5.2 Distributed Concurrency Control

As noted above, a concurrency control algorithm enforces a particular isolation level. In this chapter, we mainly focus on serializability among concurrent transactions. Serializability theory extends in a straightforward manner to the distributed databases. The history of transaction execution at each site is called a *local history*. If the database is not replicated and each local history is serializable, their union (called the *global history*) is also serializable as long as local serialization orders are identical.

Example 5.1 We give a very simple example to demonstrate the point. Consider two bank accounts, x (stored at Site 1) and y (stored at Site 2), and the following two transactions where T_1 transfers \$100 from x to y, while T_2 simply reads the balance of x and y:

T_1:	Read(x)	T_2:	Read(x)
	$x \leftarrow x - 100$		Read(y)
	Write(x)		Commit
	Read(y)		
	$y \leftarrow y + 100$		
	Write(y)		
	Commit		

Obviously, both of these transactions need to run at both sites. Consider the following two histories that may be generated locally at the two sites (H_i is the history at Site i and R_j and W_j are Read and Write operations, respectively, in transaction T_j):

$$H_1 = \{R_1(x), W_1(x), R_2(x)\}$$
$$H_2 = \{R_1(y), W_1(y), R_2(y)\}$$

Both of these histories are serializable; indeed, they are serial. Therefore, each represents a correct execution order. Furthermore, the serialization order for both are the same $T_1 \rightarrow T_2$. Therefore global history that is obtained is also serializable with the serialization order $T_1 \rightarrow T_2$.

However, if the histories generated at the two sites are as follows, there is a problem:

$$H_1' = \{R_1(x), W_1(x), R_2(x)\}$$
$$H_2' = \{R_2(y), R_1(y), W_1(y)\}$$

Although each local history is still serializable, the serialization orders are different: H_1' serializes T_1 before T_2 ($T_1 \rightarrow T_2$) while H_2' serializes T_2 before T_1 ($T_2 \rightarrow T_1$).Therefore, there can be no serializable global history. ♦

Concurrency control protocols are in charge of isolation. A protocol aims at guaranteeing a particular isolation level such as serializability, snapshot isolation, or read committed. There are different aspects or dimensions of concurrency control. The first one is obviously the isolation level(s) aimed by the algorithm. The second aspect is whether a protocol prevents the isolation to be broken (pessimistic) or whether it allows it to be broken and then aborts one of the conflicting transactions to preserve the isolation level (optimistic).

The third dimension is how transactions get serialized. They can be serialized depending on the order of conflicting accesses or a predefined order, called *timestamp order*. The first case corresponds to locking algorithms where transactions get serialized based on the order they try to acquire conflicting locks. The second case corresponds to algorithms that order transactions according to a timestamp. The timestamp can either be assigned at the start of the transaction (start timestamp) when they are pessimistic or just before committing the transaction (commit timestamp) if they are optimistic. The fourth dimension that we consider is how updates are maintained. One possibility is to keep a single version of the data (that it is possible in pessimistic algorithms). Another possibility is to keep multiple versions of the data. The latter is needed for optimistic algorithms, but some pessimistic algorithms rely on it as well for recovery purposes (basically, they keep two versions: the latest committed one and the current uncommitted one). We discuss replication in the next chapter.

Most combinations of these multiple dimensions have been explored. In the remainder of this section, we focus on the seminal techniques for pessimistic algorithms, locking (Sect. 5.2.1) and timestamp ordering (Sect. 5.2.2), and the optimistic ones (Sect. 5.2.4). Understanding these techniques will also help the reader to move on to more involved algorithms of this kind.

5.2.1 Locking-Based Algorithms

Locking-based concurrency control algorithms prevent isolation violation by maintaining a "lock" for each lock unit and requiring each operation to obtain a lock on the data item before it is accessed—either in read (shared) mode or in write (exclusive) mode. The operation's access request is decided based on the compatibility of lock modes—read lock is compatible with another read lock, a write lock is not compatible with either a read or a write lock. The system manages the locks using the two-phase locking algorithm. The fundamental decision in distributed locking-based concurrency control algorithms is where and how the locks are maintained (usually called a *lock table*). The following sections provide algorithms that make different decisions in this regard.

5.2.1.1 Centralized 2PL

The 2PL algorithm can easily be extended to the distributed DBMS environment by delegating lock management responsibility to a single site. This means that only one of the sites has a lock manager; the transaction managers at the other sites communicate with it to obtain locks. This approach is also known as the *primary site* 2PL algorithm.

The communication between the cooperating sites in executing a transaction according to a centralized 2PL (C2PL) algorithm is depicted in Fig. 5.3 where the order of messages is indicated. This communication is between the *coordinating* TM, the lock manager at the central site, and the data processors (DP) at the other participating sites. The participating sites are those that store the data items on which the operation is to be carried out.

The centralized 2PL transaction management algorithm (C2PL-TM) that incorporates these changes is given at a very high level in Algorithm 5.1, while the centralized 2PL lock management algorithm (C2PL-LM) is shown in Algorithm 5.2. A highly simplified data processor algorithm (DP) is given in Algorithm 5.3, which will see major changes when we discuss reliability issues in Sect. 5.4.

These algorithms use a 5-tuple for the operation they perform: $Op : \langle Type = \{BT, R, W, A, C\}, arg : \text{Data item}, val : \text{Value}, tid : \text{Transaction identifier}, res : \text{Result}\rangle$. For an operation $o : Op$, $o.Type \in \{BT, R, W, A, C\}$ specifies its type where BT = Begin_transaction, R = Read, W = Write, A = Abort, and C = Commit, arg is the data item that the operation accesses (reads or writes; for other operations this field is null), val is the value that has been read or to be written for data item arg

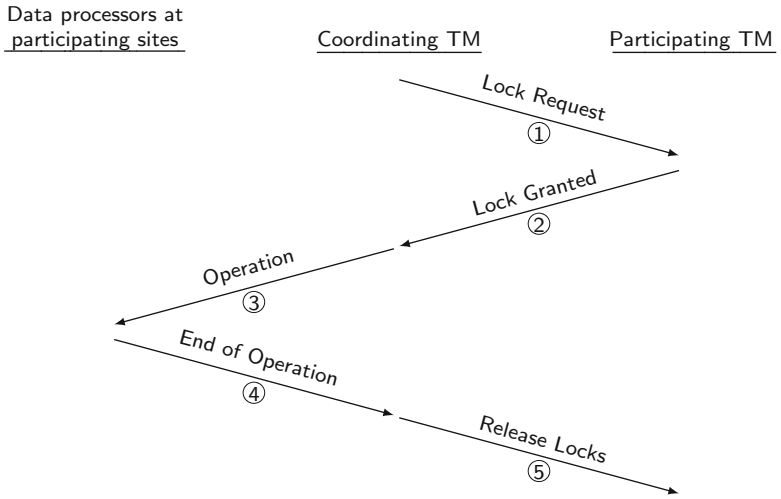

Fig. 5.3 Communication structure of centralized 2PL

Algorithm 5.1: Centralized 2PL Transaction Manager (C2PL-TM)

```
Input: msg : a message
begin
    repeat
        wait for a msg
        switch msg do
            case transaction operation do
                let op be the operation
                if op.Type = BT then DP(op)                    {call DP with operation}
                else C2PL-LM(op)                               {call LM with operation}
            end case
            case Lock Manager response do      {lock request granted or locks released}
                if lock request granted then
                    find site that stores the requested data item (say H_i)
                    DP_Si(op)                                  {call DP at site S_i with operation}
                else                                           {must be lock release message}
                    inform user about the termination of transaction
                end if
            end case
            case Data Processor response do                    {operation completed message}
                switch transaction operation do
                    let op be the operation
                    case R do
                        return op.val (data item value) to the application
                    end case
                    case W do
                        inform application of completion of the write
                    end case
                    case C do
                        if commit msg has been received from all participants then
                            inform application of successful completion of transaction
                            C2PL-LM(op)                        {need to release locks}
                        else                {wait until commit messages come from all}
                            record the arrival of the commit message
                        end if
                    end case
                    case A do
                        inform application of completion of the abort
                        C2PL-LM(op)                            {need to release locks}
                    end case
                end switch
            end case
        end switch
    until forever
end
```

Algorithm 5.2: Centralized 2PL Lock Manager (C2PL-LM)

```
Input: op : Op
begin
    switch op.Type do
        case R or W do                          {lock request; see if it can be granted}
            find the lock unit lu such that op.arg ⊆ lu
            if lu is unlocked or lock mode of lu is compatible with op.Type then
                set lock on lu in appropriate mode on behalf of transaction op.tid
                send "Lock granted" to coordinating TM of transaction
            else
                put op on a queue for lu
            end if
        end case
        case C or A do                          {locks need to be released}
            foreach lock unit lu held by transaction do
                release lock on lu held by transaction
                if there are operations waiting in queue for lu then
                    find the first operation O on queue
                    set a lock on lu on behalf of O
                    send "Lock granted" to coordinating TM of transaction O.tid
                end if
            end foreach
            send "Locks released" to coordinating TM of transaction
        end case
    end switch
end
```

(for other operations it is null), tid is the transaction that this operation belongs to (strictly speaking, this is the transaction identifier), and res indicates the completion code of operations requested of DP, which is important for reliability algorithms.

The transaction manager (C2PL-TM) algorithm is written as a process that runs forever and waits until a message arrives from either an application (with a transaction operation) or from a lock manager, or from a data processor. The lock manager (C2PL-LM) and data processor (DP) algorithms are written as procedures that are called when needed. Since the algorithms are given at a high level of abstraction, this is not a major concern, but actual implementations may, naturally, be quite different.

One common criticism of C2PL algorithms is that a bottleneck may quickly form around the central site. The communication between the cooperating sites in executing a transaction according to a centralized 2PL (C2PL) algorithm is depicted in Fig. 5.3 where the order of messages is indicated. This communication is between the *coordinating* TM, the lock manager at the central site, and the data processors (DP) at the other participating sites. The participating sites are those that store the data items on which the operation is to be carried out. Furthermore, the system may be less reliable since the failure or inaccessibility of the central site would cause major system failures.

Algorithm 5.3: Data Processor (DP)

```
Input: op : Op
begin
    switch op.Type do                                   {check the type of operation}
        case BT do                                      {details to be discussed in Sect. 5.4}
            do some bookkeeping
        end case
        case R do
            op.res ← READ(op.arg) ;                     {database READ operation}
            op.res ← "Read done"
        end case
        case W do                                       {database WRITE of val into data item arg}
            WRITE(op.arg, op.val)
            op.res ← "Write done"
        end case
        case C do
            COMMIT ;                                    {execute COMMIT }
            op.res ← "Commit done"
        end case
        case A do
            ABORT ;                                     {execute ABORT }
            op.res ← "Abort done"
        end case
    end switch
    return op
end
```

5.2.1.2 Distributed 2PL

Distributed 2PL (D2PL) requires the availability of lock managers at each site. The communication between cooperating sites that execute a transaction according to the distributed 2PL protocol is depicted in Fig. 5.4.

The distributed 2PL transaction management algorithm is similar to the C2PL-TM, with two major modifications. The messages that are sent to the central site lock manager in C2PL-TM are sent to the lock managers at all participating sites in D2PL-TM. The second difference is that the operations are not passed to the data processors by the coordinating transaction manager, but by the participating lock managers. This means that the coordinating transaction manager does not wait for a "lock request granted" message. Another point about Fig. 5.4 is the following. The participating data processors send the "end of operation" messages to the coordinating TM. The alternative is for each DP to send it to its own lock manager who can then release the locks and inform the coordinating TM. We have chosen to describe the former since it uses an LM algorithm identical to the strict 2PL lock manager that we have already discussed and it makes the discussion of the commit protocols simpler (see Sect. 5.4). Owing to these similarities, we do not give the

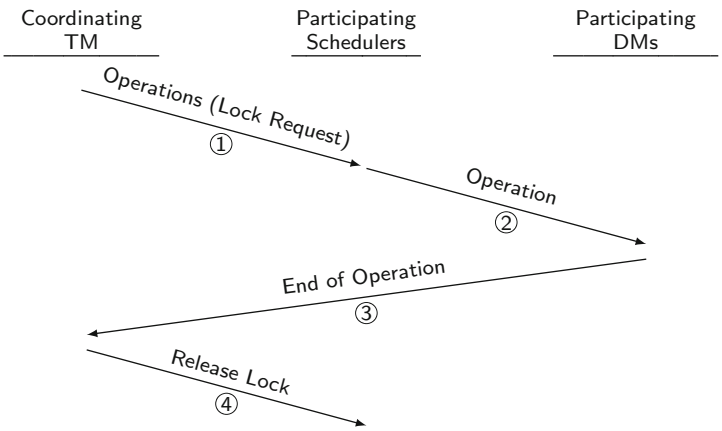

Fig. 5.4 Communication structure of distributed 2PL

distributed TM and LM algorithms here. Distributed 2PL algorithms have been used in R*and in NonStop SQL.

5.2.1.3 Distributed Deadlock Management

Locking-based concurrency control algorithms may cause deadlocks; in the case of distributed DBMSs, these could be *distributed* (or *global*) *deadlocks* due to transactions executing at different sites waiting for each other. Deadlock detection and resolution is the most popular approach to managing deadlocks in the distributed setting. The wait-for graph (WFG) can be useful for detecting deadlocks; this is a directed graph whose vertices are active transactions with an edge from T_i to T_j if an operation in T_i is waiting to access a data item that is currently locked in an incompatible mode by an operation in T_j. However, the formation of the WFG is more complicated in a distributed setting due to the distributed execution of transactions. Therefore, it is not sufficient for each site to form a *local wait-for graph* (LWFG) and check it; it is also necessary to form a *global wait-for graph* (GWFG), which is the union of all the LWFGs, and check it for cycles.

Example 5.2 Consider four transactions T_1, T_2, T_3, and T_4 with the following wait-for relationship among them: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$. If T_1 and T_2 run at site 1 while T_3 and T_4 run at site 2, the LWFGs for the two sites are shown in Fig. 5.5a. Notice that it is not possible to detect a deadlock simply by examining the two LWFGs independently, because the deadlock is global. The deadlock can easily be detected, however, by examining the GWFG where intersite waiting is shown by dashed lines (Fig. 5.5b). ♦

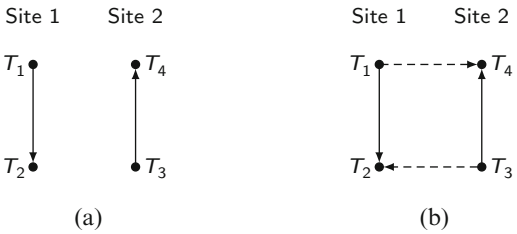

Fig. 5.5 Difference between LWFG and GWFG

The various algorithms differ in how they manage the GWFG. There are three fundamental methods of detecting distributed deadlocks, referred as *centralized, distributed*, and *hierarchical deadlock detection*. We discuss them below.

