENT−ORDER is a summary table that lists, for each client and for each warehouse, the sum of all existing orders.

(a) The company has a customer service group consisting of several employees that receive customers' orders and payments, query the data of local customers to write bills or register paychecks, etc. Furthermore, they answer any type of requests which the customers might have. For instance, ordering products changes (update/insert) the CLIENT−ORDER, ORDER, ORDER−LINE, and STOCK tables. To be flexible, each employee must be able to work with any of the clients. The workload is estimated to be 80% queries and 20% updates. Since the workload is query oriented, the management has decided to build a cluster of PCs each equipped with its own database to accelerate queries through fast local access. How would you replicate the data for this purpose? Which replica control protocol(s) wold you use to keep the data consistent?

(b) The company's management has to decide each fiscal quarter on their product offerings and sales strategies. For this purpose, they must continually observe and analyze the sales of the different products at the different warehouses as well as observe consumer behavior. How would you replicate the data for this purpose? Which replica control protocol(s) would you use to keep the data consistent?

Problem 13.3 (*). An alternative to ensuring that the refresh transactions can be applied at all of the slaves in the same order in lazy single master protocols with limited transparency is the use of a replication graph as discussed in Section 13.3.3. Develop a method for distributed management of the replication graph.

Problem 13.4. Consider data items x and y replicated across the sites as follows:

Site 1	Site 2	Site 3	Site 4
x	x		x
	y	y	y

(a) Assign votes to each site and give the read and write quorum.
(b) Determine the possible ways that the network can partition and for each specify in which group of sites a transaction that updates (reads and writes) x can be terminated and what the termination condition would be.
(c) Repeat (**b**) for y.

Problem 13.5 (**). In the NODO protocol, we have seen that each conflict class group has a master. However, this is not inherent to the protocol. Design a multi-master variation of NODO in which a transaction might be executed by any replica. What condition should be enforced to guarantee that each updated transaction is processed only by one replica?

Problem 13.6 (**). In the NODO protocol, if the DBMS could provide additional introspection functionality, it would be possible to execute in certain circumstances transactions of the same conflict class in parallel. Determine which functionality would be needed from the DBMS. Also characterize formally under which circumstances concurrent execution of transactions in the same conflict class could be allowed to be executed in parallel whilst respecting 1-copy consistency. Extend the NODO protocol with this enhancement.

Chapter 14
Parallel Database Systems

Many data-intensive applications require support for very large databases (e.g., hundreds of terabytes or petabytes). Examples of such applications are e-commerce, data warehousing, and data mining. Very large databases are typically accessed through high numbers of concurrent transactions (e.g., performing on-line orders on an electronic store) or complex queries (e.g., decision-support queries). The first kind of access is representative of On-Line Transaction Processing (OLTP) applications while the second is representative of On-Line Analytical Processing (OLAP) applications. Supporting very large databases efficiently for either OLTP or OLAP can be addressed by combining parallel computing and distributed database management.

As introduced in Chapter 1, a parallel computer, or multiprocessor, is a special kind 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. The main idea is to build a very powerful computer out of many small computers, each with a very good cost/performance ratio, at a much lower cost than equivalent mainframe computers. As discussed in Chapter 1, 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 [DeWitt and Gray, 1992; Valduriez, 1993]. 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 that provides a good basis for data-based parallelism. 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 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

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_14,

than 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 Section 14.1, we clarify the objectives, and discuss the functional and architectural aspects of parallel database systems. In particular, we discuss the respective advantages and limitations of the parallel system architectures (shared-memory, shared-disk, shared-nothing) along several important dimensions including the perspective of both end-users, database administrators and system developers. Then, we present the techniques for data placement in Section 14.2, query processing in Section 14.3 and load balancing in Section 14.4.

In Section 14.5, we present the use of parallel data management techniques in database clusters, an important type of parallel database system implemented on a cluster of PCs.

14.1 Parallel Database System Architectures

In this section we show the value of parallel systems for efficient database management. We motivate the needs for parallel database systems by reviewing the requirements of very large information systems using current hardware technology trends. We present the functional and architectural aspects of parallel database systems. In particular, we present and compare the main architectures: shared-memory, shared-disk, shared-nothing and hybrid architectures.

14.1.1 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 by improving the response time of numerical applications [Kowalik, 1985; Sharp, 1987]. The developments in both general-purpose parallel computers using standard microprocessors and parallel programming techniques [Osterhaug, 1989] 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 70s and 80s [Hsiao, 1983]. The problem faced by conventional database management has long been known as "I/O bottleneck" [Boral and DeWitt, 1983], 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. A notable exception to these failures was the CAFS-ISP hardware-based filtering device [Babb, 1979] that was bundled within disk controllers for fast associative search. 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 [Keeton et al., 1998]. 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 [Eich, 1989], 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 covered by those of distributed DBMS (performance, availability, extensibility). Ideally, a parallel database system should provide the following advantages.

1. **High-performance.** This can be obtained through several complementary solutions: database-oriented operating system support, parallel data management, query optimization, and load balancing. Having the operating system constrained and "aware" of the specific database requirements (e.g., buffer management) simplifies the implementation of low-level database functions and therefore decreases their cost. For instance, the cost of a message can be significantly reduced to a few hundred instructions by specializing the communication protocol. Parallelism can increase throughput, using inter-query parallelism, and decrease transaction response times, using intra-query parallelism. 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 run-time.
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 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 uninterupted service to users. However, it is essential that a node failure does not crate load imbalance, e.g., by doubling the load on the available copy. Solutions to this problem require partitioning copies in such a way that they can also be accessed in parallel.
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 [DeWitt and Gray, 1992]: *linear speedup* and *linear scaleup* see Figure 14.1. Linear speedup refers to a linear increase in performance for a constant database size while the number of nodes (i.e., processing and storage power) are increased linearly. Linear scaleup refers to a sustained performance for a linear increase in both database size and number of nodes. Furthermore, extending the system should require minimal reorganization of the existing database.

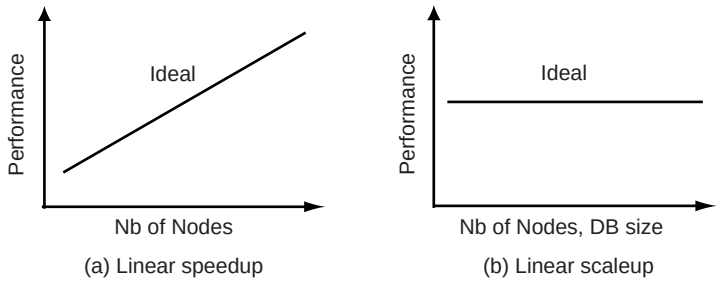

Fig. 14.1 Extensibility Metrics

14.1.2 Functional 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 14.2 shows the architecture using these subsystems due to Bergsten et al. [1991].

1. **Session Manager.** It plays the role of a transaction monitor, providing support for client interactions with the server. In particular, it performs the connections and disconnections between the client processes and the two other subsystems. Therefore, it initiates and closes user sessions (which may contain multiple transactions). In case of OLTP sessions, the session manager is able to trigger the execution of pre-loaded transaction code within data manager modules.
2. **transaction Manager.** It receives client transactions related to query compilation and execution. It can access the database directory that holds all meta-information about data and programs. The directory itself should be managed as a database in the server. Depending on the transaction, it activates the various compilation phases, triggers query execution, and returns the results as well as error codes to the client application. Because it supervises transaction execution and commit, it may trigger the recovery procedure in case of transaction failure. To speed up query execution, it may optimize and parallelize the query at compile-time.
3. **Data Manager.** It provides all the low-level functions needed to run compiled queries in parallel, i.e., database operator execution, parallel transaction support, cache management, etc. If the transaction manager is able to compile dataflow control, then synchronization and communication among data manager modules is possible. Otherwise, transaction control and synchronization must be done by a transaction manager module.

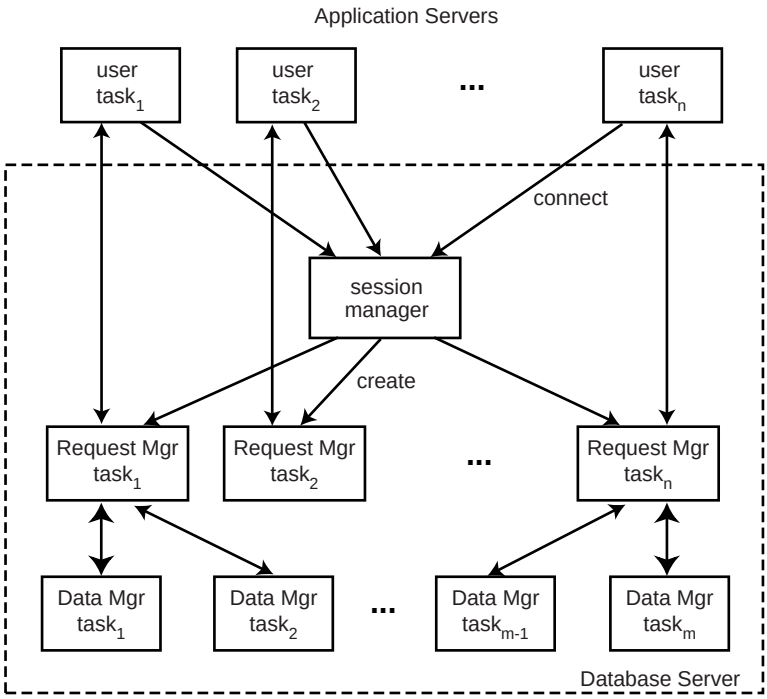

Fig. 14.2 General Architecture of a Parallel Database System

14.1.3 Parallel DBMS Architectures

As any system, 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, main memory, and disks, are connected through some fast interconnection network. There are three basic parallel computer architectures depending on how main memory or disk is shared: *shared-memory*, *shared-disk* and *shared-nothing*. Hybrid architectures such as NUMA or *cluster* try to combine the benefits of the basic architectures. In the rest of this section, when describing parallel architectures, we focus on the four main hardware elements: interconnect, processors (P), main memory (M) and disks. For simplicity, we ignore other elements such as processor cache and I/O bus.

14.1.3.1 Shared-Memory

In the shared-memory approach (see Figure 14.3), any processor has access to any memory module or disk unit through a fast interconnect (e.g., a high-speed bus or a cross-bar switch). All the processors are under the control of a single operating system.

Current mainframe designs and symmetric multiprocessors (SMP) follow this approach. Examples of shared-memory parallel database systems include XPRS [Hong, 1992], DBS3 [Bergsten et al., 1991], and Volcano [Graefe, 1990], as well as portings of major commercial DBMSs on SMP. In a sense, the implementation of DB2 on an IBM3090 with 6 processors [Cheng et al., 1984] was the first example. All shared-memory parallel database products today can exploit inter-query parallelism to provide high transaction throughput and intra-query parallelism to reduce response time of decision-support queries.

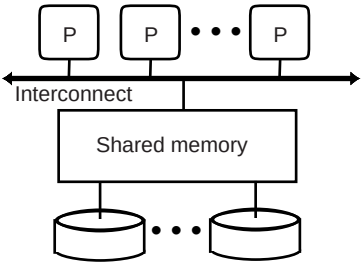

Fig. 14.3 Shared-Memory Architecture

Shared-memory has two strong advantages: simplicity and load balancing. Since meta-information (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, inter-query parallelism comes for free. Intra-query parallelism requires some parallelization but remains rather simple. Load balancing is easy to achieve since it can be achieved at run-time using the shared-memory by allocating each new task to the least busy processor.

Shared-memory has three problems: high cost, limited extensibility and low availability. High cost is 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 processors (even with larger caches), conflicting accesses to the shared-memory increase rapidly and degrade performance [Thakkar and Sweiger, 1990]. Therefore, extensibility is limited to a few tens of processors, typically up to 16 for the best cost/performance using 4-processor boards. Finally, since the memory space is shared by all processors, a memory fault may affect most processors thereby hurting availability. The solution is to use duplex memory with a redundant interconnect.

14.1.3.2 Shared-Disk

In the shared-disk approach (see Figure 14.4), any processor has access to any disk unit through the interconnect but exclusive (non-shared) access to its main memory. Each processor-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 Chapter 11. The first parallel DBMS that used shared-disk is Oracle with an efficient implementation of a distributed lock manager for cache consistency. Other major DBMS vendors such as IBM, Microsoft and Sybase provide shared-disk implementations.

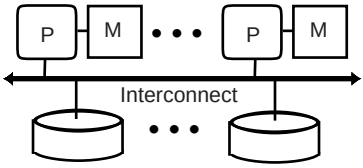

Fig. 14.4 Shared-Disk Architecture

Shared-disk has a number of advantages: lower cost, high extensibility, load balancing, availability, and easy migration from centralized systems. The cost of the interconnect is significantly less than with shared-memory since standard bus technology may be used. Given that each processor has enough main memory, interference on the shared disk can be minimized. Thus, extensibility can be better, typically up to a hundred processors. Since memory faults can be isolated from other nodes, availability can be higher. Finally, migrating from a centralized system to shared-disk is relatively straightforward since the data on disk need not be reorganized.

Shared-disk suffers from higher complexity and potential performance problems. It requires distributed database system protocols, such as distributed locking and two-phase commit. As we have discussed in previous chapters, these can be complex. Furthermore, maintaining cache consistency can incur high communication overhead among the nodes. Finally, access to the shared-disk is a potential bottleneck.

14.1.3.3 Shared-Nothing

In the shared-nothing approach (see Figure 14.5), each processor has exclusive access to its main memory and disk unit(s). Similar to shared-disk, each processor-memory-disk node is under the control of its own copy of the operating system. Then, each node can be viewed as a local site (with its own database and software) in a distributed database system. Therefore, most solutions designed for distributed databases 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).

Many research prototypes have adopted the shared-nothing architecture, e.g., BUBBA [Boral et al., 1990], EDS [Group, 1990], GAMMA [DeWitt et al., 1986], GRACE [Fushimi et al., 1986], and PRISMA [Apers et al., 1992], because it can scale. The first major parallel DBMS product was Teradata's Database Computer that could accommodate a thousand processors in its early version. Other major DBMS vendors such as IBM, Microsoft and Sybase provide shared-nothing implementations.

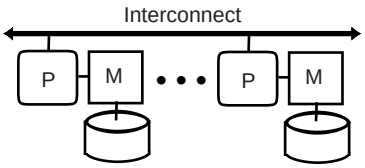

Fig. 14.5 Shared-Nothing Architecture

As demonstrated by the existing products, shared-nothing has three main virtues: lower cost, high extensibility, and high availability. The cost advantage is better than that of shared-disk that requires a special interconnect for the disks. By implementing a distributed database design that favors the smooth incremental growth of the system by the addition of new nodes, extensibility can be better (in the thousands of nodes). With careful partitioning of the data on multiple disks, almost linear speedup and linear scaleup could be achieved for simple workloads. Finally, by replicating data on multiple nodes, high availability can also be achieved.

Shared-nothing is much more complex to manage than either shared-memory or shared-disk. Higher complexity is due to the necessary implementation of distributed database functions assuming large numbers of nodes. In addition, load balancing is more difficult to achieve because it relies on the effectiveness of database partitioning for the query workloads. Unlike shared-memory and shared-disk, load balancing is decided based on data location and not the actual load of the system. Furthermore, the addition of new nodes in the system presumably requires reorganizing the database to deal with the load balancing issues.

14.1.3.4 Hybrid Architectures

Various possible combinations of the three basic architectures are possible to obtain different trade-offs between cost, performance, extensibility, availability, etc. Hybrid architectures try to obtain the advantages of different architectures: typically the efficiency and simplicity of shared-memory and the extensibility and cost of either shared disk or shared nothing. In this section, we discuss two popular hybrid architectures: NUMA and cluster.

NUMA.

With shared-memory, each processor has *uniform memory access* (UMA), with constant access time, since both the virtual memory and the physical memory are shared. One major advantage is that the programming model based on shared virtual memory is simple. With either shared-disk or shared-nothing, both virtual and shared memory are distributed, which yields scalability to large numbers of processors. The objective of NUMA is to provide a shared-memory programming model and all its benefits, in a scalable architecture with distributed memory. The term NUMA reflects the fact that an access to the (virtually) shared memory may have a different cost depending on whether the physical memory is local or remote to the processor. The most successful class of NUMA multiprocessors is Cache Coherent NUMA (CC-NUMA) [Goodman and Woest, 1988; Lenoski et al., 1992]. With CC-NUMA, the main memory is physically distributed among the nodes as with shared-nothing or shared-disk. However, any processor has access to all other processors' memories (see Figure 14.6). Each node can itself be an SMP. Similar to shared-disk, different processors can access the same data in a conflicting update mode, so global cache consistency protocols are needed. In order to make remote memory access efficient, the only viable solution is to have cache consistency done in hardware through a special consistent cache interconnect [Lenoski et al., 1992]. Because shared-memory and cache consistency are supported by hardware, remote memory access is very efficient, only several times (typically between 2 and 3 times) the cost of local access.

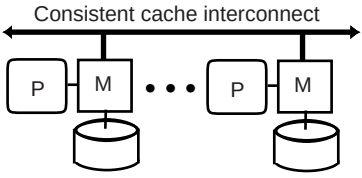

Fig. 14.6 Cache coherent NUMA (CC-NUMA)

Most SMP manufacturers are now offering NUMA systems that can scale up to a hundred processors. The strong argument for NUMA is that it does not require any rewriting of the application software. However some rewriting is still necessary in the database engine (and the operating system) to take full advantage of access locality [Bouganim et al., 1999].

Cluster.

A cluster is a set of independent server nodes interconnected to share resources and form a single system. The shared resources, called *clustered* resources, can be hardware such as disk or software such as data management services. The server nodes are made of off-the-shelf components ranging from simple PC components

to more powerful SMP. Using many off-the-shelf components is essential to obtain the best cost/performance ratio while exploiting continuing progress in hardware components. In its cheapest form, the interconnect can be a local network. However, there are now fast standard interconnects for clusters (e.g., Myrinet and Infiniband) that provide high bandwidth (Gigabits/sec) with low latency for message traffic.

Compared to a distributed system, a cluster is geographically concentrated (at a single site) and made of homogeneous nodes. Its architecture can be either shared-nothing or shared-disk. Shared-nothing clusters have been widely used because they can provide the best cost/performance ratio and scale up to very large configurations (thousands of nodes). However, because each disk is directly connected to a computer via a bus, adding or replacing cluster nodes requires disk and data reorganization. Shared-disk avoids such reorganization but 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 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). In fact, disks in a SAN are attached to the network instead to the bus as happens in Directly Attached Storage (DAS), but otherwise they are handled as sharable local disks. Existing protocols for SANs extend their local disk counterparts to run over a network (e.g., i-SCSI extends SCSI, and ATA-over-Ethernet extends ATA). As a result, SAN provides high data throughput and can scale up to large numbers of nodes. Its only limitation with respect to shared-nothing is its higher cost of ownership.

A cluster architecture has important advantages. It combines the flexibility and performance of shared-memory at each node with the extensibility and availability of shared-nothing or shared-disk. Furthermore, using off-the-shelf shared-memory nodes with a standard cluster interconnect makes it a cost-effective alternative to proprietary high-end multiprocessors such as NUMA or MPP. Finally, using SAN eases disk management and data placement.