Centralized Deadlock Detection

In the centralized deadlock detection approach, one site is designated as the deadlock detector for the entire system. Periodically, each lock manager transmits its LWFG to the deadlock detector, which then forms the GWFG and looks for cycles in it. The lock managers need only send changes in their graphs (i.e., the newly created or deleted edges) to the deadlock detector. The length of intervals for transmitting this information is a system design decision: the smaller the interval, the smaller the delays due to undetected deadlocks, but the higher the deadlock detection and communication overhead.

Centralized deadlock detection is simple and would be a very natural choice if the concurrency control algorithm were centralized 2PL. However, the issues of vulnerability to failure, and high communication overhead, must also be considered.

Hierarchical Deadlock Detection

An alternative to centralized deadlock detection is the building of a hierarchy of deadlock detectors (see Fig. 5.6). Deadlocks that are local to a single site would be detected at that site using the LWFG. Each site also sends its LWFG to the deadlock detector at the next level. Thus, distributed deadlocks involving two or more sites would be detected by a deadlock detector in the next lowest level that has control over these sites. For example, a deadlock at site 1 would be detected by the local deadlock detector (DD) at site 1 (denoted DD_{21}, 2 for level 2, 1 for site 1). If, however, the deadlock involves sites 1 and 2, then DD_{11} detects it. Finally, if the deadlock involves sites 1 and 4, DD_{0x} detects it, where x is one of 1, 2, 3, or 4.

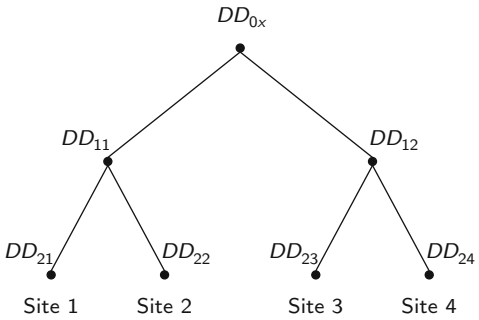

Fig. 5.6 Hierarchical deadlock detection

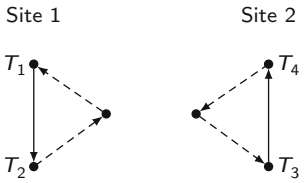

Fig. 5.7 Modified LWFGs

The hierarchical deadlock detection method reduces the dependence on the central site, thus reducing the communication cost. It is, however, considerably more complicated to implement and would involve nontrivial modifications to the lock and transaction manager algorithms.

Distributed Deadlock Detection

Distributed deadlock detection algorithms delegate the responsibility of detecting deadlocks to individual sites. Thus, as in the hierarchical deadlock detection, there are local deadlock detectors at each site that communicate their LWFGs with one another (in fact, only the potential deadlock cycles are transmitted). Among the various distributed deadlock detection algorithms, the one implemented in System R* is the more widely known and referenced, and we describe it below.

The LWFG at each site is formed and is modified as follows:

1. Since each site receives the potential deadlock cycles from other sites, these edges are added to the LWFGs.
2. The edges in the LWFG that show that local transactions are waiting for transactions at other sites are joined with edges in the LWFGs depicting that remote transactions are waiting for local ones.

Example 5.3 Consider the example in Fig. 5.5. The local WFG for the two sites are modified as shown in Fig. 5.7. ♦

Local deadlock detectors look for two things. If there is a cycle that does not include the external edges, there is a local deadlock that can be handled locally. If, on the other hand, there is a cycle involving these external edges, there is a potential distributed deadlock and this cycle information has to be communicated to other deadlock detectors. In the case of Example 5.3, the possibility of such a distributed deadlock is detected by both sites.

A question that needs to be answered at this point is to whom to transmit the information. Obviously, it can be transmitted to all deadlock detectors in the system. In the absence of any more information, this is the only alternative, but it incurs a high overhead. If, however, one knows whether the transaction is ahead or behind in the deadlock cycle, the information can be transmitted forward or backward along the sites in this cycle. The receiving site then modifies its LWFG as discussed above, and checks for deadlocks. Obviously, there is no need to transmit along the deadlock cycle in both the forward and backward directions. In the case of Example 5.3, site 1 would send it to site 2 in both forward and backward transmission along the deadlock cycle.

The distributed deadlock detection algorithms require uniform modification to the lock managers at each site. This uniformity makes them easier to implement. However, there is the potential for excessive message transmission. This happens, for example, in the case of Example 5.3: site 1 sends its potential deadlock information to site 2, and site 2 sends its information to site 1. In this case the deadlock detectors at both sites will detect the deadlock. Besides causing unnecessary message transmission, there is the additional problem that each site may choose a different victim to abort. Obermarck's algorithm solves the problem by using transaction timestamps (monotonically increasing counter—see more details in the next section) as well as the following rule. Let the path that has the potential of causing a distributed deadlock in the local WFG of a site be $T_i \rightarrow \cdots \rightarrow T_j$. A local deadlock detector forwards the cycle information only if timestamp of T_i is smaller than the timestamp of T_j. This reduces the average number of message transmissions by one-half. In the case of Example 5.3, site 1 has a path $T_1 \rightarrow T_2 \rightarrow T_3$, whereas site 2 has a path $T_3 \rightarrow T_4 \rightarrow T_1$. Therefore, assuming that the subscripts of each transaction denote their timestamp, only site 1 will send information to site 2.

5.2.2 *Timestamp-Based Algorithms*

Timestamp-based concurrency control algorithms select, a priori, a serialization order and execute transactions accordingly. To establish this ordering, the transaction manager assigns each transaction T_i a unique *timestamp*, $ts(T_i)$, at its initiation.

Assignment of timestamps in a distributed DBMS requires some attention since multiple sites will be assigning timestamps, and maintaining uniqueness and monotonicity across the system is not easy. One method is to use a global (system-wide) monotonically increasing counter. However, the maintenance of global counters is a problem in distributed systems. Therefore, it is preferable that each site autonomously assigns timestamps based on its local counter. To maintain uniqueness, each site appends its own identifier to the counter value. Thus the timestamp is a two-tuple of the form ⟨local counter value, site identifier⟩. Note that the site identifier is appended in the least significant position. Hence it serves only to order the timestamps of two transactions that might have been assigned the same local counter value. If each system can access its own system clock, it is possible to use system clock values instead of counter values.

Architecturally (see Fig. 5.2), the transaction manager is responsible for assigning a timestamp to each new transaction and attaching this timestamp to each database operation that it passes on to the scheduler. The latter component is responsible for keeping track of read and write timestamps as well as performing the serializability check.

5.2.2.1 Basic TO Algorithm

In the basic TO algorithm the coordinating TM assigns the timestamp to each transaction T_i $[ts(T_i)]$, determines the sites where each data item is stored, and sends the relevant operations to these sites. It is a straightforward implementation of the TO rule.

TO Rule Given two conflicting operations O_{ij} and O_{kl} belonging, respectively, to transactions T_i and T_k, O_{ij} is executed before O_{kl} if and only if $ts(T_i) < ts(T_k)$. In this case T_i is said to be the *older* transaction and T_k is said to be the *younger* one.

A scheduler that enforces the TO rule checks each new operation against conflicting operations that have already been scheduled. If the new operation belongs to a transaction that is younger than all the conflicting ones that have already been scheduled, the operation is accepted; otherwise, it is rejected, causing the entire transaction to restart with a *new* timestamp.

To facilitate checking of the TO Rule, each data item x is assigned two timestamps: a *read timestamp* $[rts(x)]$, which is the largest of the timestamps of the transactions that have read x, and a *write timestamp* $[wts(x)]$, which is the largest of the timestamps of the transactions that have written (updated) x. It is now sufficient to compare the timestamp of an operation with the read and write timestamps of the data item that it wants to access to determine if any transaction with a larger timestamp has already accessed the same data item.

The basic TO transaction manager algorithm (BTO-TM) is depicted in Algorithm 5.4. The histories at each site simply enforce the TO rule. The scheduler

Algorithm 5.4: Basic Timestamp Ordering (BTO-TM)

```
Input: msg : a message
begin
    repeat
        wait for a msg
        switch msg type do
            case transaction operation do          {operation from application program }
                let op be the operation
                switch op.Type do
                    case BT do
                        S ← ∅ ;                    {S: set of sites where transaction executes }
                        assign a timestamp to transaction—call it ts(T)
                        DP(op)                                  {call DP with operation}
                    end case
                    case R, W do
                        find site that stores the requested data item (say S_i)
                        BTO-SC_{S_i}(op, ts(T)) ;          {send op and ts to SC at S_i }
                        S ← S ∪ S_i                {build list of sites where transaction runs}
                    end case
                    case A, C do                   {send op to DPs that execute transaction }
                        DP_S(op)
                    end case
                end switch
            end case
            case SC response do                    {operation must have been rejected by a SC}
                op.Type ← A;                                    {prepare an abort message}
                BTO-SC_S(op, −) ;                               {ask other participating SCs}
                restart transaction with a new timestamp
            end case
            case DP response do                            {operation completed message}
                switch transaction operation type do
                    let op be the operation
                    case R do return op.val to the application
                    case W do inform application of completion of the write
                    case C do
                        if commit msg has been received from all participants then
                            inform application of successful completion of transaction
                        else                       {wait until commit messages come from all}
                            record the arrival of the commit message
                        end if
                    end case
                    case A do
                        inform application of completion of the abort
                        BTO-SC(op)                      {need to reset read and write ts}
                    end case
                end switch
            end case
        end switch
    until forever
end
```

algorithm is given in Algorithm 5.5. The data manager is still the one given in Algorithm 5.3. The same data structures and assumptions we used for centralized 2PL algorithms apply to these algorithms as well.

When an operation is rejected by a scheduler, the corresponding transaction is restarted by the transaction manager with a new timestamp. This ensures that the transaction has a chance to execute in its next try. Since the transactions never wait while they hold access rights to data items, the basic TO algorithm never causes deadlocks. However, the penalty of deadlock freedom is potential restart of a transaction numerous times. There is an alternative to the basic TO algorithm that reduces the number of restarts, which we discuss in the next section.

Another detail that needs to be considered relates to the communication between the scheduler and the data processor. When an accepted operation is passed on to the data processor, the scheduler needs to refrain from sending another conflicting, but acceptable operation to the data processor until the first is processed and acknowledged. This is a requirement to ensure that the data processor executes the operations in the order in which the scheduler passes them on. Otherwise, the read and write timestamp values for the accessed data item would not be accurate.

Algorithm 5.5: Basic Timestamp Ordering Scheduler (BTO-SC)

```
Input: op : Op; ts(T) : Timestamp
begin
    retrieve rts(op.arg) and wts(arg)
    save rts(op.arg) and wts(arg) ;                    {might be needed if aborted }
    switch op.arg do
        case R do
            if ts(T) > wts(op.arg) then
                DP(op) ;                               {operation can be executed; send it to DP}
                rts(op.arg) ← ts(T)
            else
                send "Reject transaction" message to coordinating TM
            end if
        end case
        case W do
            if ts(T) > rts(op.arg) and ts(T) > wts(op.arg) then
                DP(op) ;                               {operation can be executed; send it to DP}
                rts(op.arg) ← ts(T)
                wts(op.arg) ← ts(T)
            else
                send"Reject transaction" message to coordinating TM
            end if
        end case
        case A do
            forall op.arg that has been accessed by transaction do
                reset rts(op.arg) and wts(op.arg) to their initial values
            end forall
        end case
    end switch
end
```

Example 5.4 Assume that the TO scheduler first receives $W_i(x)$ and then receives $W_j(x)$, where $ts(T_i) < ts(T_j)$. The scheduler would accept both operations and pass them on to the data processor. The result of these two operations is that $wts(x) = ts(T_j)$, and we then expect the effect of $W_j(x)$ to be represented in the database. However, if the data processor does not execute them in that order, the effects on the database will be wrong. ♦

The scheduler can enforce the ordering by maintaining a queue for each data item that is used to delay the transfer of the accepted operation until an acknowledgment is received from the data processor regarding the previous operation on the same data item. This detail is not shown in Algorithm 5.5.

Such a complication does not arise in 2PL-based algorithms because the lock manager effectively orders the operations by releasing the locks only after the operation is executed. In one sense the queue that the TO scheduler maintains may be thought of as a lock. However, this does not imply that the history generated by a TO scheduler and a 2PL scheduler would always be equivalent. There are some histories that a TO scheduler would generate that would not be admissible by a 2PL history.

Remember that in the case of strict 2PL algorithms, the releasing of locks is delayed further, until the commit or abort of a transaction. It is possible to develop a strict TO algorithm by using a similar scheme. For example, if $W_i(x)$ is accepted and released to the data processor, the scheduler delays all $R_j(x)$ and $W_j(x)$ operations (for all T_j) until T_i terminates (commits or aborts).

5.2.2.2 Conservative TO Algorithm

We indicated in the preceding section that the basic TO algorithm never causes operations to wait, but instead, restarts them. We also pointed out that even though this is an advantage due to deadlock freedom, it is also a disadvantage, because numerous restarts would have adverse performance implications. The conservative TO algorithms attempt to lower this system overhead by reducing the number of transaction restarts.

Let us first present a technique that is commonly used to reduce the probability of restarts. Remember that a TO scheduler restarts a transaction if a younger conflicting transaction is already scheduled or has been executed. Note that such occurrences increase significantly if, for example, one site is comparatively inactive relative to the others and does not issue transactions for an extended period. In this case its timestamp counter indicates a value that is considerably smaller than the counters of other sites. If the TM at this site then receives a transaction, the operations that are sent to the histories at the other sites will almost certainly be rejected, causing the transaction to restart. Furthermore, the same transaction will restart repeatedly until the timestamp counter value at its originating site reaches a level of parity with the counters of other sites.

The foregoing scenario indicates that it is useful to keep the counters at each site synchronized. However, total synchronization is not only costly—since it requires exchange of messages every time a counter changes—but also unnecessary. Instead, each transaction manager can send its remote operations, rather than histories, to the transaction managers at the other sites. The receiving transaction managers can then compare their own counter values with that of the incoming operation. Any manager whose counter value is smaller than the incoming one adjusts its own counter to one more than the incoming one. This ensures that none of the counters in the system run away or lag behind significantly. Of course, if system clocks are used instead of counters, this approximate synchronization may be achieved automatically as long as the clocks are synchronized with a protocol like Network Time Protocol (NTP).

Conservative TO algorithms execute each operation differently than basic TO. The basic TO algorithm tries to execute an operation as soon as it is accepted; it is therefore "aggressive" or "progressive." Conservative algorithms, on the other hand, delay each operation until there is an assurance that no operation with a smaller timestamp can arrive at that scheduler. If this condition can be guaranteed, the scheduler will never reject an operation. However, this delay introduces the possibility of deadlocks.

The basic technique that is used in conservative TO is based on the following idea: the operations of each transaction are buffered until an ordering can be established so that rejections are not possible, and they are executed in that order. We will consider one possible implementation of the conservative TO algorithm.

Assume that each scheduler maintains one queue for each transaction manager in the system. The scheduler at site s stores all the operations that it receives from the transaction manager at site t in queue Q_s^t. Scheduler at site s has one such queue for each site t. When an operation is received from a transaction manager, it is placed in its appropriate queue in increasing timestamp order. The histories at each site execute the operations from these queues in increasing timestamp order.

This scheme will reduce the number of restarts, but it will not guarantee that they will be eliminated completely. Consider the case where at site s the queue for site t (Q_s^t) is empty. The scheduler at site s will choose an operation [say, $R(x)$] with the smallest timestamp and pass it on to the data processor. However, site t may have sent to s an operation [say, $W(x)$] with a smaller timestamp which may still be in transit in the network. When this operation reaches site s, it will be rejected since it violates the TO rule: it wants to access a data item that is currently being accessed (in an incompatible mode) by another operation with a higher timestamp.

It is possible to design an extremely conservative TO algorithm by insisting that the scheduler choose an operation to be sent to the data processor only if there is at least one operation in each queue. This guarantees that every operation that the scheduler receives in the future will have timestamps greater than or equal to those currently in the queues. Of course, if a transaction manager does not have a transaction to process, it needs to send dummy messages periodically to every scheduler in the system, informing them that the operations that it will send in the future will have timestamps greater than that of the dummy message.

The careful reader will realize that the extremely conservative timestamp ordering scheduler actually executes transactions serially at each site. This is very restrictive. One method that has been employed to overcome this restriction is to group transactions into classes. Transaction classes are defined with respect to their read sets and write sets. It is therefore sufficient to determine the class that a transaction belongs to by comparing the transaction's read set and write set, respectively, with the read set and write set of each class. Thus, the conservative TO algorithm can be modified so that instead of requiring the existence, at each site, of one queue for each transaction manager, it is only necessary to have one queue for each transaction class. Alternatively, one might mark each queue with the class to which it belongs. With either of these modifications, the conditions for sending an operation to the data processor are changed. It is no longer necessary to wait until there is at least one operation in each queue; it is sufficient to wait until there is at least one operation in each class to which the transaction belongs. This and other weaker conditions that reduce the waiting delay can be defined and are sufficient. A variant of this method is used in the SDD-1 prototype system.

5.2.3 Multiversion Concurrency Control

The approaches we discussed above fundamentally address in-place updates: when a data item's value is updated, its old value is replaced with the new one in the database. An alternative is to maintain the versions of data items as they get updated. Algorithms in this class are called *multiversion concurrency control* (MVCC). Then each transaction "sees" the value of a data item based on its isolation level. Multiversion TO is another attempt at eliminating the restart overhead of transactions by maintaining multiple versions of data items and scheduling operations on the appropriate version of the data item. The availability of multiple versions of the database also allows *time travel queries* that track the change of data item values over time. A concern in MVCC is the proliferation of multiple versions of updated data items. To save space, the versions of the database may be purged from time to time. This should be done when the distributed DBMS is certain that it will no longer receive a transaction that needs to access the purged versions.

Although the original proposal dates back to 1978, it has gained popularity in recent years and is now implemented in a number of DBMSs such as IBM DB2, Oracle, SQL Server, SAP HANA, BerkeleyDB, PostgreSQL as well as systems such as Spanner. These systems enforce *snapshot isolation* that we discuss in Sect. 5.3.

MVCC techniques typically use timestamps to maintain transaction isolation although proposals exist that build multiversioning on top of a locking-based concurrency control layer. Here, we will focus on timestamp-based implementation that enforces serializability. In this implementation, each version of a data item that is created is labeled with the timestamp of the transaction that creates it. The idea is that each read operation accesses the version of the data item that is appropriate for its timestamp, thus reducing transaction aborts and restarts. This ensures that

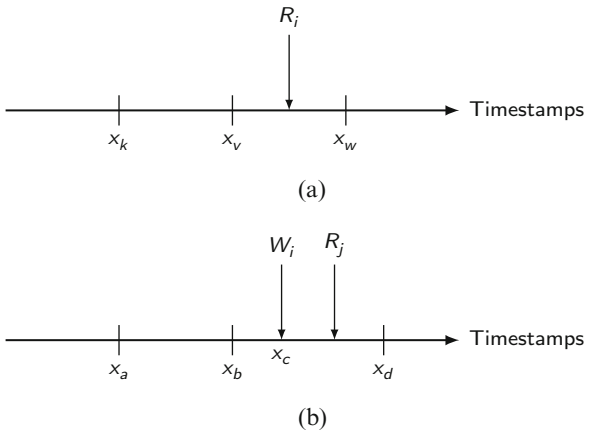

Fig. 5.8 Multiversion TO cases

each transaction operates on a state of the database that it would have seen if the transaction were executed serially in timestamp order.

The existence of versions is transparent to users who issue transactions simply by referring to data items, not to any specific version. The transaction manager assigns a timestamp to each transaction, which is also used to keep track of the timestamps of each version. The operations are processed by the histories as follows guaranteeing a serializable history:

1. A $R_i(x)$ is translated into a read on one version of x. This is done by finding a version of x (say, x_v) such that $ts(x_v)$ is the largest timestamp less than $ts(T_i)$. $R_i(x_v)$ is then sent to the data processor to read x_v. This case is depicted in Fig. 5.8a, which shows that R_i can read the version (x_v) that it would have read had it arrived in timestamp order.
2. A $W_i(x)$ is translated into $W_i(x_w)$ so that $ts(x_w) = ts(T_i)$ and sent to the data processor if and only if no other transaction with a timestamp greater than $ts(T_i)$ has read the value of a version of x (say, x_r) such that $ts(x_r) > ts(x_w)$. In other words, if the scheduler has already processed a $R_j(x_r)$ such that

$$ts(T_i) < ts(x_r) < ts(T_j)$$

then $W_i(x)$ is rejected. This case is depicted in Fig. 5.8b, which shows that if W_i is accepted, it would create a version (x_c) that R_j should have read, but did not since the version was not available when R_j was executed—it, instead, read version x_b, which results in the wrong history.

5.2.4 Optimistic Algorithms

Optimistic algorithms assume that transaction conflicts and contention for data will not be predominant, and therefore allow transactions to execute without synchronization until the very end when they are validated for correctness. Optimistic concurrency control algorithms can be based on locking or timestamping, which was the original proposal. In this section, we describe a timestamp-based distributed optimistic algorithm.

Each transaction follows five phases: read (R), execute (E), write (W), validate (V), and commit (C)—of course commit phase becomes abort if the transaction is not validated. This algorithm assigns timestamps to transactions at the beginning of their validation step rather than at the beginning of transactions as is done with (pessimistic) TO algorithms. Furthermore, it does not associate read and write timestamps with data items—it only works with transaction timestamps during the validation phase.

Each transaction T_i is subdivided (by the transaction manager at the originating site) into a number of subtransactions, each of which can execute at many sites. Notationally, let us denote by T_i^s a subtransaction of T_i that executes at site s. At the beginning of the validation phase a timestamp is assigned to the transaction, which is also the timestamp of its subtransactions. The local validation of T_i^s is performed according to the following rules, which are mutually exclusive.

Rule 1 At each site s, if all transactions T_k^s where $ts(T_k^s) < ts(T_i^s)$ have completed their write phase before T_i^s has started its read phase (Fig. 5.9a), validation succeeds, because transaction executions are in serial order.

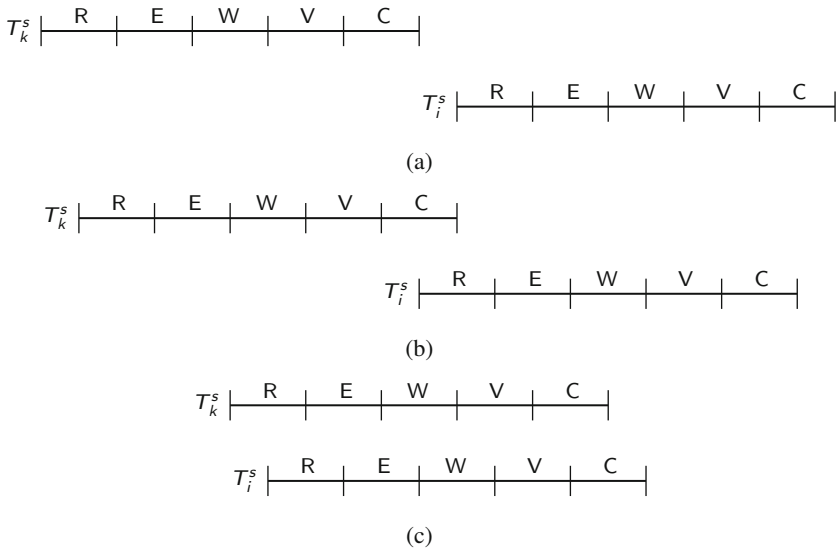

Fig. 5.9 Possible execution scenarios

Rule 2 At each site s, if there is any transaction T_k^s such that $ts(T_k^s) < ts(T_i^s)$, and which completes its write phase while T_i^s is in its read phase (Fig. 5.9b), the validation succeeds if $WS(T_k) \cap RS(T_i^s) = \emptyset$.

Rule 3 At each site s, if there is any transaction T_k^s such that $ts(T_k^s) < ts(T_i^s)$, and which completes its read phase before T_i^s completes its read phase (Fig. 5.9c), the validation succeeds if $WS(T_k^s) \cap RS(T_i^s) = \emptyset$, and $WS(T_k^s) \cap WS(T_i^s) = \emptyset$.

Rule 1 is obvious; it indicates that the transactions are actually executed serially in their timestamp order. Rule 2 ensures that none of the data items updated by T_k^s are read by T_i^s and that T_k^s finishes writing its updates into the database (i.e., commits) before T_i^s starts writing. Thus the updates of T_i^s will not be overwritten by the updates of T_k^s. Rule 3 is similar to Rule 2, but does not require that T_k^s finish writing before T_i^s starts writing. It simply requires that the updates of T_k^s not affect the read phase or the write phase of T_i^s.

Once a transaction is locally validated to ensure that the local database consistency is maintained, it also needs to be globally validated to ensure that the mutual consistency rule is obeyed. This is done by ensuring that the above rules hold at every participating site.

An advantage of optimistic concurrency control algorithms is the potential to allow a higher level of concurrency. It has been shown that when transaction conflicts are very rare, the optimistic mechanism performs better than locking. A difficulty with optimistic approaches is the maintenance of the information necessary for validation. To validate a subtransaction T_i^s the read and write sets of terminated transactions that were in progress when T_i^s arrived at site s need to be maintained.

Another problem is starvation. Consider a situation in which the validation phase of a long transaction fails. In subsequent trials it is still possible that the validation will fail repeatedly. Of course, it is possible to solve this problem by permitting the transaction exclusive access to the database after a specified number of trials. However, this reduces the level of concurrency to a single transaction. The exact mix of transactions that would cause an intolerable level of restarts is an issue that remains to be studied.

5.3 Distributed Concurrency Control Using Snapshot Isolation

Up to this point, we have discussed algorithms that enforce serializability. Although serializability is the most studied and discussed correctness criterion for concurrent transaction execution, for some applications it may be considered too strict in the sense that it disallows certain histories that might be acceptable. In particular, serializability creates a bottleneck that prevents distributed databases from scaling to large levels. The main reason is that it constrains transaction concurrency

very heavily, since large read queries conflict with updates. This has led to the definition of *snapshot isolation* (SI) as an alternative. SI has been widely adopted in commercial systems, and a number of modern systems, such as Google Spanner and LeanXcale, that have managed to scale to very large levels rely on it; we discuss these approaches in Sect. 5.5. Snapshot isolation provides repeatable reads, but not serializable isolation. Each transaction "sees" a consistent snapshot of the database when it starts, and its reads and writes are performed on this snapshot—thus its writes are not visible to other transactions and it does not see the writes of other transactions once it starts executing.

Snapshot isolation is a multiversioning approach, allowing transactions to read the appropriate snapshot (i.e., version). An important advantage of SI-based concurrency control is that read-only transactions can proceed without significant synchronization overhead. For update transactions, the concurrency control algorithm (in centralized systems) is as follows:

S1. When a transaction T_i starts, it obtains a *begin timestamp* $ts_b(T_i)$.
S2. When T_i is ready to commit, it obtains a *commit timestamp* $ts_c(T_i)$ that is greater than any of the existing ts_b or ts_c.
S3. T_i commits its updates if there is no other T_j such that $ts_c(T_j) \in [ts_b(T_i), ts_c(T_i)]$ (i.e., no other transaction has committed since T_i started); otherwise T_i is aborted. This is known as the *first committer wins* rule, and it prevents lost updates.
S4. When T_i is committed, its changes become available to all transactions T_k where $ts_b(T_k) > ts_c(T_i)$.

When SI is used as the correctness criterion in distributed concurrency control, a problem that needs to be addressed is how to compute the consistent snapshot (version) on which transaction T_i operates. If the read and write sets of the transaction are known up-front, it may be possible to centrally compute the snapshot (at the coordinating TM) by collecting information from the participating sites. This, of course, is not realistic. What is needed is a global guarantee similar to the global serializability guarantee we discussed earlier. In other words,

1. Each local history is SI, and
2. The global history is SI, i.e., the commitment orders of transactions at each site are the same.

We now identify the conditions that need to be satisfied for the above guarantee to be realized. We start with defining the *dependence relationship* between two transactions, which is important in this context since the snapshot that a transaction T_i reads should include only the updates of transactions on which it depends. A transaction T_i at site s (T_i^s) is dependent on T_j^s, denoted as $dependent(T_i^s, T_j^s)$, if and only if $(RS(T_i^s) \cap WS(T_j^s) \neq \emptyset) \vee (WS(T_i^s) \cap RS(T_j^s) \neq \emptyset) \vee (WS(T_i^s) \cap WS(T_j^s) \neq \emptyset)$. If there is any participating site where this dependence holds, then $dependent(T_i, T_j)$ holds.

Now we are ready to more precisely specify the conditions that need to hold to ensure global SI as defined above. The conditions below are given for pairwise

transactions, but they transitively hold for a set of transactions. For a transaction T_i to see a globally consistent snapshot, the following conditions have to hold for each pair of transactions:

C1. If $dependent(T_i, T_j) \wedge ts_b(T_i^s) < ts_c(T_j^s)$, then $ts_b(T_i^t) < ts_c(T_j^t)$ at every site t where T_i and T_j execute together.