14.1.3.5 Discussion

Let us briefly compare the three basic architectures based on their potential advantages (high-performance, high-availability, and extensibility). It is fair to say that, for a small configuration (e.g., less than 20 processors), shared-memory can provide the highest performance because of better load balancing. Shared-disk and shared-nothing architectures outperform shared-memory in terms of extensibility. Some years ago, shared-nothing was the only choice for high-end systems. However, recent progress in disk connectivity technologies such as SAN make shared-disk a viable

alternative with the main advantage of simplifying data administration and DBMS implementation. In particular, shared-disk is now the preferred architecture for OLTP applications because it is easier to support ACID transactions and distributed concurrency control. But for OLAP databases that are typically very large and mostly read-only, shared-nothing is the preferred architecture. Most major DBMS vendors now provide a shared-nothing implementation of their DBMS for OLAP, in addition to a shared-disk version for OLTP. The only execption is Oracle that uses shared-disk for both OLTP and OLAP.

Hybrid architectures, such as NUMA and cluster, can combine the efficiency and simplicity of shared-memory and the extensibility and cost of either shared disk or shared nothing. In particular, they can exploit continous progress in SMP and use shared-memory nodes with excellent cost/performance ratio. Both NUMA and cluster can scale up to large configurations (hundred of nodes). The main advantage of NUMA over a cluster is the simple (shared-memory) programming model that eases database administration and tuning. However, using standard PC nodes and interconnects, clusters provide a better overall cost/performance ratio, and, using shared-nothing, they can scale up to very large configurations (thousands of nodes).

14.2 Parallel Data Placement

In this section, we assume a shared-nothing architecture because it is the most general case and its implementation techniques also apply, sometimes in a simplified form, to other architectures. Data placement in a parallel database system exhibits similarities with data fragmentation in distributed databases (see Chapter 3). An obvious similarity is that fragmentation can be used to increase parallelism. In what follows, we use the terms *partitioning* and *partition* instead of horizontal fragmentation and horizontal fragment, respectively, to contrast with the alternative strategy, which consists of *clustering* a relation at a single node. The term *declustering* is sometimes used to mean partitioning [Livny et al., 1987]. Vertical fragmentation can also be used to increase parallelism and load balancing much as in distributed databases. Another similarity is that since data are much larger than programs, execution should occur, as much as possible, where the data reside. However, 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 reside, data placement is a critical performance issue.

Data placement must be done so as to maximize system performance, which can be measured by combining the total amount of work done by the system and the response time of individual queries. In Chapter 8, we have seen that maximizing response time (through intra-query parallelism) results in increased total work due

to communication overhead. For the same reason, inter-query parallelism results in increased total work. On the other hand, clustering all the data necessary to a program minimizes communication and thus the total work done by the system in executing that program. In terms of data placement, we have the following trade-off: maximizing response time or inter-query parallelism leads to partitioning, whereas minimizing the total amount of work leads to clustering. As we have seen in Chapter 3, this problem is addressed in distributed databases in a rather static manner. The database administrator is in charge of periodically examining fragment access frequencies, and when necessary, moving and reorganizing fragments.

An alternative solution to data placement is *full partitioning*, whereby each relation is horizontally fragmented across *all* the nodes in the system. There are three basic strategies for data partitioning: round-robin, hash, and range partitioning (Figure 14.7).

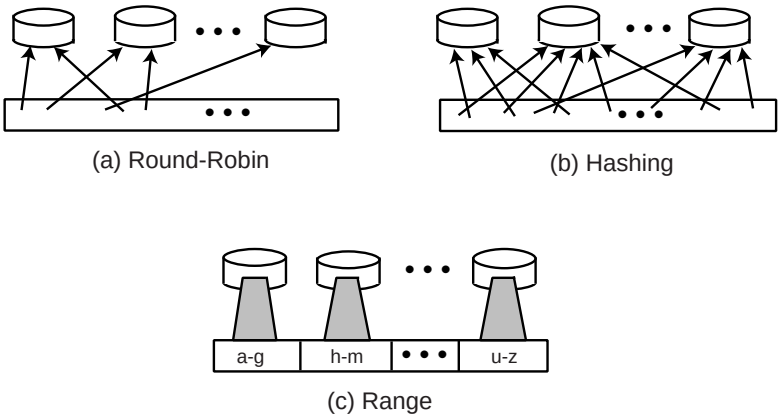

Fig. 14.7 Different Partitioning Schemes

1. *Round-robin partitioning* is the simplest strategy, it ensures uniform data distribution. With n partitions, the ith tuple in insertion order is assigned to partition (i mod n). This strategy enables the sequential access to a relation to be done in parallel. However, the direct access to individual tuples, based on a predicate, requires accessing the entire relation.
2. *Hash partitioning* applies a hash function to some attribute that yields the partition number. This strategy allows exact-match queries on the selection attribute to be processed by exactly one node and all other queries to be processed by all the nodes in parallel.
3. *Range partitioning* distributes tuples based on the value intervals (ranges) of some attribute. In addition to supporting exact-match queries (as in hashing), it is well-suited for range queries. For instance, a query with a predicate "A between A_1 and A_2" may be processed by the only node(s) containing tuples

whose A value is in range $[A_1, A_2]$. However, range partitioning can result in high variation in partition size.

Compared to clustering relations on a single (possibly very large) disk, full partitioning yields better performance [Livny et al., 1987]. Although full partitioning has obvious performance advantages, highly parallel execution might cause a serious performance overhead for complex queries involving joins. Furthermore, full partitioning is not appropriate for small relations that span a few disk blocks. These drawbacks suggest that a compromise between clustering and full partitioning (i.e., *variable partitioning*), needs to be found.

A solution is to do data placement by variable partitioning [Copeland et al., 1988]. The degree of partitioning, i.e., the number of nodes over which a relation is fragmented, is a function of the size and access frequency of the relation. This strategy is much more involved than either clustering or full partitioning because changes in data distribution may result in 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 variable 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 programs that run on the database server. In particular, programs should not be recompiled because of reorganization. Therefore, the compiled programs should remain independent of data location, which may change rapidly. Such independence can be achieved if the run-time 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 [Khoshafian and Valduriez, 1987]. 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 [Bayer and McCreight, 1972]. 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 r queried elation. Using a B-tree index (usually much larger than a hashed index) enables more efficient processing of range queries, where only the nodes containing data in the specified range are accessed.

Example 14.1. Figure 14.8 provides an example of a global index and a local index for relation EMP(ENO, ENAME, DEPT, 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 on set name maps the name EMP onto the index on attribute ENO 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

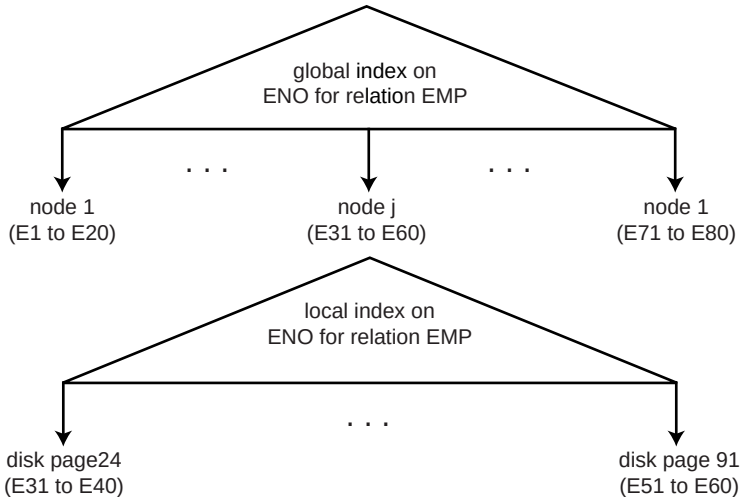

Fig. 14.8 Example of Global and Local Indexes

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. ♦

Experimental results for variable partitioning of a workload consisting of a mix of short transactions (debit-credit like) and complex ones indicate that as partitioning is increased, throughput continues to increase for short transactions. However, for complex transactions involving several large joins, further partitioning reduces throughput because of communication overhead [Copeland et al., 1988].

A serious problem in data placement is dealing with skewed data distributions that may lead to non-uniform partitioning and hurt load balancing. Range partitioning is more sensitive to skew than either round-robin or hash partitioning. A solution is to treat non-uniform partitions appropriately, e.g., by further fragmenting large partitions. The separation between logical and physical nodes is also useful since a logical node may correspond to several physical nodes.

A final complicating factor is data replication for high availability. The simple solution is to maintain two copies of the same data, a primary and a backup copy, on two separate nodes. This is the *mirrored disks* architecture promoted by many computer manufacturers. However, in case of a node failure, the load of the node with the copy may double, thereby hurting load balancing. To avoid this problem, several high-availability data replication strategies have been proposed for parallel database systems [Hsiao and DeWitt, 1991]. An interesting solution is Teradata's interleaved partitioning that further partitions the backup copy on a number of nodes. Figure 14.9 illustrates the interleaved partitioning of relation R over four nodes, where each

primary copy of a partition, e.g., R_1, is futher divided in 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.

Node	1	2	3	4
Primary copy	R_1	R_2	R_3	R_4
Backup copy	$r_{2.3}$ $r_{3.2}$	$r_{1.1}$ $r_{3.3}$	$r_{1.2}$ $r_{2.1}$	$r_{1.3}$ $r_{2.2}$ $r_{3.1}$

Fig. 14.9 Example of Interleaved Partitioning

A better solution is Gamma's *chained partitioning* [Hsiao and DeWitt, 1991], which stores the primary and backup copy on two adjacent nodes (Figure 14.10). The main idea is that 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 are 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.

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. 14.10 Example of Chained Partitioning

14.3 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 various forms of query parallelism. Then we derive basic parallel algorithms for data processing. Finally, we discuss parallel query optimization.

14.3.1 Query Parallelism

Parallel query execution can exploit two forms of parallelism: inter- and intra-query. *Inter-query parallelism* enables the parallel execution of multiple queries generated by concurrent transactions, in order to increase the transactional throughput. Within a query (*intra-query parallelism*), *inter-operator* and *intra-operator parallelism* are used to decrease response time. Inter-operator parallelism is obtained by executing in parallel several operators of the query tree on several processors while with intra-operator parallelism, the same operator is executed by many processors, each one working on a subset of the data. Note that these two forms of parallelism also exist in distributed query processing.

14.3.1.1 Intra-operator Parallelism

Intra-operator parallelism is based on the decomposition of one operator in a set of independent sub-operators, called *operator instances*. This decomposition is done using static and/or dynamic partitioning of relations. Each operator instance will then process one relation partition, also called a *bucket*. The operator decomposition frequently benefits from the initial partitioning of the data (e.g., the data are partitioned on the join attribute). To illustrate intra-operator parallelism, let us consider a simple select-join query. The select operator can be directly decomposed into several select operators, each on a different partition, and no redistribution is required (Figure 14.11). Note that if the relation is partitioned on the select attribute, partitioning properties can be used to eliminate some select instances. For example, in an exact-match select, only one select instance will be executed if the relation was partitioned by hashing (or range) on the select attribute. It is more complex to decompose the join operator. In order to have independent joins, each bucket of the first relation R may be joined to the entire relation S. Such a join will be very inefficient (unless S is very small) because it will imply a broadcast of S on each participating processor. A more efficient way is to use partitioning properties. For example, if R and S are partitioned by hashing on the join attribute and if the join is an equijoin, then we can partition the join into independent joins (see Algorithm 14.3 in Section 14.3.2). This is the ideal case that cannot be always used, because it depends on the initial partitioning of R and S. In the other cases, one or two operands may be repartitioned [Valduriez and Gardarin, 1984]. Finally, we may notice that the partitioning function (hash, range, round robin) is independent of the local algorithm (e.g., nested loop, hash, sort merge) used to process the join operator (i.e., on each processor). For instance, a hash

join using a hash partitioning needs two hash functions. The first one, h_1, is used to partition the two base relations on the join attribute. The second one, h_2, which can be different for each processor, is used to process the join on each processor.

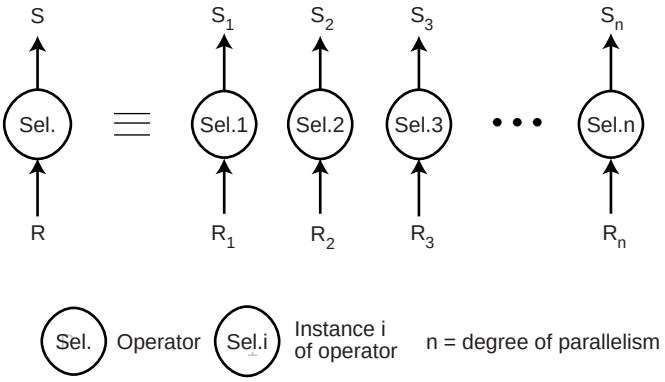

Fig. 14.11 Intra-operator Parallelism

14.3.1.2 Inter-operator Parallelism

Two forms of inter-operator parallelism can be exploited. With *pipeline parallelism*, several operators with a producer-consumer link are executed in parallel. For instance, the select operator in Figure 14.12 will be executed in parallel with the join operator. The advantage of such execution is that the intermediate result is not materialized, thus saving memory and disk accesses. In the example of Figure 14.12, only S may fit in memory. *Independent parallelism* is achieved when there is no dependency between the operators that are executed in parallel. For instance, the two select operators of Figure 14.12 can be executed in parallel. This form of parallelism is very attractive because there is no interference between the processors.

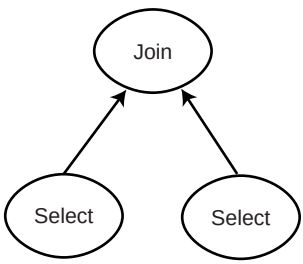

Fig. 14.12 Inter-operator Parallelism

14.3.2 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 an 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 nodes. Parallel algorithms for relational algebra operators are the building blocks necessary for parallel query processing.

Parallel data processing should exploit intra-operator parallelism. We concentrate our presentation of parallel algorithms for database operators on the select and join operators, since all other binary operators (such as union) can be handled very much like join [Bratbergsengen, 1984]. 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 Figure 14.8), a select operator with a range predicate may be executed only by the nodes that store relevant data.

The parallel processing of join is significantly more involved than that of select. The distributed join algorithms designed for high-speed networks (see Chapter 8) can be applied successfully in a partitioned database context. However, the availability of a global index at run time provides more opportunities for efficient parallel execution. In the following, we introduce three basic parallel join algorithms for partitioned databases: the parallel nested loop (PNL) algorithm, the parallel associative join (PAJ) algorithm, and the parallel hash join (PHJ) algorithm. We describe each using a pseudo-concurrent 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. **Send** and **receive** are the basic communication primitives to transfer data between nodes. **Send** enables data to be sent from one node to one or more nodes. The destination nodes are typically obtained from the global index. **Receive** gets the content of the data sent to 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).

The parallel nested loop algorithm [Bitton et al., 1983] is the simplest one and the most general. It basically composes the Cartesian product of

relations *R* and *S* in parallel. Therefore, arbitrarily complex join predicates may be supported. This algorithm has been introduced in Chapter 8 in the context of Distributed INGRES. It is more precisely described in Algorithm 14.1, where the join result is produced at the *S*-nodes. The algorithm proceeds in two phases.

In the first phase, each fragment of R is sent and replicated at each node containing a fragment of S (there are n such nodes). This phase is done in parallel by m nodes and is efficient if the communication network has a broadcast capability. In this case each fragment of R can be broadcast to n nodes in a single transfer, thereby incurring a total communication cost of m messages. Otherwise, $(m * n)$ 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 are received. If a nested loop join algorithm 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 $\cup_{i=1}^{n}(R \bowtie S_i)$.

Algorithm 14.1: PNL Algorithm

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
  for *j from* 1 *to n in parallel* **do** {perform the join at each S-node}
    $R \leftarrow \bigcup_{i=1}^{m} R_i$; {receive R_i from R-nodes; R is fully replicated at S-nodes}
    $T_j \leftarrow R \bowtie_{JP} S_j$
end

Example 14.2. Figure 14.13 shows the application of the parallel nested loop algorithm with $m = n = 2$. ♦

The parallel associative join algorithm, shown in Algorithm 14.2, applies only in the case of equijoin with one of the operand relations partitioned according to the join attribute. To simplify the description of the algorithm, we assume that the equijoin predicate is on attribute A from R, and B from S. Furthermore, relation S is partitioned according to the hash function h applied to join attribute B, meaning that all the tuples of S that have the same value for $h(B)$ are placed at the same node. No knowledge of how R is partitioned is assumed.

The application of the parallel associative join algorithm will produce the join result at the nodes where S_i exists (i.e., the S-nodes).

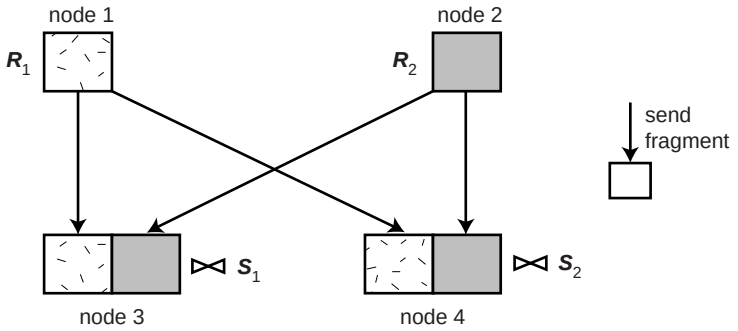

Fig. 14.13 Example of Parallel Nested Loop

Algorithm 14.2: PAJ Algorithm

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
  {we assume that JP is $R.A = S.B$ and relation S is fragmented according to the function $h(B)$}
  for *i from* 1 *to m in parallel* **do** {send R associatively to each S-node}
    $R_{ij} \leftarrow$ apply $h(A)$ to R_i $(j = 1, \ldots, n)$
  for *j from* 1 *to n in parallel* **do**
    send R_{ij} to the node storing S_j
  for *j from* 1 *to n in parallel* **do** {perform the join at each S-node}
    $R_j \leftarrow \bigcup_{i=1}^{m} R_{ij}$; {receive only the useful subset of R}
    $T_j \leftarrow R_j \bowtie_{JP} S_j$
end

The algorithm proceeds in two phases. In the first phase, relation R is sent associatively to the S-nodes based on the hash function h applied to attribute A. This guarantees that a tuple of R with hash value v is sent only to the S-node that contains tuples with hash value v. The first phase is done in parallel by m nodes where R_i's exist. Thus, unlike the parallel nested loop join algorithm, the tuples of R get distributed but not replicated across the S-nodes. This is reflected in the first two Parallel-do statements of the algorithm where each node i produces m fragments of R_i and sends each fragment R_{ij} to the node storing S_j.

In the second phase, each S-node j receives in parallel from the different R-nodes the relevant subset of R (i.e., R_j) and joins it locally with the fragments S_j. Local join processing can be done as in the parallel nested loop join algorithm.

To summarize, the parallel associative join algorithm replaces the operator $R \bowtie S$ by $\cup_{i=1}^{n}(R_i \bowtie S_i)$.

Example 14.3. Figure 14.14 shows the application of the parallel associative join algorithm with $m = n = 2$. The squares that are hatched with the same pattern indicate fragments whose tuples match the same hash function. ♦

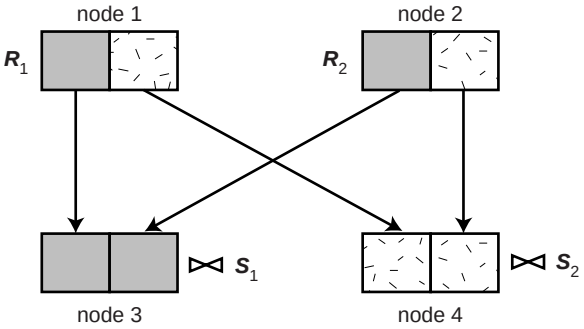

Fig. 14.14 Example of Parallel Associative Join

The parallel hash join algorithm, shown in Algorithm 14.3, can be viewed as a generalization of the parallel associative join algorithm. It also applies in the case of equijoin but does not require any particular partitioning of the operand relations. 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

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

As in the parallel associative join algorithm, the partitioning of R and S can be 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 run time based on the load of the system. The main difference with the parallel associative join algorithm is that partitioning of S is necessary and the result is produced at p nodes rather than at n S-nodes.