C2. If $dependent(T_i, T_j) \wedge ts_c(T_i^s) < ts_b(T_j^s)$, then $ts_c(T_i^t) < ts_b(T_j^t)$ at every site t where T_i and T_j execute together.

C3. If $ts_c(T_i^s) < ts_c(T_j^s)$, then $ts_c(T_i^t) < ts_b(T_j^t)$ at every site t where T_i and T_j execute together.

The first two of these conditions ensure that $dependent(T_i, T_j)$ is true at all the sites, i.e., T_i always correctly sees this relationship across sites. The third condition ensures commit order among transactions is the same at all participating sites, and prevents two snapshots from including partial commits that are incompatible with each other.

Before discussing the distributed SI concurrency control algorithm, let us identify the information that each site s maintains:

- For any active transaction T_i, the set of *active* and *committed* transactions at s are categorized into two groups: those that are concurrent with T_i (i.e., any T_j where $ts_b(T_i^s) < ts_c(T_j^s)$), and those that are serial (i.e., any T_j where $ts_c(T_j^s) < ts_b(T_i^s)$)—note that serial is not the same as dependent; local history indicates ordering in the local history at s without any statement on dependence.
- A monotonically increasing event clock.

The basic distributed SI algorithm basically implements step S3 of the centralized algorithm presented earlier (although different implementations exist), i.e., it certifies whether transaction T_i can be committed or needs to be aborted. The algorithm proceeds as follows:

D1. The coordinating TM of T_i asks each participating site s to send its set of transactions concurrent with T_i. It piggybacks to this message its own event clock.

D2. Each site s responds to the coordinating TM with its local set of transactions concurrent with T_i.

D3. The coordinating TM merges all of the local concurrent transaction sets into one global concurrent transaction set for T_i.

D4. The coordinating TM sends this global list of concurrent transactions to all of the participating sites.

D5. Each site s checks whether the conditions C1 and C2 hold. It does this by checking whether there is a transaction T_j in the global concurrent transaction list that is in local history serial list (i.e., in the local history of s, T_j has executed before T_i), and on which T_i is dependent (i.e., $dependent(T_i^s, T_j^s)$ holds). If that is the case, T_i does not see a consistent snapshot at site s, so it should be aborted. Otherwise T_i is validated at site s.

D6. Each site s sends its positive or negative validation to the coordinating TM. If a positive validation message is sent, then site s updates its own event clock to the maximum of its own event clock and the event clock of the coordinating TM that it received, and it piggybacks its new clock value to the response message.

D7. If the coordinating TM receives one negative validation message, then T_i is aborted, since there is at least one site where it does not see a consistent snapshot. Otherwise, the coordinating TM globally certifies T_i and allows it to commit its updates. If the decision is to globally validate, the coordinating TM updates its own event clock to the maximum of the event clocks it receives from the participating sites and its own clock.

D8. The coordinating TM informs all of the participating sites that T_i is validated and can be committed. It also piggybacks its new event clock value, which becomes $ts_c(T_i)$.

D9. Upon receipt of this message, each participant site s makes T_i's updates persistent, and also updates its own event clock as before.

In this algorithm, the certification of conditions C1 and C2 is done in step D5; the other steps serve to collect the necessary information and coordinate the certification check. The event clock synchronization among the sites serves to enforce condition C3 by ensuring that the commit orders of dependent transactions are consistent across sites so that the global snapshot will be consistent.

The algorithm we discussed is one possible approach to implementing SI in a distributed DBMS. It guarantees global SI, but requires that the global snapshot is computed upfront, thus introducing a number of scalability bottlenecks. For instance, sending all concurrent transactions in step D2 obviously does not scale since a system executing millions of concurrent transactions would have to send millions of transactions on each check that would severely limit scalability. Furthermore, it requires all transactions to follow the same certification process. This can be optimized by separating single-site transactions that only access data at one site, and, therefore, do not require the generation of a global snapshot from global transactions that execute across a number of sites. One way to accomplish this is to incrementally build the snapshot that a transaction T_i reads as the data are accessed across different sites.

5.4 Distributed DBMS Reliability

In centralized DBMSs, three types of errors can occur: transaction failures (e.g., transaction aborts), site failures (that cause the loss of data in memory, but not in persistent storage), and media failures (that may cause partial or total loss of data in persistent storage). In the distributed setting, the system needs to cope with a fourth failure type: *communication failures*. There are a number of types of communication failures; the most common ones are the errors in the messages, improperly ordered

messages, lost (or undeliverable) messages, and communication line failures. The first two errors are the responsibility of the computer network, and we will not consider them further. Therefore, in our discussions of distributed DBMS reliability, we expect the underlying computer network to ensure that two messages sent from a process at some originating site to another process at some destination site are delivered without error and in the order in which they were sent; i.e., we consider each communication link to be a reliable FIFO channel.

Lost or undeliverable messages are typically the consequence of communication line failures or (destination) site failures. If a communication line fails, in addition to losing the message(s) in transit, it may also divide the network into two or more disjoint groups. This is called *network partitioning*. If the network is partitioned, the sites in each partition may continue to operate. In this case, executing transactions that access data stored in multiple partitions becomes a major issue. In a distributed system, it is generally not possible to differentiate failures of destination site versus communication lines. In both cases, a source site sends a message but does not get a response within an expected time; this is called a *timeout*. At that point, reliability algorithms need to take action.

Communication failures point to a unique aspect of failures in distributed systems. In centralized systems the system state can be characterized as all-or-nothing: either the system is operational or it is not. Thus the failures are complete: when one occurs, the entire system becomes nonoperational. Obviously, this is not true in distributed systems. As we indicated a number of times before, this is their potential strength. However, it also makes the transaction management algorithms more difficult to design, since, if a message is undelivered, it is hard to know whether the recipient site has failed or a network failure occurred that prevented the message from being delivered.

If messages cannot be delivered, we assume that the network does nothing about it. It will not buffer it for delivery to the destination when the service is reestablished and will not inform the sender process that the message cannot be delivered. In short, the message will simply be lost. We make this assumption because it represents the least expectation from the network and places the responsibility of dealing with these failures to the distributed DBMS. As a consequence, the distributed DBMS is responsible for detecting that a message is undeliverable. The detection mechanism is typically dependent on the characteristics of the communication system on which the distributed DBMS is implemented. The details are beyond our scope; in this discussion we will assume, as noted above, that the sender of a message will set a timer and wait until the end of a timeout period when it will decide that the message has not been delivered.

Distributed reliability protocols aim to maintain the atomicity and durability of distributed transactions that execute over a number of databases. The protocols address the distributed execution of the Begin_transaction, Read, Write, Abort, Commit, and recover commands. The Begin_transaction, Read, and Write commands are executed by Local Recovery Managers (LRMs) in the same way as they are in centralized DBMSs. The ones that require special care in distributed DBMSs are the Commit, Abort, and Read commands. The fundamental difficulty is to ensure that all

of the sites that participate in the execution of a transaction reach the same decision (abort or commit) regarding the fate of the transaction.

The implementation of distributed reliability protocols within the architectural model we have adopted in this book raises a number of interesting and difficult issues. We discuss these in Sect. 5.4.6 after we introduce the protocols. For the time being, we adopt a common abstraction: we assume that at the originating site of a transaction there is a *coordinator* process and at each site where the transaction executes there are *participant* processes. Thus, the distributed reliability protocols are implemented between the coordinator and the participants.

The reliability techniques in distributed database systems consist of commit, termination, and recovery protocols—the commit and recovery protocols specify how the Commit and the recover commands are executed, while the termination protocols specify how the sites that are alive can terminate a transaction when they detect that a site has failed. Termination and recovery protocols are two opposite faces of the recovery problem: given a site failure, termination protocols address how the operational sites deal with the failure, whereas recovery protocols deal with the procedure that the process (coordinator or participant) at a failed site has to go through to recover its state once the site is restarted. In the case of network partitioning, the termination protocols take the necessary measures to terminate the active transactions that execute at different partitions, while the recovery protocols address how the global database consistency is reestablished following reconnection of the partitions of the network.

The primary requirement of commit protocols is that they maintain the atomicity of distributed transactions. This means that even though the execution of the distributed transaction involves multiple sites, some of which might fail while executing, the effects of the transaction on the distributed database is all-or-nothing. This is called *atomic commitment*. We would prefer the termination protocols to be *nonblocking*. A protocol is nonblocking if it permits a transaction to terminate at the operational sites without waiting for recovery of the failed site. This would significantly improve the response-time performance of transactions. We would also like the distributed recovery protocols to be *independent*. Independent recovery protocols determine how to terminate a transaction that was executing at the time of a failure without having to consult any other site. Existence of such protocols would reduce the number of messages that need to be exchanged during recovery. Note that the existence of independent recovery protocols would imply the existence of nonblocking termination protocols, but the reverse is not true.

5.4.1 Two-Phase Commit Protocol

Two-phase commit (2PC) is a very simple and elegant protocol that ensures the atomic commitment of distributed transactions. It extends the effects of local atomic commit actions to distributed transactions by insisting that all sites involved in the execution of a distributed transaction agree to commit the transaction before its

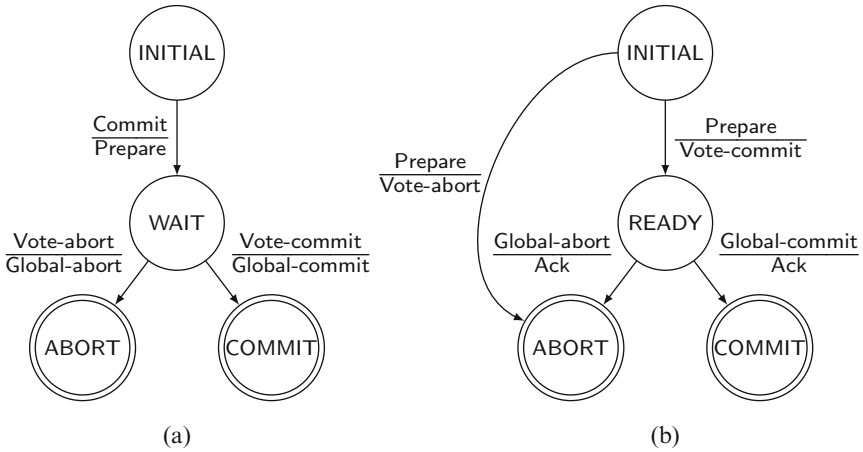

Fig. 5.10 State transitions in 2PC protocol. (**a**) Coordinator states. (**b**) Participant states

effects are made permanent. There are a number of reasons why such synchronization among sites is necessary. First, depending on the type of concurrency control algorithm that is used, some schedulers may not be ready to terminate a transaction. For example, if a transaction has read a value of a data item that is updated by another transaction that has not yet committed, the associated scheduler may not want to commit the former. Of course, strict concurrency control algorithms that avoid cascading aborts would not permit the updated value of a data item to be read by any other transaction until the updating transaction terminates. This is sometimes called the *recoverability condition*.

Another possible reason why a participant may not agree to commit is due to deadlocks that require a participant to abort the transaction. Note that, in this case, the participant should be permitted to abort the transaction without being told to do so. This capability is quite important and is called *unilateral abort*. Another reason for unilateral abort may be timeouts as discussed earlier.

A brief description of the 2PC protocol that does not consider failures is given below. This discussion is facilitated by means of the state transition diagram of the 2PC protocol (Fig. 5.10). The states are denoted by circles and the edges represent the state transitions. The terminal states are depicted by concentric circles. The interpretation of the labels on the edges is as follows: the reason for the state transition, which is a received message, is given at the top, and the message that is sent as a result of state transition is given at the bottom.

1. Initially, the coordinator writes a begin_commit record in its log, sends a "prepare" message to all participant sites, and enters the WAIT state.
2. When a participant receives a "prepare" message, it checks if it could commit the transaction. If so, the participant writes a ready record in the log, sends a "vote-commit" message to the coordinator, and enters READY state; otherwise,

the participant writes an abort record and sends a "vote-abort" message to the coordinator.

3. If the decision of the site is to abort, it can forget about that transaction, since an abort decision serves as a veto (i.e., unilateral abort).
4. After the coordinator has received a reply from every participant, it decides whether to commit or to abort the transaction. If even one participant has registered a negative vote, the coordinator has to abort the transaction globally. So it writes an abort record, sends a "global-abort" message to all participant sites, and enters the ABORT state; otherwise, it writes a commit record, sends a "global-commit" message to all participants, and enters the COMMIT state.
5. The participants either commit or abort the transaction according to the coordinator's instructions and send back an acknowledgment, at which point the coordinator terminates the transaction by writing an end_of_transaction record in the log.

Note the manner in which the coordinator reaches a global termination decision regarding a transaction. Two rules govern this decision, which, together, are called the *global-commit rule*:

1. If even one participant votes to abort the transaction, the coordinator has to reach a global-abort decision.
2. If all the participants vote to commit the transaction, the coordinator has to reach a global-commit decision.

The operation of the 2PC protocol between a coordinator and one participant in the absence of failures is depicted in Fig. 5.11, where the circles indicate the states and the dashed lines indicate messages between the coordinator and the participants. The labels on the dashed lines specify the nature of the message.

A few important points about the 2PC protocol that can be observed from Fig. 5.11 are as follows. First, 2PC permits a participant to unilaterally abort a transaction until it has logged an affirmative vote. Second, once a participant votes to commit or abort a transaction, it cannot change its vote. Third, while a participant is in the READY state, it can move either to abort the transaction or to commit it, depending on the nature of the message from the coordinator. Fourth, the global termination decision is taken by the coordinator according to the global-commit rule. Finally, note that the coordinator and participant processes enter certain states where they have to wait for messages from one another. To guarantee that they can exit from these states and terminate, timers are used. Each process sets its timer when it enters a state, and if the expected message is not received before the timer runs out, the process times out and invokes its timeout protocol (which will be discussed later).

There are a number of different ways to implement a 2PC protocol. The one discussed above and depicted in Fig. 5.11 is called a *centralized 2PC* since the communication is only between the coordinator and the participants; the participants do not communicate among themselves. This communication structure, which is the basis of our subsequent discussions, is depicted more clearly in Fig. 5.12.

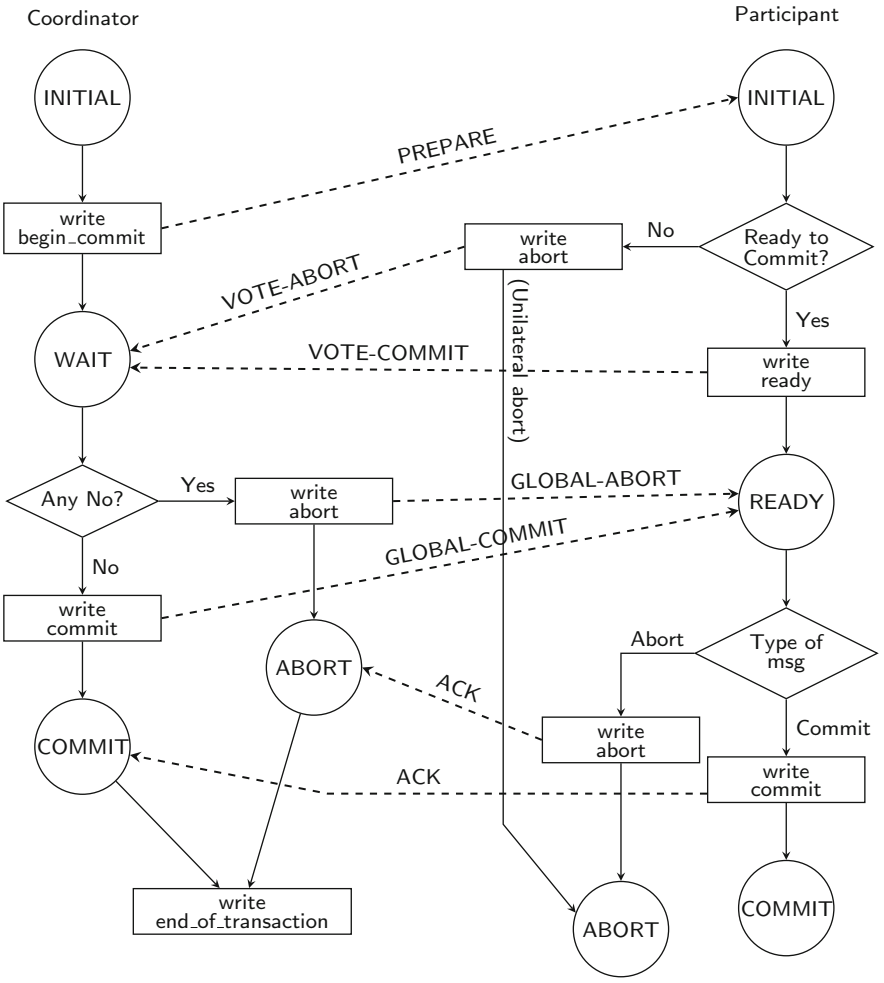

Fig. 5.11 2PC protocol actions

Another alternative is *linear 2PC* (also called *nested 2PC*) where participants can communicate with one another. There is a possibly logical ordering between the sites in the system for the purposes of communication. Let us assume that the ordering among the sites that participate in the execution of a transaction are $1, \ldots, N$, where the coordinator is the first one in the order. The 2PC protocol is implemented by a forward communication from the coordinator (number 1) to N, during which the first phase is completed, and by a backward communication from N to the coordinator, during which the second phase is completed. Thus, linear 2PC operates in the following manner.

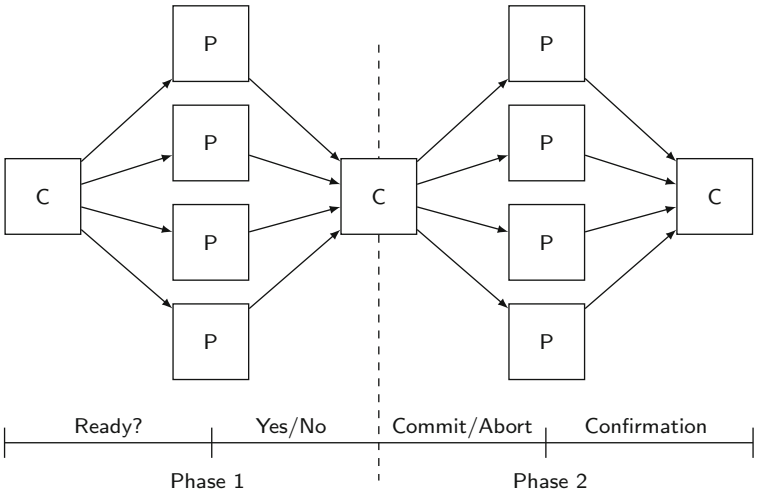

Fig. 5.12 Centralized 2PC communication structure. C: Coordinator, P: Participant

The coordinator sends the "prepare" message to participant 2. If participant 2 is not ready to commit the transaction, it sends a "vote-abort" message (VA) to participant 3 and the transaction is aborted at this point (unilateral abort by 2). If, on the other hand, participant 2 agrees to commit the transaction, it sends a "vote-commit" message (VC) to participant 3 and enters the READY state. As in the centralized 2PC implementation, each site logs its decision before sending the message to the next site. This process continues until a "vote-commit" vote reaches participant N. This is the end of the first phase. If site N decides to commit, it sends back to site $(N - 1)$ "global-commit" (GC); otherwise, it sends a "global-abort" message (GA). Accordingly, the participants enter the appropriate state (COMMIT or ABORT) and propagate the message back to the coordinator.

Linear 2PC, whose communication structure is depicted in Fig. 5.13, incurs fewer messages but does not provide any parallelism. Therefore, it suffers from low response-time performance.

Another popular communication structure for implementation of the 2PC protocol involves communication among all the participants during the first phase of the protocol so that they all independently reach their termination decisions with respect to the specific transaction. This version, called *distributed 2PC*, eliminates the need for the second phase of the protocol since the participants can reach a decision on their own. It operates as follows. The coordinator sends the prepare message to all participants. Each participant then sends its decision to all the other participants (and to the coordinator) by means of either a "vote-commit" or a "vote-abort" message. Each participant waits for messages from all the other participants and makes its termination decision according to the global-commit rule. Obviously, there is no need for the second phase of the protocol (someone sending the global-abort or global-commit decision to the others), since each participant has independently

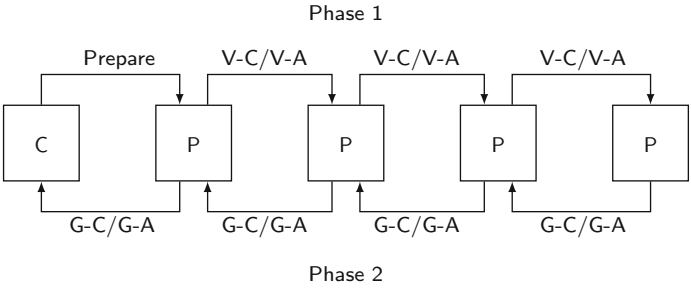

Fig. 5.13 Linear 2PC communication structure. V-C: vote.commit; V-A: vote.abort; G-C: global.commit; G-A: global.abort

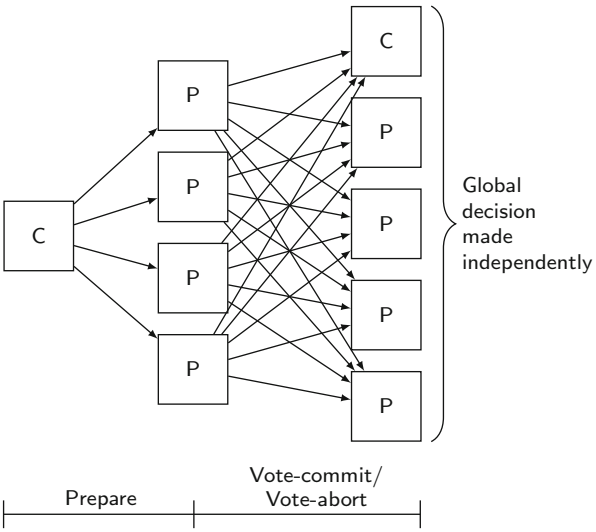

Fig. 5.14 Distributed 2PC communication structure

reached that decision at the end of the first phase. The communication structure of distributed commit is depicted in Fig. 5.14.

In linear and distributed 2PC implementation, a participant has to know the identity of either the next participant in the linear ordering (in case of linear 2PC) or of all the participants (in case of distributed 2PC). This problem can be solved by attaching the list of participants to the prepare message that is sent by the coordinator. Such an issue does not arise in the case of centralized 2PC since the coordinator clearly knows who the participants are.

The algorithm for the centralized execution of the 2PC protocol by the coordinator and by the participants are given in Algorithms 5.6 and 5.7, respectively.

Algorithm 5.6: 2PC Coordinator (2PC-C)

```
begin
    repeat
        wait for an event
        switch event do
            case Msg Arrival do
                Let the arrived message be msg
                switch msg do
                    case Commit do                    {commit command from scheduler}
                        write begin_commit record in the log
                        send "Prepared" message to all the involved participants
                        set timer
                    end case
                    case Vote-abort do {one participant has voted to abort; unilateral
                      abort}
                        write abort record in the log
                        send "Global-abort" message to the other involved participants
                        set timer
                    end case
                    case Vote-commit do
                        update the list of participants who have answered
                        if all the participants have answered then {all must have
                          voted to commit}
                            write commit record in the log
                            send "Global-commit" to all the involved participants
                            set timer
                        end if
                    end case
                    case Ack do
                        update the list of participants who have acknowledged
                        if all the participants have acknowledged then
                            write end_of_transaction record in the log
                        else
                            send global decision to the unanswering participants
                        end if
                    end case
                end switch
            end case
            case Timeout do
                execute the termination protocol
            end case
        end switch
    until forever
end
```

5.4.2 Variations of 2PC

Two variations of 2PC have been proposed to improve its performance. This is accomplished by reducing (1) the number of messages that are transmitted between the coordinator and the participants, and (2) the number of times logs are written.

Algorithm 5.7: 2PC Participant (2PC-P)

```
begin
    repeat
        wait for an event
        switch ev do
            case Msg Arrival do
                Let the arrived message be msg
                switch msg do
                    case Prepare do          {Prepare command from the coordinator}
                        if ready to commit then
                            write ready record in the log
                            send "Vote-commit" message to the coordinator
                            set timer
                        end if
                        else                                   {unilateral abort}
                            write abort record in the log
                            send "Vote-abort" message to the coordinator
                            abort the transaction
                        end if
                    end case
                    case Global-abort do
                        write abort record in the log
                        abort the transaction
                    end case
                    case Global-commit do
                        write commit record in the log
                        commit the transaction
                    end case
                end switch
            end case
            case Timeout do
                execute the termination protocol
            end case
        end switch
    until forever
end
```

These protocols are called *presumed abort* and *presumed commit*. Presumed abort is a protocol that is optimized to handle read-only transactions as well as those update transactions, some of whose processes do not perform any updates to the database (called partially read-only). The presumed commit protocol is optimized to handle the general update transactions. We will discuss briefly both of these variations.

5.4.2.1 Presumed Abort 2PC Protocol

In the presumed abort 2PC protocol, whenever a prepared participant polls the coordinator about a transaction's outcome and there is no information about it, the response to the inquiry is to abort the transaction. This works since, in the case of

a commit, the coordinator does not forget about a transaction until all participants acknowledge, guaranteeing that they will no longer inquire about this transaction.

When this convention is used, it can be seen that the coordinator can forget about a transaction immediately after it decides to abort it. It can write an abort record and not expect the participants to acknowledge the abort command. The coordinator does not need to write an end_of_transaction record after an abort record.

The abort record does not need to be forced, because if a site fails before receiving the decision and then recovers, the recovery routine will check the log to determine the fate of the transaction. Since the abort record is not forced, the recovery routine may not find any information about the transaction, in which case it will ask the coordinator and will be told to abort it. For the same reason, the abort records do not need to be forced by the participants either.

Since it saves some message transmission between the coordinator and the participants in case of aborted transactions, presumed abort 2PC is expected to be more efficient.

5.4.2.2 Presumed Commit 2PC Protocol

Presumed commit 2PC is based on the premise that if no information about the transaction exists, it should be considered committed. However, it is not an exact dual of presumed abort 2PC, since an exact dual would require that the coordinator forget about a transaction immediately after it decides to commit it, that commit records (also the ready records of the participants) not be forced, and that commit commands need not be acknowledged. Consider, however, the following scenario. The coordinator sends prepared messages and starts collecting information, but fails before being able to collect all of them and reach a decision. In this case, the participants will wait until they timeout, and then turn the transaction over to their recovery routines. Since there is no information about the transaction, the recovery routines of each participant will commit the transaction. The coordinator, on the other hand, will abort the transaction when it recovers, thus causing inconsistency.

Presumed commit 2PC solves the problem as follows. The coordinator, prior to sending the prepare message, force-writes a collecting record, which contains the names of all the participants involved in executing that transaction. The participant then enters the COLLECTING state, following which it sends the "prepare" message and enters the WAIT state. The participants, when they receive the "prepare" message, decide what they want to do with the transaction, write an abort/ready record accordingly, and respond with either a "vote-abort" or a "vote-commit" message. When the coordinator receives decisions from all the participants, it decides to abort or commit the transaction. If the decision is to abort, the coordinator writes an abort record, enters the ABORT state, and sends a "global-abort" message. If it decides to commit the transaction, it writes a commit record, sends a "global-commit" command, and forgets the transaction. When the participants receive a "global-commit" message, they write a commit record and update the database. If they receive a "global-abort" message, they write an abort record and

acknowledge. The participant, upon receiving the abort acknowledgment, writes an end_of_transaction record and forgets about the transaction.

5.4.3 Dealing with Site Failures

In this section, we consider the failure of sites in the network. Our aim is to develop nonblocking termination and independent recovery protocols. As we indicated before, the existence of independent recovery protocols would imply the existence of nonblocking recovery protocols. However, our discussion addresses both aspects separately. Also note that in the following discussion we consider only the standard 2PC protocol, not its two variants presented above.

Let us first set the boundaries for the existence of nonblocking termination and independent recovery protocols in the presence of site failures. It has been proven that such protocols exist when a single site fails. In the case of multiple site failures, however, the prospects are not as promising. A negative result indicates that it is not possible to design independent recovery protocols (and, therefore, nonblocking termination protocols) when multiple sites fail. We first develop termination and recovery protocols for the 2PC algorithm and show that 2PC is inherently blocking. We then proceed to the development of atomic commit protocols which are nonblocking in the case of single site failures.

5.4.3.1 Termination and Recovery Protocols for 2PC

Termination Protocols

The termination protocols serve the timeouts for both the coordinator and the participant processes. A timeout occurs at a destination site when it cannot get an expected message from a source site within the expected time period. In this section, we consider that this is due to the failure of the source site.

The method for handling timeouts depends on the timing of failures as well as on the types of failures. We therefore need to consider failures at various points of 2PC execution. In the following, we again refer to the 2PC state transition diagram (Fig. 5.10).

Coordinator Timeouts

There are three states in which the coordinator can timeout: WAIT, COMMIT, and ABORT. Timeouts during the last two are handled in the same manner. So we need to consider only two cases:

1. *Timeout in the WAIT state*. In the WAIT state, the coordinator is waiting for the local decisions of the participants. The coordinator cannot unilaterally commit the transaction since the global-commit rule has not been satisfied. However, it can decide to globally abort the transaction, in which case it writes an abort record in the log and sends a "global-abort" message to all the participants.
2. *Timeout in the COMMIT or ABORT states*. In this case the coordinator is not certain that the commit or abort procedures have been completed by the local recovery managers at all of the participant sites. Thus the coordinator repeatedly sends the "global-commit" or "global-abort" commands to the sites that have not yet responded, and waits for their acknowledgement.