The algorithm can be divided into two main phases, a *build* phase and a *probe* phase [DeWitt and Gerber, 1985]. The build phase hashes R 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 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 14.4. Figure 14.15 shows the application of the parallel hash join algorithm with $m = n = 2$. We assumed 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. ♦

As is common, each parallel join algorithm applies and dominates under different conditions. Join processing is achieved with a degree of parallelism of either n or p.

Algorithm 14.3: PHJ Algorithm

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**
    $R_{ij} \leftarrow$ apply $h(A)$ to R_i $(j = 1, \ldots, p)$; {hash R on A)} ;
    send R_{ij} to node j
  for *j from* 1 *to p in parallel* **do**
    $R_j \leftarrow \bigcup_{i=1}^{m} R_{ij}$ {receive from R-nodes}
  {Probe phase}
  for *i from* 1 *to n in parallel* **do**
    $S_{ij} \leftarrow$ apply $h(B)$ to S_i $(j = 1, \ldots, p)$; {hash S on B)} ;
    send S_{ij} to node j
  for *j from* 1 *to p in parallel* **do** {perform the join at each of the p nodes}
    $S_j \leftarrow \bigcup_{i=1}^{n} S_{ij}$; {receive from S-nodes} ;
    $T_j \leftarrow R_j \bowtie_{JP} S_j$
end

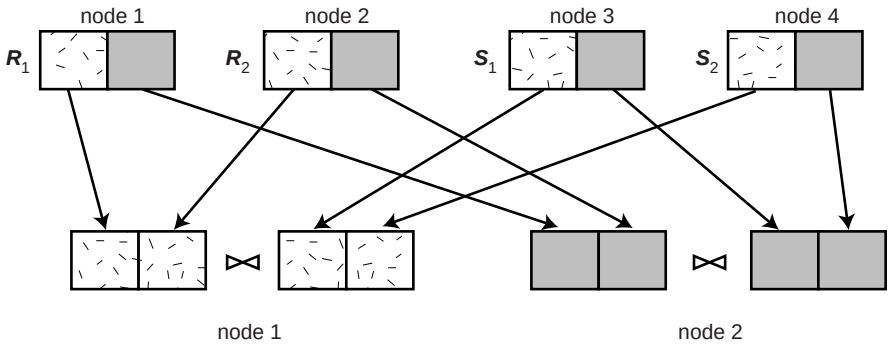

Fig. 14.15 Example of Parallel Hash Join

Since each algorithm requires moving at least one of the operand relations, a good indicator of their performance is total cost. To compare these algorithms, we now give a simple analysis of cost, 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. We assume that the local join algorithm is the same for all three parallel join algorithms. Finally, we assume that the amount of work done in parallel is uniformly distributed over all nodes allocated to the operator.

Without broadcasting capability, the parallel nested loop algorithm incurs a cost of $m*n$ messages, where a message contains a fragment of R of size $card(R)/m$ tuples. Thus we have

$$C_{COM}(PNL) = m*n*msg\left(\frac{card(R)}{m}\right)$$

Each of the S-nodes must join all of R with its S fragments. Thus we have

$$C_{PRO}(PNL) = n*C_{LOC}(card(R), card(S)/n)$$

The parallel associative join algorithm requires that each R-node partitions a fragment of R into n subsets of size $card(R)/(m*n)$ and sends them to n S-nodes. Thus we have

$$C_{COM}(PAJ) = m*n*msg\left(\frac{card(R)}{m*n}\right)$$

and

$$C_{PRO}(PAJ) = n*C_{LOC}(card(R)/n, card(S)/n)$$

The parallel hash join algorithm requires that both relations R and S be partitioned across p nodes in a way similar to the parallel associative join algorithm. Thus we have

$$C_{COM}(PHJ) = m*p*msg\left(\frac{card(R)}{m*p}\right) + n*p*msg\left(\frac{card(S)}{n*p}\right)$$

and

$$C_{PRO}(PHJ) = n*C_{LOC}(card(R)/n, card(S)/n)$$

Let us first assume that $p = n$. In this case, the join processing cost for the PAJ and PHJ algorithms is identical. However, it is higher for the PNL algorithm, because each S-node must perform the join with R entirely. From the equations above, it is clear that the PAJ algorithm incurs the least communication cost. However, the comparison of communication cost between the PNL and PHJ algorithms depends on the values of relation cardinality and degree of partitioning. If we choose $p < n$, the PHJ algorithm

incurs the least communication cost but at the expense of increased join processing cost. For example, if $p = 1$, the join is processed in a purely centralized way.

In conclusion, the PAJ algorithm is most likely to dominate and should be used when applicable. Otherwise, the choice between the PNL and PHJ algorithms requires the estimation of their total cost with the optimal value for p. The choice of a parallel join algorithm can be summarized by the procedure CHOOSE_JA shown in Algorithm 14.4, where the profile of a relation indicates whether it is partitioned and on which attribute.

Algorithm 14.4: CHOOSE_JA

```
Input: prof(R): profile of relation R ;
prof(S): profile of relation S ;
JP: join predicate
Output: JA: join algorithm
begin
    if JP is equijoin then
        if one relation is partitioned according to the join attribute then
            JA ← PAJ
        else
            if Cost(PNL) < Cost(PHJ) then
                JA ← PNL
            else
                JA ← PHJ
    else
        JA ← PNL
end
```

14.3.3 Parallel Query Optimization

Parallel query optimization exhibits similarities with distributed query processing. However, it focuses much more on taking advantage of both intra-operator parallelism (using the algorithms described above) and inter-operator parallelism. As any query optimizer (see Chapter 8), a parallel query optimizer can be seen as three components: a search space, a cost model, and a search strategy. In this section, we describe the techniques for these components.

14.3.3.1 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 is stored. For example, in the parallel hash join algorithm (see Algorithm 14.3), in the build phase, a hash table is constructed in 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 non-overlapping sub-trees, 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 14.5. Figure 14.16 shows two execution trees, one with no pipeline and one with pipeline. Pipelining a relation is indicated by an arrow with larger head. Figure 14.16(a) shows an execution without pipeline. 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, (3) probe it with R_3 and produce the result. Figure 14.16(b) 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 14.17 shows four operator trees that represent execution plans for a three-way join. Large-head arrows indicate that the input relation is consumed in pipeline, i.e., is not locally stored. 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 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, long right-deep trees are more efficient then corresponding left-deep

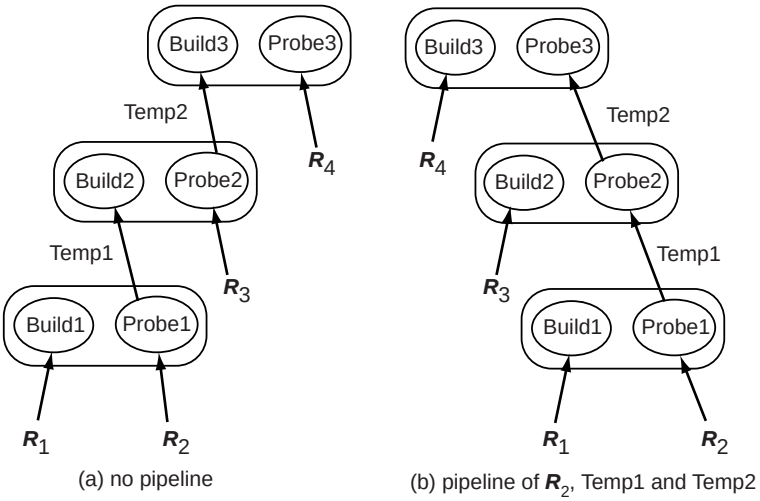

Fig. 14.16 Two hash-join trees with a different scheduling.

trees but tend to consume more memory to store left-hand side relations. In a left-deep tree such as that of Figure 14.17(a), 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 (Figure 14.17(d)) 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 Figure 14.17(d) are partitioned such that (R_1 and R_2) have the same home h_1 and (R_3 and $R4$ have the same home h_2), disjoint from 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 [Ziane et al., 1993]. 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.

14.3.3.2 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 [Lanzelotte et al., 1994]. The architecture-independent part is constituted

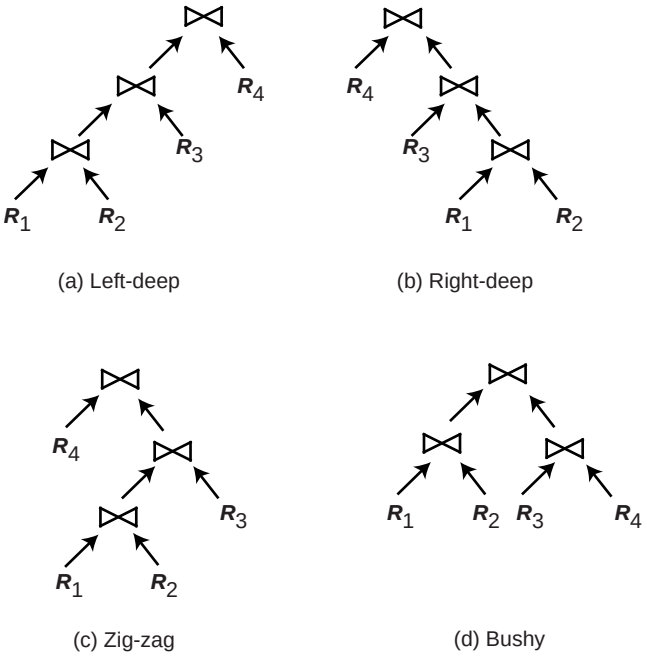

Fig. 14.17 Execution Plans as Operator Trees

of 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 inter-operator 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, it can be assumed 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 [Lanzelotte et al., 1994]:

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

where Op denotes an operator and $respTime(Op)$ 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 O 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 result 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.

14.3.3.3 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 (see Section 8.1.2) generally outperform deterministic strategies in parallel query optimization.

14.4 Load Balancing

Good load balancing is crucial for the performance of a parallel system. As noted in Chapter 8 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 transactions and queries among 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 inter-operator levels. In this section, we discuss these parallel execution problems and their solutions.

14.4.1 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 first step is generally sequential. It includes process (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 speedup reachable during the execution of an operator and to deduce the optimal number of processors [Wilshut and Apers, 1992]. 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_0 to allocate and the maximal achievable speedup (S_0).

$$n_0 = \sqrt{\frac{c * N}{a}} \qquad S_0 = \frac{n_0}{2}$$

The optimal number of processors (n_0) 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.

Interferences.

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 bus of a shared-memory system. When the number of processors is increased, the number of conflicts on the bus 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 Chapter 11).

However, shared variables may well become the bottleneck of query execution, creating hot spots and convoy effects [Blasgen et al., 1979]. 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. Studies have shown the overhead of such

strategy: 45% of the query execution time was consumed by interference among 16 processors.

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 appear with intra-operator parallelism (variation in partition size), namely *data skew*, and inter-operator parallelism (variation in the complexity of operators).

The effects of skewed data distribution on a parallel execution can be classified as follows [Walton et al., 1991]. *Attribute value skew (AVS)* is skew inherent in the dataset (e.g., there are more citizens in Paris than in Waterloo) while *tuple placement skew (TPS)* is the skew introduced when the data are 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 14.18 illustrates this classification on a query over two relations R and S that are poorly partitioned. 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).

14.4.2 Intra-Operator Load Balancing

Good intra-operator load balancing depends on the degree of parallelism and the allocation of processors for the operator. For some algorithms, e.g., the parallel hash join algorithm, 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 a 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

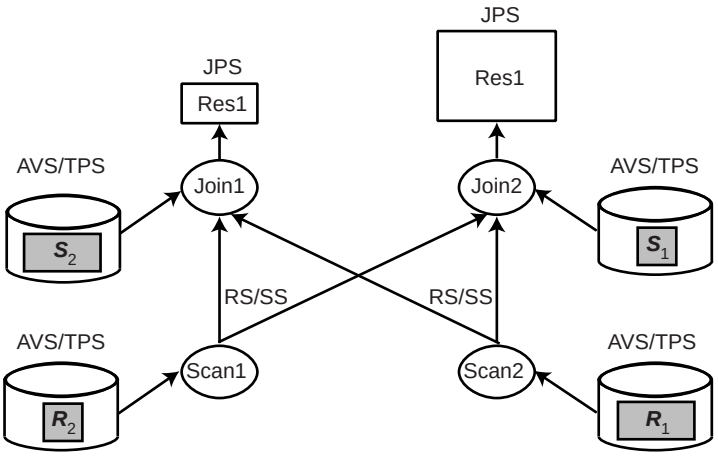

Fig. 14.18 Data skew example

techniques that can be incorporated in a hybrid query optimizer. We describe below these techniques in the context of parallel joins, 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 [Kitsuregawa and Ogawa, 1990; Omiecinski, 1991]. This approach is generalized to allow for more dynamic adjustment of the degree of parallelism [Biscondi et al., 1996]. It uses specific *control operators* in the execution plan to detect whether the static estimates for intermediate result sizes will differ from the run-time values. During execution, if the difference between tje 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 intra-operator 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 [Bergsten et al., 1991; Dageville et al., 1994] 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 intra-operator parallelism [Bouganim et al., 1996a,b].

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 [DeWitt et al., 1992]. 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 [DeWitt et al., 1992]. A similar approach is to modify the join algorithms by inserting a scheduling step that is in charge of redistributing the load at runtime [Wolf et al., 1993].

14.4.3 Inter-Operator Load Balancing

In order to obtain good load balancing at the inter-operator 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 inter-operator communication. This is harder to achieve in shared-nothing for the following reasons [Wilshut et al., 1995]. 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 [Mehta and DeWitt, 1995]. 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 [Rahm and Marek, 1995; Garofalakis and Ioannidis, 1996].

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 [Lu et al., 1991; Shekita et al., 1993]. In particular, a processor can be allocated all the operators in the same pipeline chain, thus, with no inter-operator parallelism. However, inter-operator parallelism is useful for executing independent pipeline chains. The approach proposed by Hong [1992] 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 intra-operator parallelism of the tasks in order to reach maximum resource utilization.

14.4.4 Intra-Query Load Balancing

Intra-query load balancing must combine intra- and inter-operator parallelism. To some extent, given a parallel architecture, the techniques for either intra- or inter-operator load balancing we just presented can be combined. However, in the important context of hybrid systems such as NUMA or cluster, the problems of load balancing are exacerbated because they must be addressed at two levels, locally among the processors of each shared-memory node (SM-node) and globally among all nodes. None of the approaches for intra- and inter-operator 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 in hybrid systems is the execution model called *Dynamic Processing (DP)* [Bouganim et al., 1996c]. 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 (intra-operator parallelism) and vertically (inter-operator parallelism) along the query operators. This minimizes the communication overhead of inter-node load balancing by maximizing intra and inter-operator 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: $O_1 < O_2$ indicates that operator O_1 cannot start before operator O_2.

Example 14.6. Figure 14.19 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 < Probe1
Build2 < Probe3
Build3 < Probe2

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

Heuristic1: Build1 < Scan2, Build3 < Scan4, Build2 < Scan3
Heuristic2: Build2 < Scan3

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 (Scan1) = i
home (Build1, Probe1, Scan2, Scan3) = j
home (Scan4) = Node C
home (Build2, Build3, Probe2, Probe3) = j and k

♦

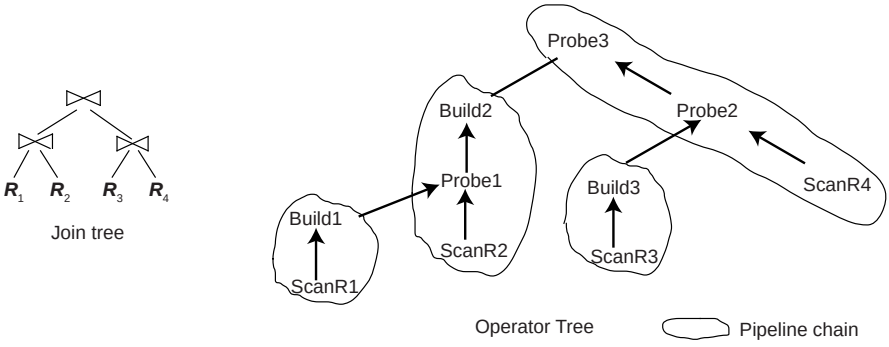

Fig. 14.19 A join tree and associated operator tree

Given such an operator tree, the problem is to produce an execution on a hybrid architecture that minimizes response time. This can be done by using a dynamic load balancing mechanism at two levels: (i) within a 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 intra-operator and inter-operator parallelism within a 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 a 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 $(Operator, Bucket)$ pair that references the scan operator and the base relation bucket to scan. A *data activation* describes a tuple produced in pipeline mode. It is represented by an $(Operator, Tuple, Bucket)$ 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 bucket'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*, also called *table queues* [Pirahesh et al., 1990], 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 Figure 14.20 with an execution snapshot for the operator tree of Figure 14.19.

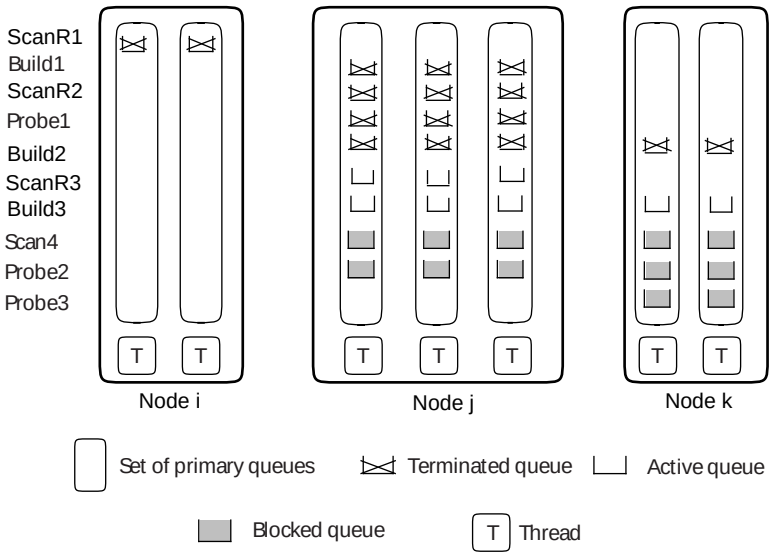

Fig. 14.20 Snapshot of an execution

Threads.

A simple strategy for obtaining good load balancing inside a 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, interference, and convoy problems [Pirahesh et al., 1990; Hong, 1992]. 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.

Load balancing within a 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 a SM-node starves, we can apply load sharing with another SM-node by acquiring some of its workload [Shatdal and Naughton, 1993]. 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, we need a mechanism that can dynamically estimate the benefit of acquiring activations and data.

Let us call "transactioner," which acquires work, the SM-node and "provider," which gets off-loaded by providing work to the transactioner, the SM-node. The problem is to select a queue to acquire activations and decide how much work to acquire. This is a dynamic optimization problem since there is a trade-off between the potential gain of off-loading the provider and the overhead of acquiring activations and data. This trade-off can be expressed by the following conditions: (i) the transactioner must be able to store in memory the activations and corresponding data; (ii) enough work must be acquired in order to amortize the overhead of acquisition; (iii) acquiring too much work should be avoided; (iv) only probe activations can be acquired since triggered activations require disk accesses and building activations require building hash tables locally; (v) there is no gain in moving activations associated with blocked operators that could not be processed anyway. Finally, to respect the decisions of the optimizer, a SM-node cannot execute activations of an operator that it does not own, i.e., the SM-node is not in the operator home.