Participant Timeouts

A participant can time out[1] in two states: INITIAL and READY. Let us examine both of these cases.

1. *Timeout in the INITIAL state*. In this state the participant is waiting for a "prepare" message. The coordinator must have failed in the INITIAL state. The participant can unilaterally abort the transaction following a timeout. If the "prepare" message arrives at this participant at a later time, this can be handled in one of two possible ways. Either the participant would check its log, find the abort record, and respond with a "vote-abort," or it can simply ignore the "prepare" message. In the latter case the coordinator would time out in the WAIT state and follow the course we have discussed above.
2. *Timeout in the READY state*. In this state the participant has voted to commit the transaction but does not know the global decision of the coordinator. The participant cannot unilaterally reach a decision. Since it is in the READY state, it must have voted to commit the transaction. Therefore, it cannot now change its vote and unilaterally abort it. On the other hand, it cannot unilaterally decide to commit it, since it is possible that another participant may have voted to abort it. In this case, the participant will remain blocked until it can learn from someone (either the coordinator or some other participant) the ultimate fate of the transaction.

Let us consider a centralized communication structure where the participants cannot communicate with one another. In this case, the participant that is trying to terminate a transaction has to ask the coordinator for its decision and wait until it receives a response. If the coordinator has failed, the participant will remain blocked. This is undesirable.

[1] In some discussions of the 2PC protocol, it is assumed that the participants do not use timers and do not time out. However, implementing timeout protocols for the participants solves some nasty problems and may speed up the commit process. Therefore, we consider this more general case.

If the participants can communicate with each other, a more distributed termination protocol may be developed. The participant that times out can simply ask all the other participants to help it make a decision. Assuming that participant P_i is the one that times out, each of the other participants (P_j) responds in the following manner:

1. P_j is in the INITIAL state. This means that P_j has not yet voted and may not even have received the "prepare" message. It can therefore unilaterally abort the transaction and reply to P_i with a "vote-abort" message.
2. P_j is in the READY state. In this state P_j has voted to commit the transaction but has not received any word about the global decision. Therefore, it cannot help P_i to terminate the transaction.
3. P_j is in the ABORT or COMMIT states. In these states, either P_j has unilaterally decided to abort the transaction, or it has received the coordinator's decision regarding global termination. It can, therefore, send P_i either a "vote-commit" or a "vote-abort" message.

Consider how the participant that times out (P_i) can interpret these responses. The following cases are possible:

1. P_i receives "vote-abort" messages from all P_j. This means that none of the other participants had yet voted, but they have chosen to abort the transaction unilaterally. Under these conditions, P_i can proceed to abort the transaction.
2. P_i receives "vote-abort" messages from some P_j, but some other participants indicate that they are in the READY state. In this case P_i can still go ahead and abort the transaction, since according to the global-commit rule, the transaction cannot be committed and will eventually be aborted.
3. P_i receives notification from all P_j that they are in the READY state. In this case none of the participants knows enough about the fate of the transaction to terminate it properly.
4. P_i receives "global-abort" or "global-commit" messages from all P_j. In this case all the other participants have received the coordinator's decision. Therefore, P_i can go ahead and terminate the transaction according to the messages it receives from the other participants. Incidentally, note that it is not possible for some of the P_j to respond with a "global-abort" while others respond with "global-commit" since this cannot be the result of a legitimate execution of the 2PC protocol.
5. P_i receives "global-abort" or "global-commit" from some P_j, whereas others indicate that they are in the READY state. This indicates that some sites have received the coordinator's decision while others are still waiting for it. In this case P_i can proceed as in case 4 above .4.

These five cases cover all the alternatives that a termination protocol needs to handle. It is not necessary to consider cases where, for example, one participant sends a "vote-abort" message while another one sends "global-commit." This cannot happen in 2PC. During the execution of the 2PC protocol, no process (participant or coordinator) is more than one state transition apart from any other process. For example, if a participant is in the INITIAL state, all other participants are in

either the INITIAL or the READY state. Similarly, the coordinator is either in the INITIAL or the WAIT state. Thus, all the processes in a 2PC protocol are said to be *synchronous within one state transition*.

Note that in case 3, the participant processes stay blocked, as they cannot terminate a transaction. Under certain circumstances there may be a way to overcome this blocking. If during termination all the participants realize that only the coordinator site has failed, they can elect a new coordinator, which can restart the commit process. There are different ways of electing the coordinator. It is possible either to define a total ordering among all sites and elect the next one in order, or to establish a voting procedure among the participants . This will not work, however, if both a participant site and the coordinator site fail. In this case it is possible for the participant at the failed site to have received the coordinator's decision and have terminated the transaction accordingly. This decision is unknown to the other participants; thus if they elect a new coordinator and proceed, there is the danger that they may decide to terminate the transaction differently from the participant at the failed site. It is clear that it is not possible to design termination protocols for 2PC that can guarantee nonblocking termination. The 2PC protocol is, therefore, a blocking protocol. Formally, the protocol is blocking because there is a state in Fig. 5.10 that is adjacent to both the commit and abort state, and when there is a coordinator failure participants are in the ready state. Therefore, it is impossible to determine whether the coordinator went to the abort or commit state until it recovers. The 3PC (three-phase commit) protocol solves this blocking situation by adding a new state, PRECOMMIT, between the wait and commit states to avoid the situation and preventing the blocking situation in the advent of a coordinator failure.

Since we had assumed a centralized communication structure in developing the 2PC algorithms in Algorithms 5.6 and 5.7, we will continue with the same assumption in developing the termination protocols. The portion of code that should be included in the timeout section of the coordinator and the participant 2PC algorithms is given in Algorithms 5.8 and 5.9, respectively.

Algorithm 5.8: 2PC Coordinator Terminate

```
begin
    if in WAIT state then                                  {coordinator is in ABORT state}
        write abort record in the log
        send "Global-abort" message to all the participants
    else                                                   {coordinator is in COMMIT state}
        check for the last log record
        if last log record = abort then
            send "Global-abort" to all participants that have not responded
        else
            send "Global-commit" to all the participants that have not responded
        end if
    end if
    set timer
end
```

Algorithm 5.9: 2PC-Participant Terminate

```
begin
    if in INITIAL state then
        write abort record in the log
    else
        send "Vote-commit" message to the coordinator
        reset timer
    end if
end
```

Recovery Protocols

In the preceding section, we discussed how the 2PC protocol deals with failures from the perspective of the operational sites. In this section, we take the opposite viewpoint: we are interested in investigating protocols that a coordinator or participant can use to recover their states when their sites fail and then restart. Remember that we would like these protocols to be independent. However, in general, it is not possible to design protocols that can guarantee independent recovery while maintaining the atomicity of distributed transactions. This is not surprising given the fact that the termination protocols for 2PC are inherently blocking.

In the following discussion, we again use the state transition diagram of Fig. 5.10. Additionally, we make two interpretive assumptions: (1) the combined action of writing a record in the log and sending a message is assumed to be atomic, and (2) the state transition occurs after the transmission of the response message. For example, if the coordinator is in the WAIT state, this means that it has successfully written the begin_commit record in its log and has successfully transmitted the "prepare" command. This does not say anything, however, about successful completion of the message transmission. Therefore, the "prepare" message may never get to the participants, due to communication failures, which we discuss separately. The first assumption related to atomicity is, of course, unrealistic. However, it simplifies our discussion of fundamental failure cases. At the end of this section we show that the other cases that arise from the relaxation of this assumption can be handled by a combination of the fundamental failure cases.

Coordinator Site Failures

The following cases are possible:

1. *The coordinator fails while in the INITIAL state*. This is before the coordinator has initiated the commit procedure. Therefore, it will start the commit process upon recovery.
2. *The coordinator fails while in the WAIT state*. In this case, the coordinator has sent the "prepare" command. Upon recovery, the coordinator will restart the

commit process for this transaction from the beginning by sending the "prepare" message one more time.

3. *The coordinator fails while in the COMMIT or ABORT states*. In this case, the coordinator will have informed the participants of its decision and terminated the transaction. Thus, upon recovery, it does not need to do anything if all the acknowledgments have been received. Otherwise, the termination protocol is involved.

Participant Site Failures

There are three alternatives to consider:

1. *A participant fails in the INITIAL state*. Upon recovery, the participant should abort the transaction unilaterally. Let us see why this is acceptable. Note that the coordinator will be in the INITIAL or WAIT state with respect to this transaction. If it is in the INITIAL state, it will send a "prepare" message and then move to the WAIT state. Because of the participant site's failure, it will not receive the participant's decision and will time out in that state. We have already discussed how the coordinator would handle timeouts in the WAIT state by globally aborting the transaction.
2. *A participant fails while in the READY state*. In this case the coordinator has been informed of the failed site's affirmative decision about the transaction before the failure. Upon recovery, the participant at the failed site can treat this as a timeout in the READY state and hand the incomplete transaction over to its termination protocol.
3. *A participant fails while in the ABORT or COMMIT state*. These states represent the termination conditions, so, upon recovery, the participant does not need to take any special action.

Additional Cases

Let us now consider the cases that may arise when we relax the assumption related to the atomicity of the logging and message sending actions. In particular, we assume that a site failure may occur after the coordinator or a participant has written a log record but before it can send a message. For this discussion, the reader may wish to refer to Fig. 5.11.

1. *The coordinator fails after the* begin_commit *record is written in the log but before the "prepare" command is sent*. The coordinator would react to this as a failure in the WAIT state (case 2 of the coordinator failures discussed above) and send the "prepare" command upon recovery.
2. *A participant site fails after writing the* ready *record in the log but before sending the "vote-commit" message*. The failed participant sees this as case 2 of the participant failures discussed before.

3. *A participant site fails after writing the* Abort *record in the log but before sending the "vote-abort" message*. This is the only situation that is not covered by the fundamental cases discussed before. However, the participant does not need to do anything upon recovery in this case. The coordinator is in the WAIT state and will time out. The coordinator termination protocol for this state globally aborts the transaction.
4. *The coordinator fails after logging its final decision record* (Abort *or* Commit)*, but before sending its "global-abort" or "global-commit" message to the participants*. The coordinator treats this as its case 3, while the participants treat it as a timeout in the READY state.
5. *A participant fails after it logs an* Abort *or a* Commit *record but before it sends the acknowledgment message to the coordinator*. The participant can treat this as its case 3. The coordinator will handle this by timeout in the COMMIT or ABORT state.

5.4.3.2 Three-Phase Commit Protocol

As noted earlier, blocking commit protocols are undesirable. The three-phase commit protocol (3PC) is designed as a nonblocking protocol when failures are restricted to site failures. When network failures occur, things are complicated.

3PC is interesting from an algorithmic viewpoint, but it incurs high communication overhead in terms of latency, since it involves three rounds of messages with forced writes to the stable log. Therefore, it has not been adopted in real systems—even 2PC is criticized for its high latency due to the sequential phases with forced writes to the log. Therefore, we summarize the approach without going into detailed analysis.

Let us first consider the necessary and sufficient conditions for designing nonblocking atomic commitment protocols. A commit protocol that is synchronous within one state transition is nonblocking if and only if its state transition diagram contains neither of the following:

1. No state that is "adjacent" to both a commit and an abort state.
2. No noncommittable state that is "adjacent" to a commit state.

The term *adjacent* here means that it is possible to go from one state to the other with a single state transition.

Consider the COMMIT state in the 2PC protocol (see Fig. 5.10). If any process is in this state, we know that all the sites have voted to commit the transaction. Such states are called *committable*. There are other states in the 2PC protocol that are *noncommittable*. The one we are interested in is the READY state, which is noncommittable since the existence of a process in this state does not imply that all the processes have voted to commit the transaction.

It is obvious that the WAIT state in the coordinator and the READY state in the participant 2PC protocol violate the nonblocking conditions we have stated above.

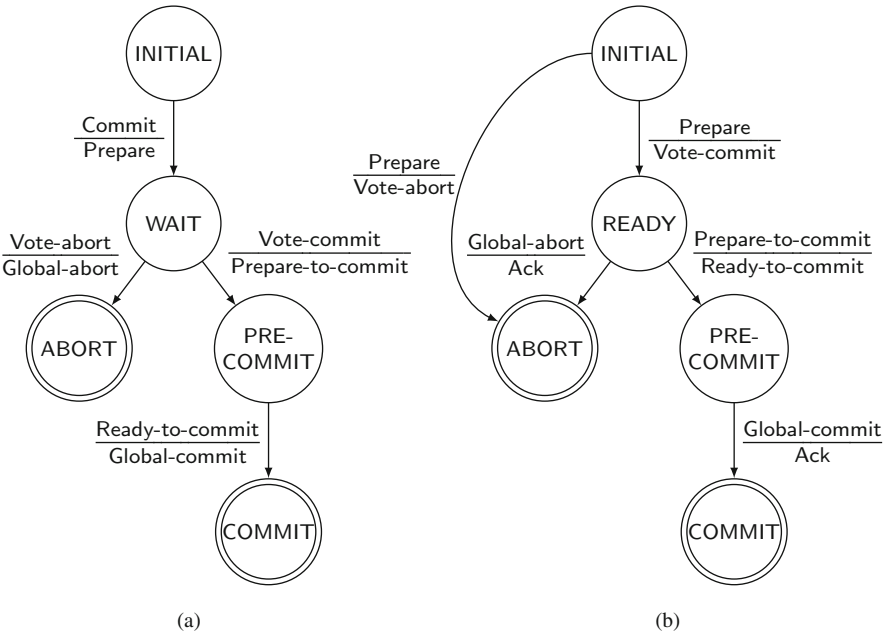

Fig. 5.15 State transitions in 3PC protocol. (**a**) Coordinator states. (**b**) Participant states

Therefore, one might be able to make the following modification to the 2PC protocol to satisfy the conditions and turn it into a nonblocking protocol.

We can add another state between the WAIT (and READY) and COMMIT states which serves as a buffer state where the process is ready to commit (if that is the final decision) but has not yet committed. The state transition diagrams for the coordinator and the participant in this protocol are depicted in Fig. 5.15. This is called the three-phase commit protocol (3PC) because there are three state transitions from the INITIAL state to a COMMIT state. The execution of the protocol between the coordinator and one participant is depicted in Fig. 5.16. Note that this is identical to Fig. 5.11 except for the addition of the PRECOMMIT state. Observe that 3PC is also a protocol where all the states are synchronous within one state transition. Therefore, the foregoing conditions for nonblocking 2PC apply to 3PC.