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 non-blocking hash-join algorithms, which allows the concurrent execution of all the operators of the bushy tree [Wilshut et al., 1995]. 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 tradeoff.

Performance evaluation of DP with a 72-processor organized as a cluster of SM-nodes has shown that DP performs as well as a dedicated model in shared-memory and can scale up very well [Bouganim et al., 1996c].

14.5 Database Clusters

Clusters of PC servers are another form of parallel computer that provides a cost-effective alternative to supercomputers or tightly-coupled multiprocessors. For in-

stance, they have been used successfully in scientific computing, web information retrieval (e.g., Google search engine) and data warehousing. However, these applications are typically read-intensive, which makes it easier to exploit parallelism. In order to support update-intensive applications that are typical of business data processing, full parallel database capabilities, including transaction support, must be provided. This can be achieved using a parallel DBMS implemented over a cluster. In this case, all cluster nodes are homogeneous, under the full control of the parallel DBMS.

The parallel DBMS solution may be not viable for some businesses such as Application Service Providers (ASP). In the ASP model, customers' applications and databases (including data and DBMS) are hosted at the provider site and need to be available, typically through the Internet, as efficiently as if they were local to the customer site. A major requirement is that applications and databases remain autonomous, i.e., remain unchanged when moved to the provider site's cluster and under the control of the customers. Thus, preserving autonomy is critical to avoid the high costs and problems associated with application code modification. Using a parallel DBMS in this case is not appropriate as it is expensive, requires heavy migration to the parallel DBMS and hurts database autonomy.

A solution is to use a *database cluster*, which is a cluster of autonomous databases, each managed by an off-the-shelf DBMS [Röhm et al., 2000, 2001]. 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. In its simplest form, a database cluster can be viewed as a multidatabase system on a cluster. However, 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, query processing, and fault-tolerance. In this section, we present these techniques after introducing a database cluster architecture.

14.5.1 Database Cluster Architecture

As discussed in Section 14.1.3.4, a cluster can have a shared-disk or shared-nothing architecture. Shared-disk requires a special interconnect that provides a shared disk space to all nodes with provision for cache consistency. Shared-nothing can better support database autonomy without the additional cost of a special interconnect and can scale up to very large configurations. This explains why most of the work in database clusters has assumed a shared-nothing architecture. However, techniques designed for shared-nothing can be applied, perhaps in a simpler way, to shared-disk.

Figure 14.21 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 (e.g., at application servers) 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 Section 14.5.4 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 intra-query parallelism. With inter-query parallelism, the query processor routes each submitted query to one node and, after query completion, sends results to the client application. Intra-query 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 on-line recovery and failover.

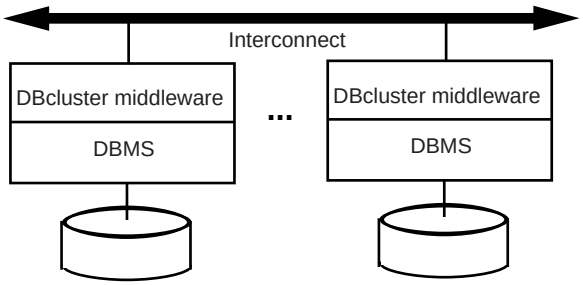

Fig. 14.21 A Database Cluster Shared-nothing Architecture

14.5.2 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). Unlike a distributed system, 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 [Chockler et al., 2001] that manages reliable communication between groups of nodes. Group communication primitives 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). The NODO protocol (see Chapter 13) is a representative of eager protocol that can be used in a database cluster. We present now another protocol for replication that is lazy and provides support for one-copy serializability and scalability.

Preventive replication protocol.

Preventive replication is a lazy protocol for lazy distributed replication in a database cluster [Pacitti et al., 2003; Coulon et al., 2005; Pacitti et al., 2006]. It also preserves DBMS autonomy. Instead of using total ordered multicast, as in eager protocols such as NODO, 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 [Pacitti et al., 1999; Pacitti and Simon, 2000; Pacitti et al., 2006].

We present the basic refreshment algorithm for updating copies, assuming full replication, for simplicity. The communication system is assumed to provide FIFO multicast [Pacitti et al., 2003]. Max is the upper bound of the time needed to multicast a message from a node i to any other node j. It is essential to have a value of Max that is not over estimated. The computation of Max resorts to scheduling theory [Pinedo, 2001] and takes into account several parameters such as the global reliable network itself, the characteristics of the messages to multicast and the failures to be tolerated. Each node has a local clock. For fairness, clocks are assumed to have a drift and to be ε-synchronized, i.e., the difference between any two correct clocks is not higher that

ε (known as the precision). Inconsistencies may arise whenever the serial orders of two transactions at two nodes are not equal. Therefore, they must be executed in the same serial order at any two nodes. Thus, global FIFO ordering is not sufficient to guarantee the correctness of the refreshment algorithm. Each transaction is associated with a chronological timestamp value C. The principle of the preventive refreshment algorithm is to submit a sequence of transactions in the same chronological order at each node. Before submitting a transaction at node i, it checks whether there is any older transaction en route to node i. To accomplish this, the submission time of a new transaction at node i is delayed by $Max + \varepsilon$. Thus the earliest time a transaction is submitted is $C + Max + \varepsilon$ (henceforth called the delivery_time).

Whenever a transaction T_i is to be triggered at some node i, node i multicasts T_i to all nodes $1, 2, ..., n$, including itself. Once T_i is received at some other node j (i may be equal to j), it is placed in the pending queue in FIFO order with respect to the triggering node i. Therefore, at each node i, there is a set of queues, $q_1, q_2, ..., q_n$, called pending queues, each of which corresponds to a node and is used by the refreshment algorithm to perform chronological ordering with respect to the delivery times. Figure 14.22 shows part of the components necessary to run the algorithm. The Refresher reads transactions from the top of pending queues and performs chronological ordering with respect to the delivery times. Once a transaction is ordered, then the refresher writes it to the running queue in FIFO order, one after the other. Finally the Deliverer keeps checking the top of the running queue to start transaction execution, one after the other, in the local DBMS.

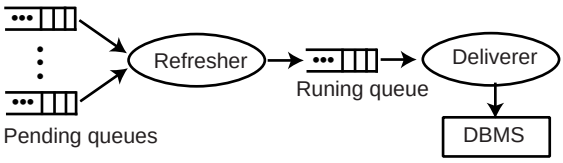

Fig. 14.22 Preventive Refreshment Architecture

Example 14.7. Let us illustrate the algorithm. Suppose we have two nodes i and j, masters of the copy R. So at node i, there are two pending queues: q_i and q_j corresponding to master nodes i and j. T_1 and T_2 are two transactions that update R at nodes i and j, respectively. Let us suppose that $Max = 10$ and $\varepsilon = 1$. So, at node i, we have the following sequence of execution:

- At time 10: T_2 arrives with a timestamp $ts(T_2) = 5$. So $q_i = [T_2(5)], q(j) = []$ and T_2 is chosen by the Refresher to be the next transaction to perform at delivery_time $16 (= 5 + 10 + 1)$, and the time is set to expire at time 16.
- At time 12: T_1 arrives from node j with a timestamp $ts(T_1) = 3$; so $q_i = [T_2(5)], q_j = [T_1(3)]$. T_1 is chosen by the Refresher to be the next transaction to perform at delivery_time $14 (= 3 + 10 + 1)$, and the time is re-set to expire at time 14.

- At time 14: the timeout expires and the Refresher writes T_1 into the running queue. Thus, $q_i = [T_2(5)], q(j) = []$. T_2 is selected to be the next transaction to perform at delivery_time $16 (= 5 + 10 + 1)$.
- At time 16: the timeout expires. The Refresher writes T_2 into the running queue. So $q_i = [], q(j) = []$.

Although the transactions are received in the wrong order with respect to their timestamps (T_2 then T_1), they are written into the running queue in chronological order according to their timestamps (T_1 then T_2). Thus, the total order is enforced even if messages are not sent in total order. ♦

The original preventive replication protocol has two limitations. First, it assumes that databases are fully replicated across all cluster nodes and thus propagates each transaction to each cluster node. This makes the algorithm unsuitable for supporting very large databases. Second, it has performance limitations since transactions are performed one after the other, and must endure waiting delays before starting. Thus, refreshment is a potential bottleneck, in particular, in the case of bursty workloads where the arrival rates of transactions are high at times.

The first limitation can be addressed by providing support for partial replication [Coulon et al., 2005]. With partial replication, some of the target nodes may not be able to perform a transaction T because they do not hold all the copies necessary to perform the read set of T. However the write set of T, which corresponds to its refresh transaction, must be ordered using T's timestamp value in order to ensure consistency. So T is scheduled as usual but not submitted for execution. Instead, the involved target nodes wait for the reception of the corresponding write set. Then, at origin node i, when the commitment of T is detected, the corresponding write set is produced and node i multicasts it towards the target nodes. Upon reception of the write set at a target node j, the content of T (still waiting) is replaced with the content of the incoming write set and T can be executed.

The second limitation is addressed by a refreshment algorithm that (potentially) eliminates the delay time [Pacitti et al., 2006]. In a cluster (which is typically fast and reliable), messages are often naturally chronologically ordered [Pedone and Schiper, 1998]. Only a few messages can be received in an order that is different than the sending order. Based on this property, the algorithm can be improved by submitting a transaction for execution as soon as it is received, thus avoiding the delay before submitting transactions. To guarantee strong consistency, the commit order of the transactions is scheduled in such a way that a transaction can be committed only after $Max + \varepsilon$. When a transaction T is received out-of-order, all younger transactions must be aborted and re-submitted according to their correct timestamp order with respect to T. Therefore, all transactions are committed in their timestamp order. To improve response time in bursty workloads, transactions can be triggered concurrently. Using the isolation property of the underlying DBMS, each node can guarantee that each transaction sees a consistent database at all times. To maintain strong consistency at all nodes, transactions are committed in the same order in which they are submitted and written to the running queue. Thus, total order is always enforced. However, without access to the DBMS concurrency controller (for autonomy reasons), one

cannot guarantee that two conflicting concurrent transactions obtain a lock in the same order at two different nodes. Therefore, conflicting transactions are not triggered concurrently. Detecting that two transactions are conflicting requires code analysis as for determining conflict classes in the NODO protocol. 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 [Pacitti et al., 2006].

14.5.3 Load Balancing

In a database cluster, replication offers good load balancing opportunities. With eager or preventive replication, query load balancing is easy to achieve. Since all copies are mutually consistent, any node that stores a copy of the transactioned data, e.g., the least loaded node, can be chosen at run-time 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.

Relaxed consistency models have been proposed for controlling replica divergence based on user requirements. User requirements on the desired consistency can be expressed by either the programmers, e.g., within SQL statements [Guo et al., 2004] or the database administrators, e.g., using access rules [Gançarski et al., 2002]. In most approaches, consistency reduces to freshness: update transactions are globally serialized over the different cluster nodes, so that whenever a query is sent to a given node, it reads a consistent state of the database. Global consistency is achieved by ensuring that conflicting transactions are executed at each node in the same relative order. However, the consistent state may not be the latest one, since transactions may be running at other nodes. The *data freshness* of a node reflects the difference between the database state of the node and the state it would have if all the running transactions had already been applied to that node. However, freshness is not easy to define, in particular for perfectly fresh database states. Thus, the opposite concept of *staleness*, is often used since it is always defined (e.g., equal to 0 for perfectly fresh database states). The staleness of a relation copy can then be captured by the quantity of change that has been made to the other copies, as measured by the number of tuples updated [Pape et al., 2004].

Example 14.8. Let us illustrate how lazy distributed replication can introduce staleness, and its impact on query answers. Consider the following query Q:

```
SELECT   PNO
FROM     ASG
WHERE    SUM(DUR) > 200
GROUP BY PNO
```

Let us assume that relation ASG is replicated at nodes i and j, both copies with a staleness of 0 at time t_0. Assume that, for the group of tuples where PNO="P1", we have SUM(DUR)=180. Consider that, at t_0+1, node i, respectively node j, commits a transaction that inserts a tuple for PNO="P1" with DUR=12, respectively DUR=18. Thus, the staleness of both i and j is 1. Now, at t_0+2, executing Q at either i or j would not retrieve "P1" since for the group of tuples where PNO="P1", we have SUM(DUR)=192 at i and 198 at j. The reason is that the two copies, although consistent, are stale. However, after reconciliation, e.g., at t_0+3, we have SUM(DUR)=210 at both nodes and executing Q would retrieve "P1". Thus, the accuracy of Q's answer depends on how much stale the node's copy is. ♦

With relaxed freshness, load balancing is more complex because the cost of copy reconciliation for enforcing user-defined freshness requirements must be considered when routing transactions and queries to cluster nodes. Röhm et al. [2002b] propose a simple solution for freshness-aware query routing in database clusters. Using single-master replication techniques (i.e., transactions are always routed to the master node), queries are routed to the least loaded node that is fresh enough. If no node is fresh enough, the query simply waits.

Gançarski et al. [2007] propose a more general solution to freshness-aware routing. It works with lazy distributed replication that yields the highest opportunities for transaction load balancing. We summarize this solution. A transaction router generates for each incoming transaction or query an execution plan based on user freshness requirements obtained from the shared directory. Then, it triggers execution at the best nodes, using run-time information on nodes' load. When necessary, it also triggers refresh transactions in order to make some nodes fresher for executing subsequent transactions or queries.

The transaction router takes into account the freshness requirements of queries at the relation level to improve load balancing. It uses cost functions that takes into account not only the cluster load in terms of concurrent transactions and queries, but also the estimated time to refresh replicas to the level required by incoming queries. The transaction router uses two cost-based routing strategies, each well-suited to different application needs. The first strategy, called cost-based only (CB), makes no assumption about the workload and assesses the synchronization cost to respect the staleness accepted by queries and transactions. CB simply evaluates, for each node, the cost of refreshing the node (if necessary) to meet the freshness requirements as well as the cost of executing the transaction itself. Then it chooses the node that minimizes the cost. The second strategy favors update transactions to deal with OLTP workloads. It is a variant of CB with bounded response time (BRT) that dynamically assigns nodes for transaction processing and nodes for query processing. It uses a parameter, $Tmax$, which represents the maximum response time users can accept for update transactions. It dedicates as many cluster nodes as necessary to ensure that updates are executed in less than $Tmax$, and uses the remaining nodes for processing queries. The validation of this approach, using implementation and emulation up to 128 nodes with the TPC-C benchmark, shows that excellent scale up can be obtained [Gançarski et al., 2007].

Other approaches have been proposed for load balancing in database clusters. The approach in [Milán-Franco et al., 2004] adjusts to changes in the load submitted to the different replicas and to the type of workload. It combines load-balancing with feedback-driven adjustments of the number of concurrent transactions. The approach is shown to provide high throughput, good scalability, and low response times for changing loads and workloads with little overhead.

14.5.4 Query Processing

In a database cluster, parallel query processing can be used successfully to yield high performance. Inter-query (or inter-transaction) 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 thoughput 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, intra-query parallelism is essential to further reduce response time. Intra-query 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 ressembles fragmentation and allocation design in distributed databases (see Chapter 3) except that the objective is to increase intra-query 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 is addressed by Stöhr et al. [2000], using small grain partitions. Under uniform data distribution, this solution is shown to yield good intra-query parallelism and outperform inter-query 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 intra-query parallelism is obtained by sending sub-queries to different virtual partitions [Akal et al., 2002]. 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 sub-query is forced to process a different subset of data items. Finally, the partitioned result needs to be combined by an aggregate query.