5.4.4 Network Partitioning

In this section, we consider how the network partitions can be handled by the atomic commit protocols that we discussed in the preceding section. Network partitions are due to communication line failures and may cause the loss of messages, depending

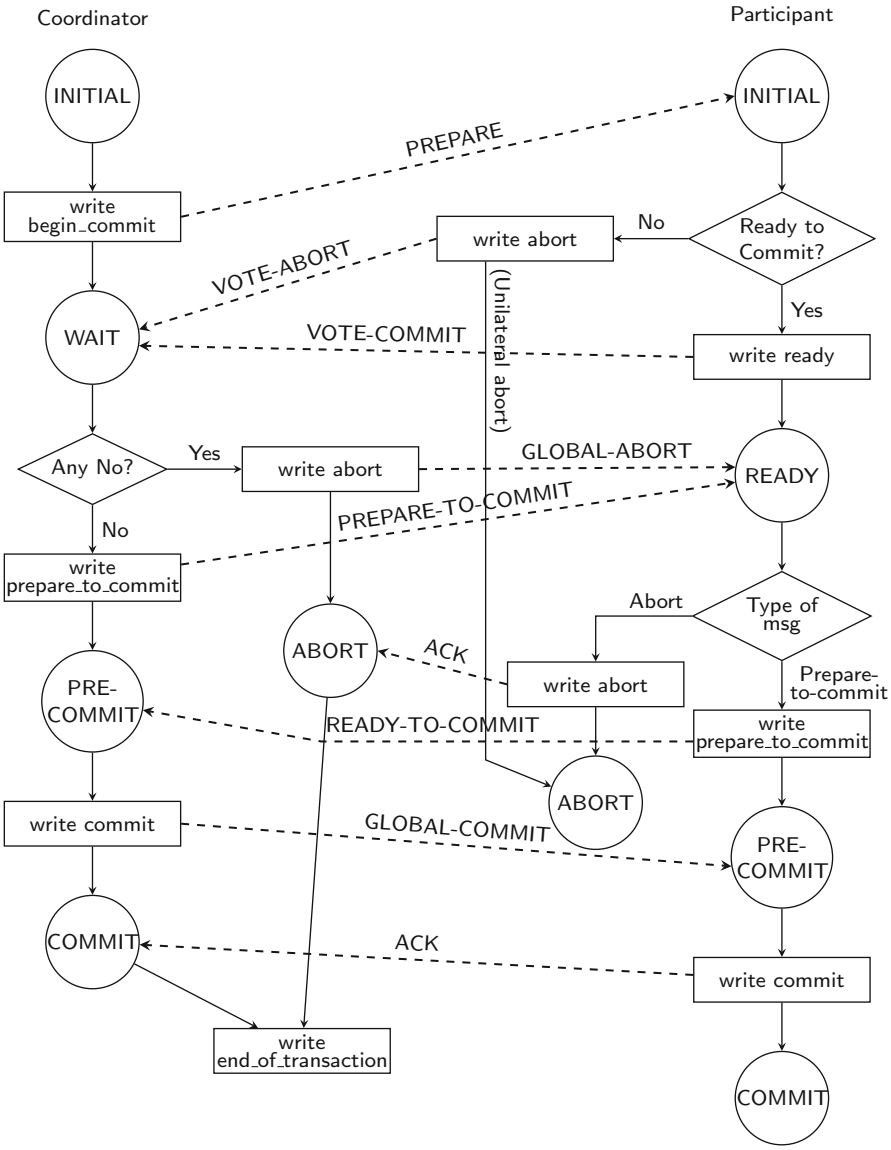

Fig. 5.16 3PC protocol actions

on the implementation of the communication network. A partitioning is called a *simple partitioning* if the network is divided into only two components; otherwise, it is called *multiple partitioning*.

The termination protocols for network partitioning address the termination of the transactions that were active in each partition at the time of partitioning. If one can

develop nonblocking protocols to terminate these transactions, it is possible for the sites in each partition to reach a termination decision (for a given transaction) which is consistent with the sites in the other partitions. This would imply that the sites in each partition can continue executing transactions despite the partitioning.

Unfortunately, generally it is not possible to find nonblocking termination protocols in the presence of network partitioning. Remember that our expectations regarding the reliability of the communication network are minimal. If a message cannot be delivered, it is simply lost. In this case it can be proven that no nonblocking atomic commitment protocol exists that is resilient to network partitioning. This is quite a negative result since it also means that if network partitioning occurs, we cannot continue normal operations in all partitions, which limits the availability of the entire distributed database system. A positive counter result, however, indicates that it is possible to design nonblocking atomic commit protocols that are resilient to simple partitions. Unfortunately, if multiple partitions occur, it is again not possible to design such protocols.

In the remainder of this section we discuss a number of protocols that address network partitioning in nonreplicated databases. The problem is quite different in the case of replicated databases, which we discuss in the next chapter.

In the presence of network partitioning of nonreplicated databases, the major concern is with the termination of transactions that were active at the time of partitioning. Any new transaction that accesses a data item that is stored in another partition is simply blocked and has to await the repair of the network. Concurrent accesses to the data items within one partition can be handled by the concurrency control algorithm. The significant problem, therefore, is to ensure that the transaction terminates properly. In short, the network partitioning problem is handled by the commit protocol, and more specifically, by the termination and recovery protocols.

The absence of nonblocking protocols that would guarantee atomic commitment of distributed transactions points to an important design decision. We can either permit all the partitions to continue their normal operations and accept the fact that database consistency may be compromised, or we guarantee the consistency of the database by employing strategies that would permit operation in one of the partitions while the sites in the others remain blocked. This decision problem is the premise of a classification of partition handling strategies. The strategies can be classified as *pessimistic* or *optimistic*. Pessimistic strategies emphasize the consistency of the database, and would therefore not permit transactions to execute in a partition if there is no guarantee that the consistency of the database can be maintained. Optimistic approaches, on the other hand, emphasize the availability of the database even if this would cause inconsistencies.

The second dimension is related to the correctness criterion. If serializability is used as the fundamental correctness criterion, such strategies are called *syntactic* since the serializability theory uses only syntactic information. However, if we use a more abstract correctness criterion that is dependent on the semantics of the transactions or the database, the strategies are said to be *semantic*.

Consistent with the correctness criterion that we have adopted in this book (serializability), we consider only syntactic approaches in this section. The following two sections outline various syntactic strategies for nonreplicated databases.

All the known termination protocols that deal with network partitioning in the case of nonreplicated databases are pessimistic. Since the pessimistic approaches emphasize the maintenance of database consistency, the fundamental issue that we need to address is which of the partitions can continue normal operations. We consider two approaches.

5.4.4.1 Centralized Protocols

Centralized termination protocols are based on the centralized concurrency control algorithms discussed in Sect. 5.2. In this case, it makes sense to permit the operation of the partition that contains the central site, since it manages the lock tables.

Primary site techniques are centralized with respect to each data item. In this case, more than one partition may be operational for different queries. For any given query, only the partition that contains the primary site of the data items that are in the write set of that transaction can execute that transaction.

Both of these are simple approaches that would work well, but they are dependent on a specific concurrency control mechanism. Furthermore, they expect each site to be able to differentiate network partitioning from site failures properly. This is necessary since the participants in the execution of the commit protocol react differently to the different types of failures. Unfortunately, in general this is not possible.

5.4.4.2 Voting-Based Protocols

Voting can also be used for managing concurrent data accesses. A straightforward voting with majority has been proposed as a concurrency control method for fully replicated databases. The fundamental idea is that a transaction is executed if a majority of the sites vote to execute it.

The idea of majority voting has been generalized to voting with *quorums*. Quorum-based voting can be used as a replica control method (as we discuss in the next chapter), as well as a commit method to ensure transaction atomicity in the presence of network partitioning. In the case of nonreplicated databases, this involves the integration of the voting principle with commit protocols. We present a specific proposal along this line.

Every site in the system is assigned a vote V_i. Let us assume that the total number of votes in the system is V, and the abort and commit quorums are V_a and V_c, respectively. Then the following rules must be obeyed in the implementation of the commit protocol:

1. $V_a + V_c > V$, where $0 \leq V_a,\ V_c \leq V$.

2. Before a transaction commits, it must obtain a commit quorum V_c.
3. Before a transaction aborts, it must obtain an abort quorum V_a.

The first rule ensures that a transaction cannot be committed and aborted at the same time. The next two rules indicate the votes that a transaction has to obtain before it can terminate one way or the other. The integration of quorum techniques into commit protocols is left as an exercise.

5.4.5 Paxos Consensus Protocol

Up to this point, we have studied 2PC protocols for reaching agreement among transaction managers as to the resolution of a distributed transaction and discovered that it has the undesirable property of blocking when the coordinator is down as well as one other participant. We discussed how to overcome this by using 3PC protocol, which is expensive and is not resilient to network partitioning. Our treatment of network partitioning considered voting to determine the partition where a "majority" of transaction managers reside and terminate the transaction in that partition. These may seem like piece-meal solutions to the fundamental problem of finding fault-tolerant mechanisms for reaching an agreement (consensus) among transaction managers about the fate of the transaction under consideration. As it turns out, reaching a consensus among sites is a general problem in distributed computing known as *distributed consensus*. A number of algorithms have been proposed for addressing this problem; in this section, we discuss the Paxos family of algorithms and point to others in Bibliographic Notes.

We will first discuss Paxos in the general setting in which it was originally defined and then consider how it can be used in commit protocols. In the general context, the algorithm achieves a consensus among sites about the value of a variable (or decision). The important consideration is that a consensus is reached if a majority of the sites agree on the value, not all of them. So, certain number of sites may fail, but as long as a majority exists, consensus can be reached. It identifies three roles: *proposer* who recommends a value for the variable, *acceptor* who decides whether or not to accept he recommended value, and *learner* who discovers the agreed-upon value by asking one of the learners (or the value is pushed to it by an acceptor). Note that these are roles all of which can be colocated in one site, but each site can have only one instance of each. The learners are not very important so we do not consider them in any detail in our exposition.

Paxos protocol is simple if there is only one proposer, and it operates like the 2PC protocol: in the first round, the proposer suggests a value for the variable and acceptors send their responses (accept/not accept). If the proposer gets accepts from a majority of the acceptors, then it determines that particular value to be the value of the variable and notifies the acceptors who now record that value the final one. A learner can, at any point, ask an acceptor what the value of the variable is and learn the latest value.

Of course, reality is not this simple and the Paxos protocol needs to be able to deal with the following complications:

1. Since this is, by definition, a distributed consensus protocol, multiple proposers can put forward a value for the same variable. Therefore, an acceptor needs to pick one of the proposed values.
2. Given multiple proposals, it is possible to get split votes on multiple proposals with no proposed value receiving a majority.
3. It is possible that some of the acceptors fail after they accept a value. If the remaining acceptors who accepted that value do not constitute a majority, this causes a problem.

Paxos deals with the first problem by using a ballot number so that acceptors can differentiate different proposals as we discuss below. The second problem can be addressed by running multiple consensus rounds—if no proposal achieves a majority, then another round is run and this is repeated until one value achieves majority. In some cases, this can go on for a number of iterations and this can degrade its performance. Paxos deals with the problem by having a designated *leader* to which every proposer sends its value proposal. The leader then picks one value for each variable and seeks to obtain the majority. This reduces the distributed nature of the consensus protocol. In different rounds of the protocol execution, the leader could be different. The third problem is more serious. Again, this could be treated as the second issue and a new round can be started. However, the complication is that some learners may have learned the accepted value from acceptors in the previous round, and if a different value is chosen in the new round we have inconsistency. Paxos deals with this again by using ballot numbers.

We present below the steps of "basic Paxos" that focuses on determining the value of a single variable (hence the omission of the variable name in the following). Basic Paxos also simplifies the determination of ballot numbers: the ballot number in this case only needs to be unique and monotonic for each proposer; there is no attempt to make them globally unique since a consensus is reached when a majority of the acceptors settle on *some* value for the variable regardless of who proposed it. Below is the basic Paxos operation in the absence of failures:

S1. The proposer who wishes to start a consensus sends to all the acceptors a "prepare" message with its ballot number [prepare(bal)].

S2. Each acceptor that receives the prepare message performs the following:
if it had not received any proposals before
it records prepare(bal) in its log and responds with ack(bal)
else if bal $>$ any ballot number that it had received from any proposer before
then it records prepare(bal) in its log and responds with the ballot number (bal') and value (val') of the highest proposal number it had accepted prior to this: ack(bal, bal', val');
else it ignores the prepare message.

S3. When the proposer receives an ack message from an acceptor, it logs the message.

S4. When the proposer has received acks from a majority of acceptors (i.e., it has a quorum to establish a consensus) it sends an accept($nbal$, val) message to acceptors where $nbal$ is the ballot number to accept and val is the value to accept where $nbal$ and val are determined as follows:
if all of the ack messages it received indicate that no acceptors have previously accepted a value
then the proposed value val is set to what the proposer wanted to suggest in the first place and $nbal \leftarrow bal$;
else val is set to the val' in the return ack messages with the largest bal' and $nbal \leftarrow bal$ (so everyone converges to the value with the largest ballot number);
the proposer now sends accept(bal, val) to all the acceptors (val is the proposed value).

S5. Each acceptor performs the following upon receipt of the accept($nbal$, val) message:
if $nbal$ = ack.bal (i.e., the ballot number of the accept message is the one it had promised earlier)
then it records accepted($nbal$, val)
else it ignores the message.

A number of points about the above protocol. First note that in step S2, an acceptor ignores the prepare message if it has already received a prepare message higher than the one it just received. Although the protocol works correctly in this case, the proposer may continue to communicate with this acceptor later on (e.g., in step S5) and this can be avoided if it were to send back a negative acknowledgement so that the proposer can take it off of future consideration. The second point is that when an acceptor acknowledges a prepare message, it is also acknowledging that the proposer is the Paxos leader for this round. So, in a sense, leader selection is done as part of the protocol. However, to deal with the second problem discussed earlier, it is possible to have a designated leader, which then initiates the rounds. If this is the chosen implementation, then the elected leader can choose which proposal it will put forward if it receives multiple proposals. Finally, since the protocol advances if a majority of participants (sites) are available, for a system with N sites, it can tolerate $\frac{N}{2} - 1$ simultaneous site failures.

Let us briefly analyze how Paxos deals with failures. The simplest case is when some acceptors fail but there is still the quorum for reaching a decision. In this case, the protocol proceeds as usual. If sufficient acceptors fail to eliminate the possibility of a quorum, then this is handled naturally in the protocol by running a new ballot (or multiple ballots) when a quorum can be achieved. The case that is always challenging for consensus algorithms is the failure of the proposer (which is also the leader); in this case Paxos chooses a new leader by some mechanism (there are a number of proposals in literature for this) and the new leader initiates a new round with a new ballot number.