Example 14.9. Let us illustrate SVP with the following query Q:

```
SELECT   PNO, AVG(DUR)
FROM     ASG
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 ASG. Thus, the degree of intra-query parallelism is n. Furthermore, the "AVG(DUR)" operation must be rewriten 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. Akal et al. [2002] propose a classification of OLAP queries such that queries of the same class have similar parallelization properties. This classification relies on how the largest relations, called fact tables (e.g., Orders and LineItems) in a typical OLAP application, are accessed. The rationale is that such the virtual partitioning of such relations yields much intra-operator 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 incidentally 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 sub-queries instead of one per DBMS [Lima et al., 2004a]. Working with smaller sub-queries 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 [Lima et al., 2004b]. AVP runs independently at each participating cluster node, avoiding inter-node 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 sub-query with this interval.
3. Increase the partition size and execute the corresponding sub-query 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 6 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 by Röhm et al. [2000] uses physical partitioning for the largest and most important relations and fully replicates the small tables. Thus, intra-query parallelism can be achieved with lesser disk space requirements. The hybrid solution due to Furtado et al. [2005, 2006] 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.

14.5.5 Fault-tolerance

In the previous sections, the focus has been on how to attain consistency, performance and scalability when the system does not fail. 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, 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 either be 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 at the database cluster. 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 non-master 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 Section 13.1 that, in a

[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.

replicated database, one-copy serializability can 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.

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 transaction. 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 abortions due to failures [Perez-Sorrosal et al., 2006]. It has been applied to the NODO replication protocol (see Chapter 13) as follows. The write set and the transaction response for each update transaction are multicast to the other replicas before answering the client. Thus, any other replica can take over at any point in a transactional interaction.

The actions to be taken in the case of a master replica failure are more involved than for the non-master case. First, a new master should be appointed to take over the failed master. The appointment of a new master should be agreed upon 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). For instance, the NODO protocol handles failures in this way. When appointing a new master, it is necessary to take care of consistency.

Another essential aspect of fault-tolerance is recovery after failure. High availability has two faces. One is how 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. However, this *offline recovery* protocol hurts availability, which contradicts the initial goal of replication. Therefore, if high availability and performance should be provided, the only option is to perform *online recovery* [Kemme et al., 2001; Jiménez-Peris et al., 2002].

14.6 Conclusion

Parallel database systems strive to exploit multiprocessor architectures using software-oriented solutions for data management. Their promises are high-performance, high-availability, and extensibility with a good cost/performance ratio. Furthermore, paral-

lelism is the only viable solution for supporting very large databases within a single system.

Parallel database systems can be supported by various parallel architectures among shared-memory, shared-disk, shared-nothing and hybrid architectures. Each architecture has advantages and limitations in terms of performance, availability, and extensibility. For small configurations (e.g., less than 20 processors), shared-memory can provide the highest performance because of better load balancing. Shared-disk and shared-nothing architectures outperform shared-memory in terms of extensibility. Some years ago, shared-nothing was the only choice for high-end systems. However, recent progress in disk connectivity technologies such as SAN make shared-disk a viable alternative with the main advantage of simplifying data administration. Hybrid architectures such as NUMA and cluster can combine the efficiency and simplicity of shared-memory and the extensibility and cost of either shared disk or shared nothing. In particular, they can use shared-memory nodes with excellent performance/cost. Both NUMA and cluster can scale up to large configurations (hundred of nodes). The main advantage of NUMA over a cluster is the simple (shared-memory) programming model that eases database design and administration. However, using standard PC nodes and interconnects, clusters provide a better overall cost/performance ratio and, using shared-nothing, can scale up to very large configurations (thousands of nodes).

Parallel data management techniques extend distributed database techniques in order to obtain high-performance, high-availability, and extensibility. Essentially, the solutions for transaction management, i.e., distributed concurrency control, reliability, atomicity, and replication can be reused. However, the critical issues for such architectures are data placement, parallel query execution, parallel data processing, parallel query optimization and load balancing. The solutions to these issues are more involved than in distributed DBMS because the number of nodes may be much higher. Furthermore, parallel data management techniques use different assumptions such as fast interconnect and homogeneous nodes that provide more opportunities for optimization.

A database cluster is an important kind of parallel database system that uses 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, query processing, and fault-tolerance.

14.7 Bibliographic Notes

The earlier proposal of a database server or database machine is given in [Canaday et al., 1974]. Comprehensive surveys of parallel database systems are provided in [Graefe, 1993].

Parallel database system architectures are discussed in [Bergsten et al., 1993; Stonebraker, 1986], and compared using a simple simulation model in [Bhide and Stonebraker, 1988]. NUMA architectures are described in [Lenoski et al., 1992;

Goodman and Woest, 1988]. Their influence on query execution and performance can be found in [Bouganim et al., 1999] and [Dageville et al., 1994]. Examples of parallel database prototypes or products are described in [DeWitt et al., 1986; Tandem, 1987; Pirahesh et al., 1990; Graefe, 1990; Group, 1990; Bergsten et al., 1991; Hong, 1992], and [Apers et al., 1992]. Data placement in a parallel database server is treated in [Livny et al., 1987]. Parallel optimization studies appear in [Shekita et al., 1993], [Ziane et al., 1993], and [Lanzelotte et al., 1994].

Load balancing in parallel database systems have been extensively studied. [Walton et al., 1991] presents a taxonomy of intra-operator load balancing problems, namely, data skew. [DeWitt et al., 1992], [Kitsuregawa and Ogawa, 1990], [Shatdal and Naughton, 1993], [Wolf et al., 1993], [Rahm and Marek, 1995], [Mehta and DeWitt, 1995] and [Garofalakis and Ioannidis, 1996] present several aproaches for load balancing in shared-nothing architectures. [Omiecinski, 1991] and [Bouganim et al., 1996b] focus on shared-memory architectures while [Bouganim et al., 1996c] and [Bouganim et al., 1999] consider load balancing in the hybrid architecure context.

The concept of database cluster as a cluster of autonomous DBMS is defined in [Röhm et al., 2000]. Several protocols for scalable eager replication in database clusters using group communication are proposed in [Kemme and Alonso, 2000a,b; 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 Section 14.5.2 is based on [Pacitti et al., 2003; Coulon et al., 2005; Pacitti et al., 2006]. Most of the content of Section 14.5.3 on freshness-aware load balancing is based on [Gançarski et al., 2002; Pape et al., 2004; Gançarski et al., 2007]. Load balancing in database clusters is also addressed in [Milán-Franco et al., 2004]. The content of Section 14.5.5 on fault tolerance in database clusters is based on [Kemme et al., 2001; Jiménez-Peris et al., 2002; Perez-Sorrosal et al., 2006]. Query processing based on virtual partitioning has been first proposed in [Akal et al., 2002]. Combining physical and virtual partitioning is proposed in [Röhm et al., 2000]. Most of the content of Section 14.5.4 is based on the work on adaptive virtual partitioning [Lima et al., 2004a,b] and hybrid partitioning [Furtado et al., 2005, 2006].

Exercises

Problem 14.1 (*). Consider the centralized server organization with several application servers accessing one database server. Also assume that each application server stores a subset of the data directory that is fully stored on the database server. Assume also that the local data directories at different application servers are not necessarily disjoint. What are the implications on data directory management and query processing for the database server if the local data directories can be updated by the application servers rather than the database server?

Problem 14.2 (**). Propose an architecture for a parallel shared-memory database server and provide a qualitative comparison with shared-nothing architecture on the

basis of expected performance, software complexity (in particular, data placement and query processing), extensibility, and availability.

Problem 14.3. Specify the parallel hash join algorithm for the parallel shared-memory database server architecture proposed in Exercise 14.2.

Problem 14.4 (*). Explain the problems associated with clustering and full partitioning in a shared-nothing parallel database system. Propose several solutions and compare them.

Problem 14.5. Propose a parallel semijoin algorithm for a shared-nothing parallel database system. How should the parallel join algorithms be extended to exploit this semijoin algorithm?

Problem 14.6. 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 14.7. 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 14.8 (**). Propose a data placement strategy for a cluster architecture that maximizes *intra-node* parallelism (intra-operator parallelism within a shared-memory node).

Problem 14.9 (**). How should the DP execution model presented in Section 14.4.4 be changed to deal with inter-query parallelism?

Problem 14.10 (**). Consider a multi-user centralized database system. Describe the main change to allow inter-query 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 14.11 (**). Same question for intra-query parallelism on a shared-memory architecture or for a shared-nothing architecture.

Problem 14.12 (*). Consider the database cluster architecture in Figure 14.21. Assuming that each cluster node can accept incoming transactions, make precise the DBcluster 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 14.13 (**). Discuss the issues of fault-tolerance for the preventive replication protocol (see Section 14.5.2).

Problem 14.14 (**). Compare the preventive replication protocol with the NODO replication protocol (see Chapter 13) in the context of a cluster system in terms of: replication configurations supported, network requirements, consistency, performance, fault-tolerance.

Problem 14.15 (*). Let us consider a database cluster for an online store application. The database is concurrently accessed by short update transactions (e.g., product orders) and long read-only decision support queries (e.g., stock analysis). Discuss how database replication with freshness control can be useful in improving the response time of the decision support queries. What can be the impact on transaction load?

Problem 14.16 (**). 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 Partioning can be used to obtain intra-query parallelism and, if so, write the corresponding subquery and the final result composition query.

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

Chapter 15
Distributed Object Database Management

In this chapter, we relax another one of the fundamental assumptions we made in Chapter 1 — namely that the system implements the relational data model. Relational databases have proven to be very successful in supporting business data processing applications. However, there are many applications for which relational systems may not be appropriate. Examples include XML data management, computer-aided design (CAD), office information systems (OIS), document management systems, and multimedia information systems. For these applications, different data models and languages are more suitable. Object database management systems (object DBMSs) are better candidates for the development of some of these applications due to the following characteristics [Özsu et al., 1994b]:

1. These applications require explicit storage and manipulation of more abstract data types (e.g., images, design documents) and the ability for the users to define their own application-specific types. Therefore, a rich type system supporting user-defined abstract types is required. Relational systems deal with a single object type, a relation, whose attributes come from simple and fixed data type domains (e.g., numeric, character, string, date). There is no support for explicit definition and manipulation of application-specific types.
2. The relational model structures data in a relatively simple and flat manner. Representing structural application objects in the flat relational model results in the loss of natural structure that may be important to the application. For example, in engineering design applications, it may be preferable to explicitly represent that a vehicle object contains an engine object. Similarly, in a multimedia information system, it is important to note that a hyperdocument object contains a particular video object and a captioned text object. This "containment" relationship between application objects is not easy to represent in the relational model, but is fairly straightforward in object models by means of *composite objects* and *complex objects*, which we discuss shortly.
3. Relational systems provide a declarative and (arguably) simple language for accessing the data – SQL. Since this is not a computationally complete lan-

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_15,

guage, complex database applications have to be written in general programming languages with embedded query statements. This causes the well-known "impedance mismatch" [Copeland and Maier, 1984] problem, which arises because of the differences in the type systems of the relational languages and the programming languages with which they interact. The concepts and types of the query language, typically set-at-a-time, do not match with those of the programming language, which is typically record-at-a-time. This has resulted in the development of DBMS functions, such as cursor processing, that enable iterating over the sets of data objects retrieved by query languages. In an object system, complex database applications may be written entirely in a single object database programming language.

The main issue in object DBMSs is to improve application programmer productivity by overcoming the impedence mismatch problem with acceptable performance. It can be argued that the above requirements can be met by relational DBMSs, since one can possibly map them to relational data structures. In a strict sense this is true; however, from a modeling perspective, it makes little sense, since it forces programmers to map semantically richer and structurally complex objects that they deal with in the application domain to simple structures in representation.

Another alternative is to extend relational DBMSs with "object-oriented" functionality. This has been done, leading to "object-relational DBMS" [Stonebraker and Brown, 1999; Date and Darwen, 1998]. Many (not all) of the problems in object-relational DBMSs are similar to their counterparts in object DBMSs. Therefore, in this chapter we focus on the issues that need to be addressed in object DBMSs.

A careful study of the advanced applications mentioned above indicates that they are inherently distributed, and require distributed data management support. This gives rise to distributed object DBMSs, which is the subject of this chapter.

In Section 15.1, we provide the necessary background of the fundamental object concepts and issues in developing object models. In Section 15.2, we consider the distribution design of object databases. Section 15.3 is devoted to the discussion of the various distributed object DBMS architectural issues. In Section 15.4, we present the new issues that arise in the management of objects, and in Section 15.5 the focus is on object storage considerations. Sections 15.6 and 15.7 are devoted to fundamental DBMS functions: query processing and transaction management. These issues take interesting twists when considered within the context of this new technology; unfortunately, most of the existing work in these areas concentrate on non-distributed object DBMSs. We, therefore, provide a brief overview and some discussion of distribution issues.

We note that the focus in this chapter is on fundamental object DBMS technology. We do not discuss related issues such as Java Data Objects (JDO), the use of object models in XML work (in particular the DOM object interface), or Service Oriented Architectures (SOA) that use object technology. These require more elaborate treatment than we have room in this chapter.

15.1 Fundamental Object Concepts and Object Models

An object DBMS is a system that uses an "object" as the fundamental modeling and access primitive. There has been considerable discussion on the elements of an object DBMS [Atkinson et al., 1989; Stonebraker et al., 1990] as well as significant amount of work on defining an "object model". Although some have questioned whether it is feasible to define an object model, in the same sense as the relational model [Maier, 1989], a number of object models have been proposed. There are a number of features that are common to most model specifications, but the exact semantics of these features are different in each model. Some standard object model specifications have emerged as part of language standards, the most important of which is that developed by the Object Data Management Group (ODMG) that includes an object model (commonly referred to as the ODMG model), an Object Definition Language (ODL), and an Object Query Language (OQL)[1] [Cattell et al., 2000]. As an alternative, there has been a proposal for extending the relational model in SQL3 (now known as SQL:1999) [Melton, 2002]. There has also been a substantial amount of work on the foundations of object models [Abadi and Cardelli, 1996; Abiteboul and Beeri, 1995; Abiteboul and Kanellakis, 1998a]. In the remainder of this section, we will review some of the design issues and alternatives in defining an object model.

15.1.1 Object

As indicated above, all object DBMSs are built around the fundamental concept of an *object*. An object represents a real entity in the system that is being modeled. Most simply, it is represented as a tiple ⟨OID, state, interface⟩, in which OID is the object identifier, the corresponding state is some representation of the current state of the object, and the interface defines the behavior of the object. Let us consider these in turn.

Object identifier is an invariant property of an object which permanently distinguishes it logically and physically from all other objects, regardless of its state [Khoshafian and Copeland, 1986]. This enables referential object sharing [Khoshafian and Valduriez, 1987], which is the basis for supporting composite and complex (i.e., graph) structures (see Section 15.1.3). In some models, OID equality is the only comparison primitive; for other types of comparisons, the type definer is expected to specify the semantics of comparison. In other models, two objects are said to be *identical* if they have the same OID, and *equal* if they have the same state.

The *state* of an object is commonly defined as either an atomic value or a constructed value (e.g., tuple or set). Let D be the union of the system-defined domains

[1] The ODMG was an industrial consortium that completed its work on object data management standards in 2001 and disbanded. There are a number of systems now that conform to the developed standard listed here: http://www.barryandassociates.com/odmg-compliance.html.

(e.g., domain of integers) and of user-defined abstract data type (ADT) domains (e.g., domain of companies), let I be the domain of identifiers used to name objects, and let A be the domain of attribute names. A *value* is defined as follows:

1. An element of D is a value, called an *atomic value*.
2. $[a_1 : v_1, \ldots, a_n : v_n]$, in which a_i is an element of A and v_i is either a value or an element of I, is called a *tuple value*. [ ] is known as the tuple constructor.
3. $\{v_1, \ldots, v_n\}$, in which v_i is either a value or an element of I, is called a *set value*. { } is known as the set constructor.

These models consider object identifiers as values (similar to pointers in programming languages). Set and tuple are data constructors that we consider essential for database applications. Other constructors, such as list or array, could also be added to increase the modeling power.

Example 15.1. Consider the following objects:

(i_1, 231)
(i_2, S70)
(i_3, $\{i_6, i_{11}$)
(i_4, $\{1, 3, 5\}$)
(i_5, [LF: i_7, RF: i_8, LR: i_9, RR: i_10])

Objects i_1 and i_2 are atomic objects and i_3 and i_4 are constructed objects. i_3 is the OID of an object whose state consists of a set. The same is true of i_4. The difference between the two is that the state of i_4 consists of a set of values, while that of i_3 consists of a set of OIDs. Thus, object i_3 references other objects. By considering object identifiers (e.g., i_6) as values in the object model, arbitrarily complex objects may be constructed. Object i_5 has a tuple valued state consisting of four attributes (or instance variables), the values of each being another object. ♦

Contrary to values, objects support a well-defined update operation that changes the object state without changing the object identifier (i.e., the identity of the object), which is immutable. This is analogous to updates in imperative programming languages in which object identifier is implemented by main memory pointers. However, object identifier is more general than pointers in the sense that it persists following the program termination. Another implication of object identifier is that objects may be shared without incurring the problem of data redundancy. We will discuss this further in Section 15.1.3.

Example 15.2. Consider the following objects:

(i_1, Volvo)
(i_2, [name: John, mycar: i_1])
(i_3, [name: Mary, mycar: i_1])

John and Mary share the object denoted by i_1 (they both own Volvo cars). Changing the value of object i_1 from "Volvo" to "Chevrolet" is automatically seen by both objects i_2 and i_3. ♦

The above discussion captures the structural aspects of a model – the state is represented as a set of *instance variables* (or *attributes*) that are values. The behavioral aspects of the model are captured in *methods*, which define the allowable operations on these objects and are used to manipulate them. Methods represent the behavioral side of the model because they define the legal behaviors that the object can assume. A classical example is that of an elevator [Jones, 1979]. If the only two methods defined on an elevator object are "up" and "down", they together define the behavior of the elevator object: it can go up or down, but not sideways, for example.

The *interface* of an object consist of its properties. These properties include instance variables that reflect the state of the object, and the methods that define the operations that can be performed on this object. All instance variables and all methods of an object do not need to be visible to the "outside world". An object's *public interface* may consist of a subset of its instance variables and methods.

Some object models take a uniform and behavioral approach. In these models, the distinction between values and objects are eliminated and everything is an object, providing uniformity, and there is no differentiation between intance variables and methods – there are only methods (usually called behaviors) [Dayal, 1989; Özsu et al., 1995a].

An important distinction emerges from the foregoing discussion between relational model and object models. Relational databases deal with data values in a uniform fashion. Attribute values are the atoms with which structured values (tuples and relations) may be constructed. In a value-based data model, such as the relational model, data are identified by values. A relation is identified by a name, and a tuple is identified by a key, a combination of values. In object models, by contrast, data are identified by its OID. This distinction is crucial; modeling of relationships among data leads to data redundancy or the introduction of foreign keys in the relational model. The automatic management of foreign keys requires the support of integrity constraints (referential integrity).

Example 15.3. Consider Example 15.2. In the relational model, to achieve the same purpose, one would typically set the value of attribute `mycar` to "Volvo", which would require both tuples to be updated when it changes to "Chevrolet". To reduce redundancy, one can still represent i_1 as a tuple in another relation and reference it from i_1 and i_2 using foreign keys. Recall that this is the basis of 3NF and BCNF normalization. In this case, the elimination of redundancy requires, in the relational model, normalization of relations. However, i_1 may be a structured object whose representation in a normalized relation may be awkward. In this case, we cannot assign it as the value of the `mycar` attribute even if we accept the redundancy, since the relational model requires attribute values to be atomic. ♦

15.1.2 Types and Classes

The terms "type" and "class" have caused confusion as they have sometimes been used interchangeably and sometimes to mean different things. In this chapter, we will use the more common term "class" when we refer to the specific object model construct and the term "type" to refer to a domain of objects (e.g., integer, string).

A class is a template for a group of objects, thus defining a common type for these objects that conform to the template. In this case, we don't make a distinction between primitive system objects (i.e., values), structural (tuple or set) objects, and user-defined objects. A class describes the type of data by providing a domain of data with the same structure, as well as methods applicable to elements of that domain. The abstraction capability of classes, commonly referred to as *encapsulation*, hides the implementation details of the methods, which can be written in a general-purpose programming language. As indicated earlier, some (possibly proper) subset of its class structure and methods make up the publicly visible interface of objects that belong to that class.

Example 15.4. In this chapter, we will use an example that demonstrates the power of object models. We will model a car that consists of various parts (engine, bumpers, tires) and will store other information such as make, model, serial number, etc. In our examples, we will use an abstract syntax. ODMG ODL is considerably more powerful than the syntax we use, but it is also more complicated, which is not necessary to demonstrate the concepts. The type definition of `Car` can be as follows using this abstract syntax:

```
type Car
   attributes
     engine : Engine
     bumpers : {Bumper}
     tires : [lf: Tire, rf: Tire, lr: Tire, rr: Tire]
     make : Manufacturer
     model : String
     year : Date
     serial_no : String
     capacity : Integer
   methods
     age: Real
     replaceTire(place, tire)
```

The class definition specifies that `Car` has eight attributes and two method. Four of the attributes (model, year, serial_no, capacity) are value-based, while the others (engine, bumpers, tires and make) are object-based (i.e., have other objects as their values). Attribute bumpers is set valued (i.e., uses the set constructor), and attribute tires is tuple-valued where the left front (lf), right front (rf), left rear (lr) and right rear (rr) tires are individually identified. Incidentally, we follow a notation where the attributes are lower case and types are capitalized. Thus, `engine` is an attribute and `Engine` is a type in the system.

The method `age` takes the system date, and the `year` attribute value and calculates the date. However, since both of these arguments are internal to the object, they are not shown in the type definition, which is the interface for the user. By contrast, `replaceTire` method requires users to provide two external arguments: place (where the tire replacement was done), and tire (which tire was replaced). ♦

The interface data structure of a class may be arbitrarily complex or large. For example, `Car` class has an operation `age`, which takes today's date and the manufacturing date of a car and calculates its age; it may also have more complex operations that, for example, calculate a promotional price based on the time of year. Similarly, a long document with a complex internal structure may be defined as a class with operations specific to document manipulation.

A class has an *extent* that is the collection of all objects that conform to the class specification. In some cases, a class extent can be materialized and maintained, but this is not a requirement for all classes.

Classes provide two major advantages. First, the primitive types provided by the system can easily be extended with user-defined types. Since there are no inherent constraints on the notion of relational domain, such extensibility can be incorporated in the context of the relational model [Osborn and Heaven, 1986]. Second, class operations capture parts of the application programs that are more closely associated with data. Therefore, it becomes possible to model both data and operations at the same time. This does not imply, however, that operations are stored with the data; they may be stored in an operation library.

We end this section with the introduction of another concept, collection, that appears explicitly in some object models. A *collection* is a grouping of objects. In this sense, a class extent is a particular type of collection – one that gathers all objects that conform to a class. However, collections may be more general and may be based on user-defined predicates. The results of queries, for example, are collections of objects. Most object models do not have an explicit collection concept, but it can be argued that they are useful [Beeri, 1990], in particular since collections provide for a clear closure semantics of the query models and facilitate definition of user views. We will return to the relationship between classes and collections after we introduce subtyping and inheritance 15.1.4.

15.1.3 Composition (Aggregation)

In the examples we have discussed so far, some of the instance variables have been value-based (i.e., their domains are simple values), such as the `model` and `year` in Example 15.3, while others are object-based, such as the `make` attribute, whose domain is the set of objects that are of type `Manufacturer`. In this case, the `Car` type is a *composite type* and its instances are referred to as *composite objects*. Composition is one of the most powerful features of object models. It allows sharing of objects, commonly referred to as *referential sharing*, since objects "refer" to each other by their OIDs as values of object-based attributes.

Example 15.5. Let us revise Example 15.3 as follows. Assume that c_1 is one instance of `Car` type that is defined in Example 15.3. If the following is true:

(i_2, [name: John, mycar: c_1])
(i_3, [name: Mary, mycar: c_1])

then this indicates that John and Mary own the same car. ♦

A restriction on composite objects results in *complex objects*. The difference between a composite and a complex object is that the former allows referential sharing while the latter does not[2]. For example, `Car` type may have an attribute whose domain is type `Tire`. It is not natural for two instances of type `Car`, c_1 and c_2, to refer to the same set of instances of `Tire`, since one would not expect in real life for tires to be used on multiple vehicles at the same time. This distinction between composite and complex objects is not always made, but it is an important one.

The composite object relationship between types can be represented by a *composition (aggregation) graph* (or *composition (aggregation) hierarchy* in the case of complex objects). There is an edge from instance variable I of type T_1 to type T_2 if the domain of I is T_2. The composition graphs give rise to a number of issues that we will discuss in the upcoming sections.

15.1.4 Subclassing and Inheritance

Object systems provide extensibility by allowing user-defined classes to be defined and managed by the system. This is accomplished in two ways: by the definition of classes using type constructors or by the definition of classes based on existing classes through the process of *subclassing*[3]. Subclassing is based on the *specialization* relationship among classes (or types that they define). A class `A` is a *specialization* of another class `B` if its interface is a superset of `B`'s interface. Thus, a specialized class is more defined (or more specified) than the class from which it is specialized. A class may be a specialization of a number of classes; it is explicitly specified as a *subclass* of a subset of them. Some object models require that a class is specified as a subclass of only one class, in which case the model supports *single subclassing*; others allow *multiple subclassing*, where a class may be specified as a subclass of more than one class. Subclassing and specialization indicate an **is-a** relationship between classes (types). In the above example, `A` **is-a** `B`, resulting in *substitutability*: an instance of a subclass (`A`) can be substituted in place of an instance of any of its *superclasses* (`B`) in any expression.

[2] This distinction between composite and complex objects is not always made, and the term "composite object" is used to refer to both. Some authors reverse the definition between composite and complex objects. We will use the terms as defined here consistently in this chapter.

[3] This is also referred to as *subtyping*. We use the term "subclassing" to be consistent with our use of terminology. However, recall from Section 15.1.2 that each class defines a type; hence the term "subtyping" is also appropriate.

If multiple subclassing is supported, the class system forms a semilattice that can be represented as a graph. In many cases, there is a single root of the class system, which is the least specified class. However, multiple roots are possible, as in C++ [Stroustrup, 1986], resulting in a class system with multiple graphs. If only single subclasssing is allowed, as in Smalltalk [Goldberg and Robson, 1983], the class system is a tree. Some systems also define a most specified type, which forms the bottom of a full lattice. In these graphs/trees, there is an edge from type (class) `A` to type (class) `B` if `A` is a subtype of `B`.

A class structure establishes the database schema in object databases. It enables one to model the common properties and differences among types in a concise manner.

Declaring a class to be a subclass of another results in *inheritance*. If class `A` is a subclass of `B`, then its its properties consist of the properties that it natively defines as well as the properties that it inherits from `B`. Inheritance allows reuse. A subclass may inherit either the behavior (interface) of its superclass, or its implementation, or both. We talk of single inheritance and multiple inheritance based on the subclass relationship between the types.

Example 15.6. Consider the `Car` type we defined earlier. A car can be modeled as a special type of Vehicle. Thus, it is possible to define `Car` as a subtype of `Vehicle` whose other subtypes may be `Motorcycle`, `Truck`, and `Bus`. In this case, `Vehicle` would define the common properties of all of these:

```
type Vehicle as Object
   attributes
     engine : Engine
     make : Manufacturer
     model : String
     year : Date
     serial_no : String
   methods
     age: Real
```

`Vehicle` is defined as a subclass of `Object` that we assume is the root of the class lattice with common methods such as `Put` or `Store`. Vehicle is defined with five attributes and one method that takes the `date` of manufacture and today's date (both of which are of system-defined type `Date`) and returns a real value. Obviously, `Vehicle` is a generalization of `Car` that we defined in Example 15.3. `Car` can now be defined as follows:

```
type Car as Vehicle
  attributes
    bumpers : {Bumper}
    tires : [LF: Tire, RF: Tire, LR: Tire, RR: Tire]
    capacity : Integer
```

Even though `Car` is defined with only two attributes, its interface is the same as the definition given in Example 15.3. This is because `Car` **is-a** `Vehicle`, and therefore inherits the attributes and methods of `Vehicle`. ♦

Subclassing and inheritance allows us to discuss an issue related to classes and collections. As we defined in Section 15.1.2, each class extent is a collection of objects that conform to that class definition. With subclassing, we need to be careful – the class extent consists of the objects that immediately conform to its definition, which is referred to as (*shallow extent*), along with the extensions of its subtypes (*deep extent*). For example in Example 15.6, the extent of `Vehicle` class consists of all vehicle objects (shallow extent) as well as all car objects (deep extent of `Vehicle`). One consequence of this is that the objects in the extent of a class are homogeneous with respect to subclassing and inheritance – they are all of the superclass's type. In contrast, a user-defined collection may be heterogeneous in that it can contain objects of types unrelated by subclassing.

15.2 Object Distribution Design

Recall from Chapter 3 that the two important aspects of distribution design are fragmentation and allocation. In this section we consider the analogue, in object databases, of the distribution design problem.

Distribution design in the object world brings new complexities due to the encapsulation of methods together with object state. An object is defined by its state and its methods. We can fragment the state, the method definitions, and the method implementation. Furthermore, the objects in a class extent can also be fragmented and placed at different sites. Each of these raise interesting problems and issues. For example, if fragmentation is performed only on state, are the methods duplicated with each fragment, or can one fragment methods as well? The location of objects with respect to their class definition becomes an issue, as does the type of attributes (instance variables). As discussed in Section 15.1.3, the domain of some attributes may be other classes. Thus, the fragmentation of classes with respect to such an attribute may have effects on other classes. Finally, if method definitions are fragmented as well, it is necessary to distinguish between simple methods and complex methods. Simple methods are those that do not invoke other methods, while complex ones can invoke methods of other classes.

Similar to the relational case, there are three fundamental types of fragmentation: horizontal, vertical, and hybrid [Karlapalem et al., 1994]. In addition to these two fundamental cases, derived horizontal partitioning , associated horizontal partitioning , and path partitioning have been defined [Karlapalem and Li, 1995]. Derived horizontal partitioning has similar semantics to its counterpart in relational databases, which we will discuss further in Section 15.2.1. Associated horizontal partitioning, is similar to derived horizontal partitioning except that there is no "predicate clause", like minterm predicate, constraining the object instances. Path partitioning is discussed in Section 15.2.3. In the remainder, for simplicity, we assume a class-based object model that does not distinguish between types and classes.

15.2.1 Horizontal Class Partitioning

There are analogies between horizontal fragmentation of object databases and their relational counterparts. It is possible to identify primary horizontal fragmentation in the object database case identically to the relational case. Derived fragmentation shows some differences, however. In object databases, derived horizontal fragmentation can occur in a number of ways:

1. Partitioning of a class arising from the fragmentation of its subclasses. This occurs when a more specialized class is fragmented, so the results of this fragmentation should be reflected in the more general case. Clearly, care must be taken here, because fragmentation according to one subclass may conflict with those imposed by other subclasses. Because of this dependence, one starts with the fragmentation of the most specialized class and moves up the class lattice, reflecting its effects on the superclasses.
2. The fragmentation of a complex attribute may affect the fragmentation of its containing class.
3. Fragmenation of a class based on a method invocation sequence from one class to another may need to be reflected in the design. This happens in the case of complex methods as defined above.

Let us start the discussion with the simplest case: namely, fragmentation of a class with simple attributes and methods. In this case, primary horizontal partitioning can be performed according to a predicate defined on attributes of the class. Partitioning is easy: given class C for partitioning, we create classes $C_1, \ldots, C_n$, each of which takes the instances of C that satisfy the particular partitioning predicate. If these predicates are mutually exclusive, then classes $C_1, \ldots, C_n$ are disjoint. In this case, it is possible to define $C_1, \ldots, C_n$ as subclasses of C and change C's definition to an *abstract class* – one that does not have an explicit extent (i.e., no instances of its own). Even though this significantly forces the definition of subtyping (since the subclasses are not any more specifically defined than their superclass), it is allowed in many systems.

A complication arises if the partitioning predicates are not mutually exclusive. There are no clean solutions in this case. Some object models allow each object to belong to multiple classes. If this is an option, it can be used to address the problem. Otherwise, "overlap classes" need to be defined to hold objects that satisfy multiple predicates.

Example 15.7. Consider the definition of the `Engine` class that is referred to in Example 15.6:

```
Class Engine as Object
   attributes
     no_cylinder : Integer
     capacity : Real
     horsepower: Integer
```

In this simple definition of Engine, all the attributes are simple. Consider the partitioning predicates

p_1: horsepower ≤ 150
p_2: horsepower > 150

In this case, `Engine` can be partitioned into two classes, `Engine1` and `Engine2`, which inherit all of their properties from the `Engine` class, which is redefined as an abstract class (i.e,. a class that cannot have any objects in its shallow extent). The objects of `Engine` class are distributed to the `Engine1` and `Engine2` classes based on the value of their horsepower attribute value. ♦

We should first note that this example points to a significant advantage of object models – we can explicitly state that methods in `Engine1` class mention only those with horsepower less-than-or-equal-to 150. Consequently, we are able to make distribution explicit (with state and behavior) that is not possible in the relational model.

This primary horizontal fragmentation of classes is applied to all classes in the system that are subject to fragmentation. At the end of this process, one obtains fragmentation schemes for every class. However, these schemes do not reflect the effect of derived fragmentation as a result of subclass fragmentation (as in the example above). Thus, the next step is to produce a set of derived fragments for each superclass using the set of predicates from the previous step. This essentially requires propagation of fragmentation decisions made in the subclasses to the superclasses. The output from this step is the set of primary fragments created in step two and the set of derived fragments from step three.

The final step is to combine these two sets of fragments in a consistent way. The final horizontal fragments of a class are composed of objects accessed by both applications running only on a class and those running on its subclasses. Therefore, we must determine the most appropriate primary fragment to merge with each derived fragment of every class. Several simple heuristics could be used, such as selecting the smallest or largest primary fragment, or the primary fragment that overlaps the most with the derived fragment. But, although these heuristics are simple and intuitive, they do not capture any quantitative information about the distributed object database. Therefore, a more precise approach would be based on an affinity measure between fragments. As a result, fragments are joined with those fragments with which they have the highest affinity.

Let us now consider horizontal partitioning of a class with object-based instance variables (i.e., the domain of some of its instance variables is another class), but all the methods are simple. In this case, the composition relationship between classes comes into effect. In a sense, the composition relationship establishes the owner-member relationship that we discussed in Chapter 3: If class C_1 has an attribute A_1 whose domain is class C_2, then C_1 is the owner and C_2 is the member. Thus, the decomposition of C_2 follows the same principles as derived horizontal partitioning, discussed in Chapter 3.

So far, we have considered fragmentation with respect to attributes only, because the methods were simple. Let us now consider complex methods; these require some

care. For example, consider the case where all the attributes are simple, but the methods are complex. In this case, fragmentation based on simple attributes can be performed as described above. However, for methods, it is necessary to determine, at compile time, the objects that are accessed by a method invocation. This can be accomplished with static analysis. Clearly, optimal performance will result if invoked methods are contained within the same fragment as the invoking method. Optimization requires locating objects accessed together in the same fragment because this maximizes local relevant access and minimizes local irrelevant accesses.

The most complex case is where a class has complex attributes and complex methods. In this case, the subtyping relationships, aggregation relationships and relationships of method invocations have to be considered. Thus, the fragmentation method is the union of all of the above. One goes through the classes multiple times, generating a number of fragments, and then uses an affinity-based method to merge them.

15.2.2 Vertical Class Partitioning

Vertical fragmentation is considerably more complicated. Given a class C, fragmenting it vertically into $C_1, \ldots, C_m$ produces a number of classes, each of which contains some of the attributes and some of the methods. Thus, each of the fragments is less defined than the original class. Issues that must be addressed include the subtyping relationship between the original class' superclasses and subclasses and the fragment classes, the relationship of the fragment classes among themselves, and the location of the methods. If all the methods are simple, then methods can be partitioned easily. However, when this is not the case, the location of these methods becomes a problem.

Adaptations of the affinity-based relational vertical fragmentation approaches have been developed for object databases [Ezeife and Barker, 1995, 1998]. However, the break-up of encapsulation during vertical fragmentation has created significant doubts as to the suitability of vertical fragmentation in object DBMSs.

15.2.3 Path Partitioning

The composition graph presents a representation for composite objects. For many applications, it is necessary to access the complete composite object. Path partitioning is a concept describing the clustering of all the objects forming a composite object into a partition. A path partition consists of grouping the objects of all the domain classes that correspond to all the instance variables in the subtree rooted at the composite object.

A path partition can be represented as a hierarchy of nodes forming a structural index. Each node of the index points to the objects of the domain class of the component object. The index thus contains the references to all the component

objects of a composite object, eliminating the need to traverse the class composition hierarchy. The instances of the structural index are a set of OIDs pointing to all the component objects of a composite class. The structural index is an orthogonal structure to the object database schema, in that it groups all the OIDs of component objects of a composite object as a structured index class.

15.2.4 Class Partitioning Algorithms

The main issue in class partitioning is to improve the performance of user queries and applications by reducing the irrelevant data access. Thus, class partitioning is a logical database design technique that restructures the object database schema based on the application semantics. It should be noted that class partitioning is more complicated than relation fragmentation, and is also NP-complete. The algorithms for class partitioning are based on affinity-based and cost-driven approaches.

15.2.4.1 Affinity-based Approach

As covered in Section 3.3.2, affinity among attributes is used to vertically fragment relations. Similarly, affinity among instance variables and methods, and affinity among multiple methods can be used for horizontal and vertical class partitioning. Horizontal and vertical class partitioning algorithms have been developed that are based on classifying instance variables and methods as being either simple or complex [Ezeife and Barker, 1995]. A complex instance variable is an object-based instance variable and is part of the class composition hierarchy. An alternative is a method-induced partitioning scheme, which applies the method semantics and appropriately generates fragments that match the methods data requirements [Karlapalem et al., 1996a].

15.2.4.2 Cost-Driven Approach

Though the affinity-based approach provides "intuitively" appealing partitioning schemes, it has been shown that these partitioning schemes do not always result in the greatest reduction of disk accesses required to process a set of applications [Florescu et al., 1997]. Therefore, a cost model for the number of disk accesses for processing both queries [Florescu et al., 1997] and methods [Fung et al., 1996] on an object oriented database has been developed. Further, an heuristic "hill-climbing" approach that uses both the affinity approach (for initial solution) and the cost-driven approach (for further refinement) has been proposed [Fung et al., 1996]. This work also develops structural join index hierarchies for complex object retrieval, and studies its effectiveness against pointer traversal and other approaches, such as join index hierarchies, multi-index and access support relations (see next section). Each

structural join index hierarchy is a materialization of path fragment, and facilitates direct access to a complex object and its component objects.

15.2.5 Allocation

The data allocation problem for object databases involves allocation of both methods and classes. The method allocation problem is tightly coupled to the class allocation problem because of encapsulation. Therefore, allocation of classes will imply allocation of methods to their corresponding home classes. But since applications on object-oriented databases invoke methods, the allocation of methods affects the performance of applications. However, allocation of methods that need to access multiple classes at different sites is a problem that has been not yet been tackled. Four alternatives can be identified [Fang et al., 1994]:

1. **Local behavior – local object.** This is the most straightforward case and is included to form the baseline case. The behavior, the object to which it is to be applied, and the arguments are all co-located. Therefore, no special mechanism is needed to handle this case.
2. **Local behavior – remote object.** This is one of the cases in which the behavior and the object to which it is applied are located at different sites. There are two ways of dealing with this case. One alternative is to move the remote object to the site where the behavior is located. The second is to ship the behavior implementation to the site where the object is located. This is possible if the receiver site can run the code.
3. **Remote behavior – local object.** This case is the reverse of case (2).
4. **Remote function – remote argument.** This case is the reverse of case (1).

Affinity-based algorithms for static allocation of class fragments that use a graph partitioning technique have also been proposed [Bhar and Barker, 1995]. However, these algorithms do not address method allocation and do not consider the interdependency between methods and classes. The issue has been addressed by means of an iterative solution for methods and class allocation [Bellatreche et al., 1998].

15.2.6 Replication

Replication adds a new dimension to the design problem. Individual objects, classes of objects, or collections of objects (or all) can be units of replication. Undoubtedly, the decision is at least partially object-model dependent. Whether or not type specifications are located at each site can also be considered a replication problem.

15.3 Architectural Issues

The preferred architectural model for object DBMSs has been client/server. We had discussed the advantages of these systems in Chapter 1. The design issues related to these systems are somewhat more complicated due to the characteristics of object models. The major concerns are listed below.

1. Since data and procedures are encapsulated as objects, the unit of communication between the clients and the server is an issue. The unit can be a page, an object, or a group of objects.
2. Closely related to the above issue is the design decision regarding the functions provided by the clients and the server. This is especially important since objects are not simply passive data, and it is necessary to consider the sites where object methods are executed.
3. In relational client/server systems, clients simply pass queries to the server, which executes them and returns the result tables to the client. This is referred to as *function shipping*. In object client/server DBMSs, this may not be the best approach, as the navigation of composite/complex object structures by the application program may dictate that data be moved to the clients (called *data shipping systems*). Since data are shared by many clients, the management of client cache buffers for data consistency becomes a serious concern. Client cache buffer management is closely related to concurrency control, since data that are cached to clients may be shared by multiple clients, and this has to be controlled. Most commercial object DBMSs use locking for concurrency control, so a fundamental architectural issue is the placement of locks, and whether or not the locks are cached to clients.
4. Since objects may be composite or complex, there may be possibilities for prefetching component objects when an object is requested. Relational client/server systems do not usually prefetch data from the server, but this may be a valid alternative in the case of object DBMSs.

These considerations require revisiting some of the issues common to all DBMSs, along with several new ones. We will consider these issues in three sections: those directly related to architectural design (architectural alternatives, buffer management, and cache consistency) are discussed in this section; those related to object management (object identifier management, pointer swizzling, and object migration) are discussed in Section 15.4; and storage management issues (object clustering and garbage collection) are considered in Section 15.5.

15.3.1 Alternative Client/Server Architectures

Two main types of client/server architectures have been proposed: object servers and page servers. The distinction is partly based on the granularity of data that are shipped between the clients and the servers, and partly on the functionality provided to the clients and servers.

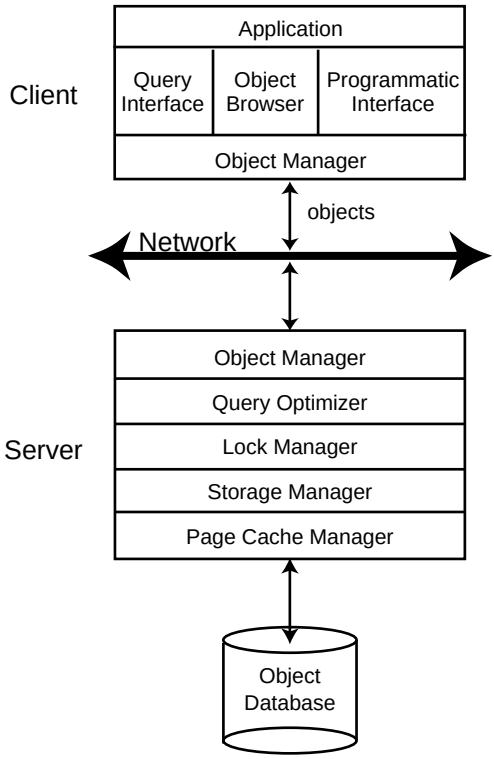

Fig. 15.1 Object Server Architecture

The first alternative is that clients request "objects" from the server, which retrieves them from the database and returns them to the requesting client. These systems are called *object servers* (Figure 15.1). In object servers, the server undertakes most of the DBMS services, with the client providing basically an execution environment for the applications, as well as some level of object management functionality (which will be discussed in Section 15.4). The object management layer is duplicated at both the client and the server in order to allow both to perform object functions. Object manager serves a number of functions. First and foremost, it provides a context for method execution. The replication of the object manager in both the server and the client enables methods to be executed at both the server and the clients. Executing methods in the client may invoke the execution of other methods,

which may not have been shipped to the server with the object. The optimization of method executions of this type is an important research problem. Object manager also deals with the implementation of the object identifier (logical, physical, or virtual) and the deletion of objects (either explicit deletion or garbage collection). At the server, it also provides support for object clustering and access methods. Finally, the object managers at the client and the server implement an object cache (in addition to the page cache at the server). Objects are cached at the client to improve system performance by localizing accesses. The client goes to the server only if the needed objects are not in its cache. The optimization of user queries and the synchronization of user transactions are all performed in the server, with the client receiving the resulting objects.

It is not necessary for servers in these architectures to send individual objects to the clients; if it is appropriate, they can send groups of objects. If the clients do not send any prefetching hints then the groups correspond to contiguous space on a disk page [Gerlhof and Kemper, 1994]. Otherwise, the groups can contain objects from different pages. Depending upon the group hit rate, the clients can dynamically either increase or decrease the group size [Liskov et al., 1996]. In these systems, one complication needs to be dealt with: clients return updated objects to clients. These objects have to be installed onto their corresponding data pages (called the *home page*). If the corresponding data page does not exist in the server buffer (such as, for example, if the server has already flushed it out), the server must perform an *installation read* to reload the home page for this object.

An alternative organization is a *page server* client/server architecture, in which the unit of transfer between the servers and the clients is a physical unit of data, such as a page or segment, rather than an object (Figure 15.2). Page server architectures split the object processing services between the clients and the servers. In fact, the servers do not deal with objects anymore, acting instead as "value-added" storage managers.

Early performance studies (e.g., [DeWitt et al., 1990]) favored page server architectures over object server architectures. In fact, these results have influenced an entire generation of research into the optimal design of page server-based object DBMSs. However, these results were not conclusive, since they indicated that page server architectures are better when there is a match between a data clustering pattern[4] and the users' access pattern, and that object server architectures are better when the users' data access pattern is not the same as the clustering pattern. These earlier studies were further limited in their consideration of only single client/single server and multiple client/single server environments. There is clearly a need for further study in this area before a final judgment may be reached.

Page servers simplify the DBMS code, since both the server and the client maintain page caches, and the representation of an object is the same all the way from the disk to the user interface. Thus, updates to the objects occur only in client caches and these updates are reflected on disk when the page is flushed from the client to

[4] Clustering is an issue we will discuss later in this chapter. Briefly, it refers to how objects are placed on physical disk pages. Because of composite and complex objects, this becomes an important issue in object DBMSs.

the server. Another advantage of page servers is their full exploitation of the client workstation power in executing queries and applications. Thus, there is less chance of the server becoming a bottleneck. The server performs a limited set of functions and can therefore serve a large number of clients. It is possible to design these systems such that the work distribution between the server and the clients can be determined by the query optimizer. Page servers can also exploit operating systems and even hardware functionality to deal with certain problems, such as pointer swizzling (see Section 15.4.2), since the unit of operation is uniformly a page.

Intuitively, there should be significant performance advantages in having the server understand the "object" concept. One is that the server can apply locking and logging functions to the objects, enabling more clients to access the same page. Of course, this is relevant for small objects less than a page in size.

The second advantage is the potential for savings in the amount of data transmitted to the clients by filtering them at the server, which is possible if the server can perform some of the operations. Note that the concern here is not the relative cost of sending one object versus one page, but that of filtering objects at the server and sending them versus sending all of the pages on which these objects may reside. This is indeed what the relational client/server systems do where the server is responsible for optimizing and executing the entire SQL query passed to it from a client. The situation is not as straightforward in object DBMSs, however, since the applications mix query access with object-by-object navigation. It is generally not a good idea to perform navigation at the server, since doing so would involve continuous interaction between the application and the server, resulting in a remote procedure call (RPC) for each object. In fact, the earlier studies were preferential towards page servers, since they mainly considered workloads involving heavy navigation from object to object.

One possibility of dealing with the navigation problem is to ship the user's application code to the server and execute it there as well. This is what is done in Web access, where the server simply serves as storage. Code shipping may be cheaper than data shipping. This requires significant care, however, since the user code cannot be considered safe and may threaten the safety and reliability of the DBMS. Some systems (e.g., Thor [Liskov et al., 1996]) use a safe language to overcome this problem. Furthermore, since the execution is now divided between the client and the server, data reside in both the server and the client cache, and its consistency becomes a concern. Nevertheless, the "function shipping" approach involving both the clients and the servers in the execution of a query/application must be considered to deal with mixed workloads. The distribution of execution between different machines must also be accommodated as systems move towards peer-to-peer architectures.

Clearly, both of these architectures have important advantages and limitations. There are systems that can shift from one architecture to the other – for example, O_2 would operate as a page server, but if the conflicts on pages increase, would shift to object shipping. Unfortunately, the existing performance studies do not establish clear tradeoffs, even though they provide interesting insights. The issue is complicated further by the fact that some objects, such as multimedia documents, may span multiple pages.

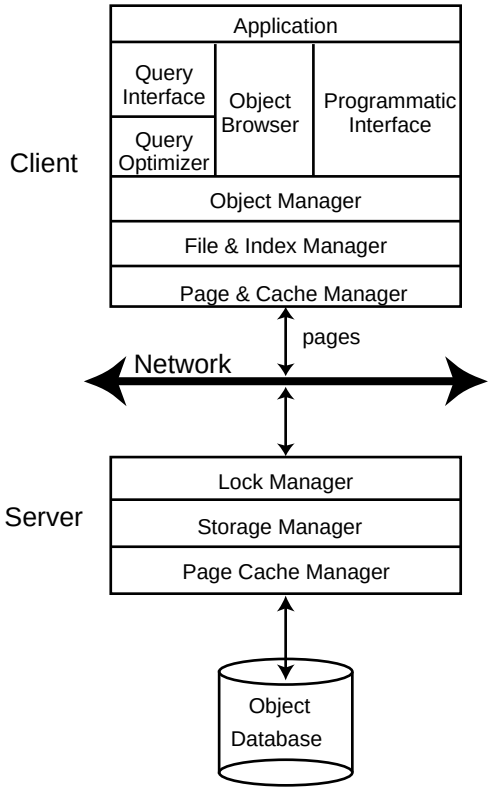

Fig. 15.2 Page Server Architecture

15.3.1.1 Client Buffer Management

The clients can manage either a page buffer, an object buffer, or a dual (i.e., page/object) buffer. If clients have a page buffer, then entire pages are read or written from the server every time a page fault occurs or a page is flushed. Object buffers can read/write individual objects and allow the applications object-by-object access.

Object buffers manage access at a finer granularity and, therefore, can achieve higher levels of concurrency. However, they may experience buffer fragmentation, as the buffer may not be able to accommodate an integral multiple of objects, thereby leaving some unused space. A page buffer does not encounter this problem, but if the data clustering on the disk does not match the application data access pattern, then the pages contain a great deal of unaccessed objects that use up valuable client buffer space. In these situations, buffer utilization of a page buffer will be lower than the buffer utilization of an object buffer.

To realize the benefits of both the page and the object buffers, dual page/object buffers have been proposed [Kemper and Kossmann, 1994; Castro et al., 1997]. In a

dual buffer system, the client loads pages into the page buffer. However, when the client flushes out a page, it retains the useful objects from the page by copying the objects into the object buffer. Therefore, the client buffer manager tries to retain well-clustered pages and isolated objects from non-well-clustered pages. The client buffer managers retain the pages and objects across the transaction boundaries (commonly referred to as *inter-transaction caching*). If the clients use a log-based recovery mechanism (see Chapter 12), they also manage an in-memory log buffer in addition to the data buffer. Whereas the data buffers are managed using a variation of the least recently used (LRU) policy, the log buffer typically uses a first-in/first-out buffer replacement policy. As in centralized DBMS buffer management, it is important to decide whether all client transactions at a site should share the cache, or whether each transaction should maintain its own private cache. The recent trend is for systems to have both shared and private buffers [Carey et al., 1994; Biliris and Panagos, 1995].

15.3.1.2 Server Buffer Management

The server buffer management issues in object client/server systems are not much different than their relational counterparts, since the servers usually manage a page buffer. We nevertheless discuss the issues here briefly in the interest of completeness. The pages from the page buffer are, in turn, sent to the clients to satisfy their data requests. A grouped object-server constructs its object groups by copying the necessary objects from the relevant server buffer pages, and sends the object group to the clients. In addition to the page level buffer, the servers can also maintain a modified object buffer (MOB) [Ghemawat, 1995]. A MOB stores objects that have been updated and returned by the clients. These updated objects have to be installed onto their corresponding data pages, which may require installation reads as described earlier. Finally, the modified page has to be written back to the disk. A MOB allows the server to amortize its disk I/O costs by batching the installation read and installation write operations.

In a client/server system, since the clients typically absorb most of the data requests (i.e., the system has a high cache hit rate), the server buffer usually behaves more as a staging buffer than a cache. This, in turn, has an impact on the selection of server buffer replacement policies. Since it is desirable to minimize the duplication of data in the client and the server buffers, the *LRU with hate hints* buffer replacement policy can be used by the server [Franklin et al., 1992]. The server marks the pages that also exist in client caches as *hated*. These pages are evicted first from the server buffer, and then the standard LRU buffer replacement policy is used for the remaining pages.

15.3.2 Cache Consistency

Cache consistency is a problem in any data shipping system that moves data to the clients. So the general framework of the issues discussed here also arise in relational client/server systems. However, the problems arise in unique ways in object DBMSs.

The study of DBMS cache consistency is very tightly coupled with the study of concurrency control (see Chapter 11), since cached data can be concurrently accessed by multiple clients, and locks can also be cached along with data at the clients. The DBMS cache consistency algorithms can be classified as avoidance-based or detection-based [Franklin et al., 1997]. *Avoidance-based algorithms* prevent the access to stale cache data[5] by ensuring that clients cannot update an object if it is being read by other clients. So they ensure that stale data never exists in client caches. *Detection-based algorithms* allow access of stale cache data, because clients can update objects that are being read by other clients. However, the detection-based algorithms perform a validation step at commit time to satisfy data consistency requirements.

Avoidance-based and detection-based algorithms can, in turn, be classified as *synchronous*, *asynchronous* or *deferred*, depending upon when they inform the server that a write operation is being performed. In synchronous algorithms, the client sends a lock escalation message at the time it wants to perform a write operation, and it blocks until the server responds. In asynchronous algorithms, the client sends a lock escalation message at the time of its write operation, but does not block waiting for a server response (it optimistically continues). In deferred algorithms, the client optimistically defers informing the server about its write operation until commit time. In deferred mode, the clients group all their lock escalation requests and send them together to the server at commit time. Thus, communication overhead is lower in a deferred cache consistency scheme, in comparison to synchronous and asynchronous algorithms.

The above classification results in a design space of possible algorithms covering six alternatives. Many performance studies have been conducted to assess the strengths and weaknesses of the various algorithms. In general, for data-caching systems, inter-transaction caching of data and locks is accepted as a performance enhancing optimization [Wilkinson and Neimat, 1990; Franklin and Carey, 1994], because this reduces the number of times a client has to communicate with the server. On the other hand, for most user workloads, invalidation of remote cache copies during updates is preferred over propagation of updated values to the remote client sites [Franklin and Carey, 1994]. Hybrid algorithms that dynamically perform either invalidation or update propagation have been proposed [Franklin and Carey, 1994]. Furthermore, the ability to switch between page and object level locks is generally considered to be better than strictly dealing with page level locks [Carey et al., 1997] because it increases the level of concurrency.

[5] An object in a client cache is considered to be *stale* if that object has already been updated and committed into the database by a different client.

We discuss each of the alternatives in the design space and comment on their performance characteristics.

- **Avoidance-based synchronous:** Callback-Read Locking (CBL) is the most common synchronous avoidance-based cache consistency algorithm [Franklin and Carey, 1994]. In this algorithm, the clients retain read locks across transactions, but they relinquish write locks at the end of the transaction. The clients send lock requests to the server and they block until the server responds. If the client requests a write lock on a page that is cached at other clients, the server issues callback messages requesting that the remote clients relinquish their read locks on the page. Callback-Read ensures a low abort rate and generally outperforms deferred avoidance-based, synchronous detection-based, and asynchronous detection-based algorithms.
- **Avoidance-based asynchronous:** Asynchronous avoidance-based cache consistency algorithms (AACC) [Özsu et al., 1998] do not have the message blocking overhead present in synchronous algorithms. Clients send lock escalation messages to the server and continue application processing. Normally, optimistic approaches such as this face high abort rates, which is reduced in avoidance-based algorithms by immediate server actions to invalidate stale cache objects at remote clients as soon as the system becomes aware of the update. Thus, asynchronous algorithms experience lower deadlock abort rates than deferred avoidance-based algorithms, which are discussed next.
- **Avoidance-based deferred:** Optimistic Two-Phase Locking (O2PL) family of cache consistency are deferred avoidance-based algorithms [Franklin and Carey, 1994]. In these algorithms, the clients batch their lock escalation requests and send them to the server at commit time. The server blocks the updating client if other clients are reading the updated objects. As the data contention level increases, O2PL algorithms are susceptible to higher deadlock abort rates than CBL algorithms.
- **Detection-based synchronous:** Caching Two-Phase Locking (C2PL) is a synchronous detection-based cache consistency algorithm [Carey et al., 1991]. In this algorithm, clients contact the server whenever they access a page in their cache to ensure that the page is not stale or being written to by other clients. C2PL's performance is generally worse than CBL and O2PL algorithms, since it does not cache read locks across transactions.
- **Detection-based asynchronous:** No-Wait Locking (NWL) with Notification is an asynchronous detection-based algorithm [Wang and Rowe, 1991]. In this algorithm, the clients send lock escalation requests to the server, but optimistically assume that their requests will be successful. After a client transaction commits, the server propagates the updated pages to all the other clients that have also cached the affected pages. It has been shown that CBL outperforms the NWL algorithm.
- **Detection-based deferred:** Adaptive Optimistic Concurrency Control (AOCC) is a deferred detection-based algorithm. It has been shown that AOCC can

outperform callback locking algorithms even while encountering a higher abort rate if the client transaction state (data and logs) completely fits into the client cache, and all application processing is strictly performed at the clients (purely data-shipping architecture) [Adya et al., 1995]. Since AOCC uses deferred messages, its messaging overhead is less than CBL. Furthermore, in a purely data-shipping client/server environment, the impact of an aborting client on the performance of other clients is quite minimal. These factors contribute to AOCC's superior performance.

15.4 Object Management

Object management includes tasks such as object identifier management, pointer swizzling, object migration, deletion of objects, method execution, and some storage management tasks at the server. In this section we will discuss some of these problems; those related to storage management are discussed in the next section.

15.4.1 Object Identifier Management

As indicated in Section 15.1, object identifiers (OIDs) are system-generated and used to uniquely identify every object (transient or persistent, system-created or user-created) in the system. Implementing the identity of persistent objects generally differs from implementing transient objects, since only the former must provide global uniqueness. In particular, transient object identity can be implemented more efficiently.

The implementation of persistent object identifier has two common solutions, based on either physical or logical identifiers, with their respective advantages and shortcomings. The physical identifier (POID) approach equates the OID with the physical address of the corresponding object. The address can be a disk page address and an offset from the base address in the page. The advantage is that the object can be obtained directly from the OID. The drawback is that all parent objects and indexes must be updated whenever an object is moved to a different page.

The logical identifier (LOID) approach consists of allocating a system-wide unique OID (i.e., a surrogate) per object. LOIDs can be generated either by using a system-wide unique counter (called pure LOID) or by concatenating a server identifier with a counter at each server (called pseudo-LOID). Since OIDs are invariant, there is no overhead due to object movement. This is achieved by an OID table associating each OID with the physical object address at the expense of one table look-up per object access. To avoid the overhead of OIDs for small objects that are not referentially shared, both approaches can consider the object value as their identifier. Object-oriented database systems tend to prefer the logical identifier approach, which better supports dynamic environments.

Implementing transient object identifier involves the techniques used in programming languages. As for persistent object identifier, identifiers can be physical or logical. The physical identifier can be the real or virtual address of the object, depending on whether virtual memory is provided. The physical identifier approach is the most efficient, but requires that objects do not move. The logical identifier approach, promoted by object-oriented programming, treats objects uniformly through an indirection table local to the program execution. This table associates a logical identifier, called an *object oriented pointer* (OOP) in Smalltalk, to the physical identifier of the object. Object movement is provided at the expense of one table look-up per object access.

The dilemma for an object manager is a trade-off between generality and efficiency. For example, supporting object-sharing explicitly requires the implementation of object identifiers for all objects within the object manager and maintaining the sharing relationship. However, object identifiers for small objects can make the OID table quite large. If object sharing is not supported at the object manager level, but left to the higher levels of system (e.g., the compiler of the database language), more efficiency may be gained. Object identifier management is closely related to object storage techniques, which we will discuss in Section 15.5.

In distributed object DBMSs, it is more appropriate to use LOIDs, since operations such as reclustering, migration, replication and fragmentation occur frequently. The use of LOIDs raises the following distribution related issues:

- **LOID Generation:** LOIDs must be unique within the scope of the entire distributed domain. It is relatively easy to ensure uniqueness if the LOIDs are generated at a central site. However, a centralized LOID generation scheme is not desirable because of the network latency overhead and the load on the LOID generation site. In multi-server environments, each server site generates LOIDs for the objects stored at that site. The uniqueness of the LOID is ensured by incorporating the server identifier as part of the LOID. Therefore, the LOID consists of both a server identifier part and a sequence number. The sequence number is the logical representation of the disk location of the object and is unique within a particular server. Sequence numbers are usually not reused to prevent anomalies: an object o_i is deleted, and its sequence number is subsequently assigned to a newly created object o_j, but existing references to o_i now point to the new object o_j, which is not intended.
- **LOID Mapping Location and Data Structures:** The location of the LOID-to-POID mapping information is important. If pure LOIDs are used, and if a client can be directly connected to multiple servers simultaneously, then the LOID-to-POID mapping information must be present at the client. If pseudo-LOIDs are used, the mapping information needs to be present only at the server. The presence of the mapping information at the client is not desirable, because this solution is not scalable (i.e.,the mapping information has to be updated at all the clients that might access the object).
  The LOID-to-POID mapping information is usually stored in hash tables or in B+ trees. There are advantages and disadvantages to both [Eickler et al.,

1995]. Hash tables provide fast access, but are not scalable as the database size increases. B^+-trees are scalable, but have logarithmic access time, and require complex concurrency control and recovery strategies. B^+-trees also support range queries, facilitating easy access to a collection of objects.

15.4.2 Pointer Swizzling

In object systems, one can navigate from one object to another using *path expressions* that involve attributes with object-based values. For example, if object `c` is of type `Car`, then `c.engine.manufacturer.name` is a path expression[6]. These are basically pointers. Usually on disk, object identifiers are used to represent these pointers. However, in memory, it is desirable to use in-memory pointers for navigating from one object to another. The process of converting a disk version of the pointer to an in-memory version of a pointer is known as "pointer-swizzling". Hardware-based and software-based schemes are two types of pointer-swizzling mechanisms [White and DeWitt, 1992]. In hardware-based schemes, the operating system's page-fault mechanism is used; when a page is brought into memory, all the pointers in it are swizzled, and they point to reserved virtual memory frames. The data pages corresponding to these reserved virtual frames are only loaded into memory when an access is made to these pages. The page access, in turn, generates an operating system page-fault, which must be trapped and processed. In software-based schemes, an object table is used for pointer-swizzling purposes so that a pointer is swizzled to point to a location in the object table – that is LOIDs are used. There are eager and lazy variations to the software-based schemes, depending upon when exactly the pointer is swizzled. Therefore, every object access has a level of indirection associated with it. The advantage of the hardware-based scheme is that it leads to better performance when repeatedly traversing a particular object hierarchy, due to the absence of a level of indirection for each object access. However, in bad clustering situations where only a few objects per page are accessed, the high overhead of the page-fault handling mechanism makes hardware-based schemes unattractive. Hardware-based schemes also do not prevent client applications from accessing deleted objects on a page. Moreover, in badly clustered situations, hardware-based schemes can exhaust the virtual memory address space, because page frames are aggressively reserved regardless of whether the objects in the page are actually accessed. Finally, since the hardware-based scheme is implicitly page-oriented, it is difficult to provide object-level concurrency control, buffer management, data transfer and recovery features. In many cases, it is desirable to manipulate data at the object level rather than the page level.

[6] We assume that `Engine` class is defined with at least one attribute, `manufacturer`, whose domain is the extent of class `Manufacturer`. `Manufacturer` class has an attribute called `name`.

15.4.3 Object Migration

One aspect of distributed systems is that objects move, from time to time, between sites. This raises a number of issues. First is the unit of migration. It is possible to move the object's state without moving its methods. The application of methods to an object requires the invocation of remote procedures. This issue was discussed above under object distribution. Even if individual objects are units of migration [Dollimore et al., 1994], their relocation may move them away from their type specifications and one has to decide whether types are duplicated at every site where instances reside or the types are accessed remotely when behaviors or methods are applied to objects. Three alternatives can be considered for the migration of classes (types):

1. the source code is moved and recompiled at the destination,
2. the compiled version of a class is migrated just like any other object, or
3. the source code of the class definition is moved, but not its compiled operations, for which a lazy migration strategy is used.

Another issue is that the movements of the objects must be tracked so that they can be found in their new locations. A common way of tracking objects is to leave *surrogates* [Hwang, 1987; Liskov et al., 1994], or *proxy objects* [Dickman, 1994]. These are place-holder objects left at the previous site of the object, pointing to its new location. Accesses to the proxy objects are directed transparently by the system to the objects themselves at the new sites. The migration of objects can be accomplished based on their current state [Dollimore et al., 1994]. Objects can be in one of four states:

1. Ready: Ready objects are not currently invoked, or have not received a message, but are ready to be invoked to receive a message.
2. Active: Active objects are currently involved in an activity in response to an invocation or a message.
3. Waiting: Waiting objects have invoked (or have sent a message to) another object and are waiting for a response.
4. Suspended: Suspended objects are temporarily unavailable for invocation.

Objects in active or waiting state are not allowed to migrate, since the activity they are currently involved in would be broken. The migration involves two steps:

1. shipping the object from the source to the destination, and
2. creating a proxy at the source, replacing the original object.

Two related issues must also be addressed here. One relates to the maintenance of the system directory. As objects move, the system directory must be updated to reflect the new location. This may be done lazily, whenever a surrogate or proxy

object redirects an invocation, rather than eagerly, at the time of the movement. The second issue is that, in a highly dynamic environment where objects move frequently, the surrogate or proxy chains may become quite long. It is useful for the system to transparently compact these chains from time to time. However, the result of compaction must be reflected in the directory, and it may not be possible to accomplish that lazily.

Another important migration issue arises with respect to the movement of composite objects. The shipping of a composite object may involve shipping other objects referenced by the composite object. An alternative method of dealing with this is a method called *object assembly* that we will consider under query processing in Section 15.6.3.

15.5 Distributed Object Storage

Among the many issues related to object storage, two are particularly relevant in a distributed system: object clustering and distributed garbage collection. Composite and complex objects provide opportunities, as we mentioned earlier, for clustering data on disk such that the I/O cost of retrieving them is reduced. Garbage collection is a problem that arises in object databases due to reference-based sharing. Indeed, in many object DBMSs, the only way to delete an object is to delete all references to it. Thus, object deletion and subsequent storage reclamation are critical and require special care.

15.5.0.1 Object Clustering

An object model is essentially conceptual, and should provide high physical data independence to increase programmer productivity. The mapping of this conceptual model to a physical storage is a classical database problem. As indicated in Section 15.1, in the case of object DBMSs, two kinds of relationships exist between types: subtyping and composition. By providing a good approximation of object access, these relationships are essential to guide the physical clustering of persistent objects. Object clustering refers to the grouping of objects in physical containers (i.e., disk extents) according to common properties, such as the same value of an attribute or sub-objects of the same object. Thus, fast access to clustered objects can be obtained.

Object clustering is difficult for two reasons. First, it is not orthogonal to object identifier implementation (i.e, LOID vs. POID). LOIDs incur more overhead (an indirection table), but enable vertical partitioning of classes. POIDs yield more efficient direct object access, but require each object to contain all inherited attributes. Second, the clustering of complex objects along the composition relationship is more involved because of object sharing (objects with multiple parents). In this case, the

use of POIDs may incur high update overhead as component objects are deleted or change ownership.

Given a class graph, there are three basic storage models for object clustering [Valduriez et al., 1986]:

1. The *decomposition storage model* (DSM) partitions each object class into binary relations (OID, attribute) and therefore relies on logical OID. The advantage of DSM is simplicity.
2. The *normalized storage model* (NSM) stores each class as a separate relation. It can be used with logical or physical OID. However, only logical OID allows the vertical partitioning of objects along the inheritance relationship [Kim et al., 1987].
3. The *direct storage model* (DSM) enables multi-class clustering of complex objects based on the composition relationship. This model generalizes the techniques of hierarchical and network databases, and works best with physical OID [Benzaken and Delobel, 1990]. It can capture object access locality and is therefore potentially superior when access patterns are well-known. The major difficulty, however, is to clustering an object whose parent has been deleted.

In a distributed system, both DSM and NSM are straightforward using horizontal partitioning. Goblin [Kersten et al., 1994] implements DSM as a basis for a distributed object DBMS with large main memory. DSM provides flexibility, and its performance disadvantage is compensated by the use of large main memory and caching. Eos [Gruber and Amsaleg, 1994] implements the direct storage model in a distributed single-level store architecture, where each object has a physical, system-wide OID. The Eos grouping mechanism is based on the concept of most relevant composition links and solves the problem of multiparent shared objects. When an object moves to a different node, it gets a new OID. To avoid the indirection of forwarders, references to the object are subsequently changed as part of the garbage collection process without any overhead. The grouping mechanism is dynamic to achieve load balancing and cope with the evolutions of the object graph.

15.5.0.2 Distributed Garbage Collection

An advantage of object-based systems is that objects can refer to other objects using object identifier. As programs modify objects and remove references, a persistent object may become unreachable from the persistent roots of the system when there is no more reference to it. Such an object is “garbage” and should be de-allocated by the garbage collector. In relational DBMSs, there is no need for automatic garbage collection, since object references are supported by join values. However, cascading updates as specified by referential integrity constraints are a simple form of “manual”

garbage collection. In more general operating system or programming language contexts, manual garbage collection is typically error-prone. Therefore, the generality of distributed object-based systems calls for automatic distributed garbage collection.

The basic garbage collection algorithms can be categorized as *reference counting* or tracing-based. In a reference counting system, each object has an associated count of the references to it. Each time a program creates an additional reference that points to an object, the object's count is incremented. When an existing reference to an object is destroyed, the corresponding count is decremented. The memory occupied by an object can be reclaimed when the object's count drops to zero and become unreachable (at which time, the object is garbage). In reference counting, a problem can arise where two objects only refer to each other but not referred to by anyone else; in this case, the two objects are basically unreachable (except from each other) but their reference count has not dropped to zero.

Tracing-based collectors are divided into *mark and sweep* and *copy-based* algorithms. *Mark and sweep* collectors are two-phase algorithms. The first phase, called the "mark" phase, starts from the root and marks every reachable object (for example, by setting a bit associated to each object). This mark is also called a "color", and the collector is said to color the objects it reaches. The mark bit can be embedded in the objects themselves or in *color maps* that record, for every memory page, the colors of the objects stored in that page. Once all live objects are marked, the memory is examined and unmarked objects are reclaimed. This is the "sweep" phase.

Copy-based collectors divide memory into two disjoint areas called *from-space* and *to-space*. Programs manipulate from-space objects, while the to-space is left empty. Instead of marking and sweeping, copying collectors copy (usually in a depth first manner) the from-space objects reachable from the root into the to-space. Once all live objects have been copied, the collection is over, the contents of the from-space are discarded, and the roles of from- and to-spaces are exchanged. The copying process copies objects linearly in the to-space, which compacts memory.

The basic implementations of mark and sweep and copy-based algorithms are "stop-the-world"; i.e., user programs are suspended during the whole collection cycle. For many applications, however, stop-the-world algorithms cannot be used because of their disruptive behavior. Preserving the response time of user applications requires the use of incremental techniques. Incremental collectors must address problems raised by concurrency. The main difficulty with incremental garbage collection is that, while the collector is tracing the object graph, program activity may change other parts of the object graph. Garbage collection algorithms typically avoid the cases where the collector may miss tracing some reachable objects, due to concurrent changes to other parts of the object graph, and may erroneously reclaim them. On the other hand, although not desirable, it is acceptable to miss reclaiming a garbage and believe that it is alive.

Designing a garbage collection algorithm for object DBMSs is very complex. These systems have several features that pose additional problems for incremental garbage collection, beyond those typically addressed by solutions for non-persistent systems. These problems include the ones raised by the resilience to system failures and the semantics of transactions, and, in particular, by the rollbacks of partially

completed transactions, by traditional client-server performance optimizations (such as client caching and flexible management of client buffers), and by the huge volume of data to analyze in order to detect garbage objects. There have been a number of proposals starting with [Butler, 1987]. More recent work has investigated fault-tolerant garbage collection techniques for transactional persistent systems in centralized [Kolodner and Weihl, 1993; O'Toole et al., 1993] and client-server [Yong et al., 1994; Amsaleg, 1995; Amsaleg et al., 1995] architectures.

Distributed garbage collection, however, is even harder than centralized garbage collection. For scalability and efficiency reasons, a garbage collector for a distributed system combines independent per-site collectors with a global inter-site collector. Coordinating local and global collections is difficult because it requires carefully keeping track of reference exchanges between sites. Keeping track of such exchanges is necessary because an object may be referenced from several sites. In addition, an object located at one site may be referenced from live objects at remote sites, but not by any local live object. Such an object must not be reclaimed by the local collector, since it is reachable from the root of a remote site. It is difficult to keep track of inter-site references in a distributed environment where messages can be lost, duplicated or delayed, or where individual sites may crash.

Distributed garbage collectors typically rely either on distributed reference counting or distributed tracing. Distributed reference counting is problematic for two reasons. First, reference counting cannot collect unreachable cycles of garbage objects (i.e., mutually-referential garbage objects). Second, reference counting is defeated by common message failures; that is, if messages are not delivered reliably in their causal order, then maintaining the reference counting invariant (i.e., equality of the count with the actual number of references) is problematic. However, several algorithms propose distributed garbage collection solutions based on reference counting [Bevan, 1987; Dickman, 1991]. Each solution makes specific assumptions about the failure model, and is therefore incomplete. A variant of a reference counting collection scheme, called "reference listing" [Plainfossé and Shapiro, 1995], is implemented in Thor [Maheshwari and Liskov, 1994]. This algorithm tolerates server and client failures, but does not address the problem of reclaiming distributed cycles of garbage.

Distributed tracing usually combines independent per-site collectors with a global inter-site collector. The main problem with distributed tracing is synchronizing the distributed (global) garbage detection phase with independent (local) garbage reclamation phases. When local collectors and user programs all operate in parallel, enforcing a global, consistent view of the object graph is impossible, especially in an environment where messages are not received instantaneously, and where communications failures are likely. Therefore, distributed tracing-based garbage collection relies on inconsistent information in order to decide if an object is garbage or not. This inconsistent information makes distributed tracing collector very complex, because the collector tries to accurately track the minimal set of reachable objects to at least eventually reclaim some objects that really are garbage. Ladin and Liskov [1992] propose an algorithm that computes, on a central space, the global graph of remote references. Ferreira and Shapiro [1994] present an algorithm that can reclaim

cycles of garbage that span several disjoint object spaces. Finally, Fessant et al. [1998] present a complete (i.e., both acyclic and cyclic), asynchronous, distributed garbage collector.

15.6 Object Query Processing

Relational DBMSs have benefitted from the early definition of a precise and formal query model and a set of universally-accepted algebraic primitives. Although object models were not initially defined with a full complement of a query language, there is now a declarative query facility, OQL [Cattell et al., 2000], defined as part of the ODMG standard. In the remainder, we use OQL as the basis of our discussion. As we did earlier with SQL, we will take liberties with the language syntax.

Although there has been significant amount of work on object query processing and optimization, these have primarily focused on centralized systems. Almost all object query processors and optimizers that have been proposed to date use techniques developed for relational systems. Consequently, it is possible to claim that distributed object query processing and optimization techniques require the extension of centralized object query processing and optimization with the distribution approaches we discussed in Chapters 7 and 8. In this section, we will provide a brief review of the object query processing and optimization issues and approaches; the extension we refer to remains an open issue.

Although most object query processing proposals are based on their relational counterparts, there are a number of issues that make query processing and optimization more difficult in object DBMSs [Özsu and Blakeley, 1994]:

1. Relational query languages operate on very simple type systems consisting of a single type: relation. The closure property of relational languages implies that each relational operator takes one or two relations as operands and generates a relation as a result. In contrast, object systems have richer type systems. The results of object algebra operators are usually sets of objects (or collections), which may be of different types. If the object languages are closed under the algebra operators, these heterogeneous sets of objects can be operands to other operators. This requires the development of elaborate type inferencing schemes to determine which methods can be applied to **all** the objects in such a set. Furthermore, object algebras often operate on semantically different collection types (e.g., set, bag, list), which imposes additional requirements on the type inferencing schemes to determine the type of the results of operations on collections of different types.
2. Relational query optimization depends on knowledge of the physical storage of data (access paths) that is readily available to the query optimizer. The encapsulation of methods with the data upon which they operate in object DBMSs raises at least two important issues. First, determining (or estimating) the cost of executing methods is considerably more difficult than calculating

or estimating the cost of accessing an attribute according to an access path. In fact, optimizers have to worry about optimizing method execution, which is not an easy problem because methods may be written using a general-purpose programming language and the evaluation of a particular method may involve some heavy computation (e.g., comparing two DNA sequences). Second, encapsulation raises issues related to the accessibility of storage information by the query optimizer. Some systems overcome this difficulty by treating the query optimizer as a special application that can break encapsulation and access information directly [Cluet and Delobel, 1992]. Others propose a mechanism whereby objects "reveal" their costs as part of their interface [Graefe and Maier, 1988].

3. Objects can (and usually do) have complex structures whereby the state of an object references another object. Accessing such complex objects involves *path expressions*. The optimization of path expressions is a difficult and central issue in object query languages. Furthermore, objects belong to types related through inheritance hierarchies. Optimizing the access to objects through their inheritance hierarchies is also a problem that distinguishes object-oriented from relational query processing.

Object query processing and optimization has been the subject of significant research activity. Unfortunately, most of this work has not been extended to distributed object systems. Therefore, in the remainder of this chapter, we will restrict ourselves to a summary of the important issues: object query processing architectures (Section 15.6.1), object query optimization (Section 15.6.2), and query execution strategies (Section 15.6.3).

15.6.1 Object Query Processor Architectures

As indicated in Chapter 6, query optimization can be modeled as an optimization problem whose solution is the choice, based on a *cost function*, of the "optimum" *state*, which corresponds to an algebraic query, in a *search space* that represents a family of equivalent algebraic queries. Query processors differ, architecturally, according to how they model these components.

Many existing object DBMS optimizers are either implemented as part of the object manager on top of a storage system, or as client modules in a client/server architecture. In most cases, the above-mentioned components are "hardwired" into the query optimizer. Given that extensibility is a major goal of object DBMSs, one would hope to develop an extensible optimizer that accommodates different search strategies, algebra specifications (with their different transformation rules), and cost functions. Rule-based query optimizers [Freytag, 1987; Graefe and DeWitt, 1987] provide some amount of extensibility by allowing the definition of new transformation rules. However, they do not allow extensibility in other dimensions.

It is possible to make the query optimizer extensible with respect to algebraic operators, logical transformation rules, execution algorithms, implementation rules (i.e., logical operator-to-execution algorithm mappings), cost estimation functions, and physical property enforcement functions (e.g., presence of objects in memory). This can be achieved by means of modularization that separates of a number of concerns [Blakeley et al., 1993]. For example, the user query language parsing structures can be separated from the operator graph on which the optimizer operates, allowing the replacement of the user language (i.e., using something other than OQL at the top) or making changes to the optimizer without modifying the parse structures. Similarly, the algebraic operator manipulation (logical optimization, or re-writing) can be separated from the execution algorithms, allowing exploration with alternative methods for implementing algebraic operators. These are extensions that may be achieved by means of well-considered modularization and structuring of the optimizer.

An approach to providing search space extensibility is to consider it as a group of *regions* where each region corresponds to an equivalent family of query expressions that are reachable from each other [Mitchell et al., 1993]. The regions are not necessarily mutually exclusive and differ in the queries they manipulate, the control (search) strategies they use, the query transformation rules they incorporate (e.g., one region may cover transformation rules dealing with simple select queries, while another region may deal with transformations for nested queries), and the optimization objectives they achieve (e.g., one region may have the objective of minimizing a cost function, while another region may attempt to transform queries to some desirable form).

The ultimate extensibility can be achieved by using object-oriented approach to develop the query processor and optimizer. In this case, everything (queries, classes, operators, operator implementations,, meta-information, etc) are all first-class objects [Peters et al., 1993]. The search space, the search strategy and the cost function are modeled as objects. Consequently, using object-oriented techniques, it is easy to add new operators, new re-write rules, or new operator implementations [Özsu et al., 1995b; Lanzelotte and Valduriez, 1991].

15.6.2 Query Processing Issues

As indicated earlier, query processing methodology in object DBMSs is similar to its relational counterpart, but with differences in details as a result of the object model and query model characteristics. In this section we will highlight these differences as they apply to algebraic optimization. We will also discuss a particular problem unique to object query models — namely, the execution of path expressions.