Paxos has the multiround decision making characteristic of 2PC and 3PC and the majority voting method of quorum algorithms; it generalizes these into one coherent protocol to reach consensus in the presence of a set of distributed processes. It has been pointed out that 2PC and 3PC (and other commit protocols) are special cases of Paxos. In the following, we describe one proposal for running 2PC with Paxos in order to achieve a nonblocking 2PC protocol, called Paxos 2PC.

In Paxos 2PC, the transaction managers act as leaders—this is basically noting that what we called coordinator before is now called a leader. A main feature of the protocol is for the leader to use a Paxos protocol to reach consensus and record its decision in a replicated log. The first feature is important since the protocol does not need to have all of the participants active and participating in the decision—a majority will do and others can converge on the decided value when they recover. The second is important because leader (coordinator) failures are no longer blocking—if a leader fails, a new leader can get elected and the state of the transaction decision is available in the replicated log at other sites.

5.4.6 Architectural Considerations

In the previous sections, we have discussed the atomic commit protocols at an abstract level. Let us now look at how these protocols can be implemented within the framework of our architectural model. This discussion involves specification of the interface between the concurrency control algorithms and the reliability protocols. In that sense, the discussions of this chapter relate to the execution of Commit, Abort, and recover commands.

It is not straightforward to specify precisely the execution of these commands for two reasons. First, a significantly more detailed model of the architecture than the one we have presented needs to be considered for correct implementation of these commands. Second, the overall scheme of implementation is quite dependent on the recovery procedures that the local recovery manager implements. For example, implementation of the 2PC protocol on top of a LRM that employs a no-fix/no-flush recovery scheme is quite different from its implementation on top of a LRM that employs a fix/flush recovery scheme. The alternatives are simply too numerous. We therefore confine our architectural discussion to three areas: implementation of the coordinator and participant concepts for the commit and replica control protocols within the framework of the transaction manager-scheduler-local recovery manager architecture, the coordinator's access to the database log, and the changes that need to be made in the local recovery manager operations.

One possible implementation of the commit protocols within our architectural model is to perform both the coordinator and participant algorithms within the transaction managers at each site. This provides some uniformity in executing the distributed commit operations. However, it entails unnecessary communication between the participant transaction manager and its scheduler; this is because

the scheduler has to decide whether a transaction can be committed or aborted. Therefore, it may be preferable to implement the coordinator as part of the transaction manager and the participant as part of the scheduler. If the scheduler implements a strict concurrency control algorithm (i.e., does not allow cascading aborts), it will be ready automatically to commit the transaction when the prepare message arrives. Proof of this claim is left as an exercise. However, even this alternative of implementing the coordinator and the participant outside the data processor has problems. The first issue is database log management. Recall that the database log is maintained by the LRM and the buffer manager. However, implementation of the commit protocol as described here requires the transaction manager and the scheduler to access the log as well. One possible solution to this problem is to maintain a commit log (which could be called the *distributed transaction log*) that is accessed by the transaction manager and is separate from the database log that the LRM and buffer manager maintain. The other alternative is to write the commit protocol records into the same database log. This second alternative has a number of advantages. First, only one log is maintained; this simplifies the algorithms that have to be implemented in order to save log records on stable storage. More important, the recovery from failures in a distributed database requires the cooperation of the local recovery manager and the scheduler (i.e., the participant). A single database log can serve as a central repository of recovery information for both components.

A second problem associated with implementing the coordinator within the transaction manager and the participant as part of the scheduler is integration with the concurrency control protocols. This implementation is based on the schedulers determining whether a transaction can be committed. This is fine for distributed concurrency control algorithms where each site is equipped with a scheduler. However, in centralized protocols such as the centralized 2PL, there is only one scheduler in the system. In this case, the participants may be implemented as part of the data processors (more precisely, as part of local recovery managers), requiring modification to both the algorithms implemented by the LRM and, possibly, to the execution of the 2PC protocol. We leave the details to exercises.

Storing the commit protocol records in the database log maintained by the LRM and the buffer manager requires some changes to the LRM algorithms. This is the third architectural issue we address. These changes are dependent on the type of algorithm that the LRM uses. In general, the LRM algorithms have to be modified to handle separately the prepare command and global-commit (or global-abort) decisions. Furthermore, upon recovery, the LRM should be modified to read the database log and to inform the scheduler as to the state of each transaction, in order that the recovery procedures discussed before can be followed. Let us take a more detailed look at this function of the LRM.

The LRM first has to determine whether the failed site is the host of the coordinator or of a participant. This information can be stored together with the Begin_transaction record. The LRM then has to search for the last record written

in the log record during execution of the commit protocol. If it cannot even find a begin_commit record (at the coordinator site) or an abort or commit record (at the participant sites), the transaction has not started to commit. In this case, the LRM can continue with its recovery procedure. However, if the commit process has started, the recovery has to be handed over to the coordinator. Therefore, the LRM sends the last log record to the scheduler.

5.5 Modern Approaches to Scaling Out Transaction Management

All algorithms presented above introduce bottlenecks at different points in the transactional processing. Those implementing serializability severely limit the potential concurrency due to conflicts between large queries that read many data items and update transactions. For instance, an analytical query that makes a full table scan with a predicate that is not based on the primary key causes a conflict with any update on the table. All algorithms need a centralized processing step in order to commit transactions one by one.

This creates a bottleneck since they cannot process transactions at a rate higher than a single node is able to. Locking algorithms require deadlock management and many use deadlock detection, which is difficult in a distributed setting as we discussed earlier. The algorithm which we presented in the last section on snapshot isolation performs centralized certification, which again introduces a bottleneck.

Scaling transaction execution to achieve high transaction throughput in a distributed or parallel system has been a topic of interest for a long time. In recent years solutions have started to emerge; we discuss two approaches in this section: Google Spanner and LeanXcale. Both of these implement each of the ACID properties in a scalable and composable manner. In both approaches, a new technique is proposed to serialize transactions that can support very high throughput rates (millions or even billion transactions per second). Both approaches have a way to timestamp the commit of transactions and use this commit timestamp to serialize transactions. Spanner uses real time to timestamp transactions, while LeanXcale uses logical time. Real time has the advantage that it does not require any communication, but requires high accuracy and a highly reliable real-time infrastructure. The idea is to use real time as timestamp and wait for the accuracy to elapse to make the result visible to transactions.

LeanXcale adopts an approach in which transactions are timestamped with a logical time and committed transactions are made visible progressively as gaps in the serialization order are filled by newly committed transactions. Logical time avoids having to rely on creating a real-time infrastructure.

5.5.1 Spanner

Spanner uses traditional locking and 2PC and provides serializability as isolation level. Since locking results in high contention between large queries and update transactions, Spanner also implements multiversioning. In order to avoid the bottleneck of centralized certification, updated data items are assigned timestamps (using real time) upon commit. For this purpose, Spanner implements an internal service called TrueTime that provides the current time and its current accuracy. In order to make the TrueTime service reliable and accurate, it uses both atomic clocks and GPS since they have different failures modes that can compensate each other. For instance, atomic clocks they have a continuous drift, while GPS loses accuracy in some meteorological conditions, when the antenna gets broken, etc. The current time obtained through TrueTime is used to timestamp transactions that are going to be committed. The reported accurate is used to compensate during timestamp assignment: after obtaining the local time, there is a wait time for the length of the inaccuracy, typically around 10 milliseconds. To deal with deadlocks, Spanner adopts deadlock avoidance using a wound-and-wait approach (see Appendix C) thereby eliminating the bottleneck that deadlock detection.

Storage management in Spanner is made scalable by leveraging Google Bigtable, a key-value data store (see Chap. 11).

Multiversioning is implemented as follows. Private versions of the data items are kept at each site until commitment. Upon commit, the 2PC protocol is started during which buffered writes are propagated to each participant. Each participant sets locks on the updated data items. Once all locks have been acquired, it assigns a commit timestamp larger than any previously assigned timestamp. The coordinator also acquires the write locks. Upon acquiring all write locks and receiving the prepared message from all participants, the coordinator chooses a timestamp larger than the current time plus the inaccuracy, and bigger than any other timestamps assigned locally. The coordinator waits for the assigned timestamp to pass (recall waiting due to inaccuracy) and then communicates the commit decision to the client.

Using multiversioning, Spanner also implements read-only transactions that read over the snapshot at the current time.

5.5.2 LeanXcale

LeanXcale uses a totally different approach to scalable transactional management. First, it uses logical time to timestamp transactions and to set visibility over committed data. Second, it provides snapshot isolation. Third, all the functions that are intensive in resource usage such as concurrency control, logging, storage and query processing, are fully distributed and parallel, without any coordination.

For logical time, LeanXcale uses two services: the commit sequencer and the snapshot server. The commit sequencer distributes commit timestamps and the

snapshot server regulates the visibility of committed data by advancing the snapshot visible to transactions.

Since LeanXcale provides snapshot isolation, it only needs to detect write-write conflicts. It implements a (possibly distributed) Conflict Manager. Each Conflict Manager takes care of checking conflicts for a subset of the data items. Basically, a Conflict Manager gets requests from LTMs to check whether a data item that is going to be updated conflicts with any concurrent updates. If there is a conflict, then the LTM will abort the transaction, otherwise it will be able to progress without aborting.

The storage functionality is provided by a relational key-value data store called KiVi. Tables are horizontally partitioned into units called *regions*. Each region is stored at one KiVi server.

When a commit is started, it is managed by an LTM that starts the commit processing as follows. First, the LTM takes a commit timestamp from the local range and uses it to timestamp the writeset of the transaction that is then logged. Logging is scaled by using multiple loggers. Each logger serves a subset of LTMs. A logger replies to the LTM when the writeset is durable. LeanXcale implements multiversioning at the storage layer. As in Spanner, private copies exist at each site. When the writeset is durable, the updates are propagated to the corresponding KiVi servers, timestamped with the commit timestamp. Once all updates have been propagated, the transaction is readable if the right start timestamp is used (equal or higher than the commit timestamp). However, it is still invisible. Then, the LTM informs to the snapshot server that the transaction is durable and readable. The snapshot server keeps track of the current snapshot, that is, the start timestamp that will be used by new transactions. It also keeps track of transactions that are durable and readable with a commit timestamp higher than the current snapshot. Whenever there are no gaps between the current snapshot and a timestamp, the snapshot server advances the snapshot to that timestamp. At that point, the committed data with a commit timestamp lower than the current snapshot becomes visible to new transactions since they will get a start timestamp at the current snapshot. Let us see how it works with an example.

Example 5.5 Consider 5 transactions committing in parallel with commit timestamps 11 to 15. Let the current snapshot at the snapshot server be 10. The order in which the LTMs report to the snapshot server that the transactions are durable and readable is 11, 15, 14, 12, and 13. When the snapshot server is notified about transaction with commit timestamp 11, it advances the snapshot from 10 to 11 since there are no gaps. With transaction with commit timestamp 15, it cannot advance the snapshot since otherwise new transactions could observe an inconsistent state that misses updates from transactions with commit timestamps 12 to 14. Now transaction with timestamp 14 reports that its updates are durable and readable. Again, the snapshot cannot progress. But now transaction with commit timestamp 13 becomes durable and readable and now the snapshot advances till 15 since there are no gaps in the serialization order. ♦

Note that although the algorithm so far provides snapshot isolation, it does not provide session consistency, which is a desirable feature for transactions within the same session to be able to read the writes of previously committed transactions. To provide session consistency, a new mechanism is added. Basically, when a session commits a transaction that made updates, it will wait till the snapshot progresses beyond the commit timestamp of the update transaction to start and then it will start with a snapshot that guarantees that it observes its own writes.

5.6 Conclusion

In this chapter, we discussed issues around distributed transaction processing. A transaction is an atomic unit of execution that transforms one consistent database to another consistent database. The ACID properties of transactions also indicate what the requirements for managing them are. Consistency requires a definition of integrity enforcement (which we did in Chap. 3), as well as concurrency control algorithms. Concurrency control also deals with the issue of isolation. The distributed concurrency control mechanism of a distributed DBMS ensures that the consistency of the distributed database is maintained and is therefore one of the fundamental components of a distributed DBMS. We introduced distributed concurrency control algorithms of three types: locking-based, timestamp ordering-based, and optimistic. We noted that locking-based algorithms can lead to distributed (or global) deadlocks and introduced approach for detecting and resolving these deadlocks.

Durability and atomicity properties of transactions require a discussion of distributed DBMS reliability. Specifically, durability is supported by various commit protocols and commit management, whereas atomicity requires the development of appropriate recovery protocols. We introduced two commit protocols, 2PC and 3PC, which guarantee the atomicity and durability of distributed transactions even when failures occur. One of these algorithms (3PC) can be made nonblocking, which would permit each site to continue its operation without waiting for recovery of the failed site. The performance of the distributed commit protocols with respect to the overhead they add to the concurrency control algorithms is an interesting issue.

Achieving very high transaction throughput in distributed and parallel DBMSs has been a long-standing topic of interest with some positive developments in recent years. We discussed two approaches to achieving this objective as part of Spanner and LeanXcale systems.

There are a few issues that we have omitted from this chapter:

1. *Advanced transaction models*. The transaction model that we used In this chapter, is commonly referred to as the *flat transaction* model that have a single start point (Begin_transaction) and a single termination point (End_transaction). Most of the transaction management work in databases has concentrated on flat transactions. However, there are more advanced transaction models. One of these is called the

nested transaction model where a transaction includes other transactions with their own begin and end points. The transactions that are embedded in another one are usually called *subtransactions*. The structure of a nested transaction is a trie where the outermost transaction is the root with sub transactions represented as the other nodes. These differ in their termination characteristics. One category, called *closed nested transactions* commit in a bottom-up fashion through the root. Thus, a nested subtransaction begins *after* its parent and finishes *before* it, and the commitment of the subtransactions is conditional upon the commitment of the parent. The semantics of these transactions enforce atomicity at the topmost level. The alternative is open nesting, which relaxes the top-level atomicity restriction of closed nested transactions. Therefore, an open nested transaction allows its partial results to be observed outside the transaction. Sagas and split transactions are examples of open nesting.

Even more advanced transactions are *workflows*. The term "workflow," unfortunately, does not have a clear and uniformly accepted meaning. A working definition is a set of tasks with a partial order among them that collectively perform some complicated process.

Although the management of these advanced transaction models is important, they are outside our scope, so we have not considered them in this chapter.

2. *Assumptions about transactions*. In our discussions, we did not make any distinction between read-only transactions and update transactions. It is possible to improve significantly the performance of transactions that only read data items, or of systems with a high ratio of read-only transactions to update transactions. These issues are beyond the scope of this book.

We have also treated read and write locks in an identical fashion. It is possible to differentiate between them and develop concurrency control algorithms that permit "lock conversion," whereby transactions can obtain locks in one mode and then modify their lock modes as they change their requirements. Typically, the conversion is from read locks to write locks.

3. *Transaction execution models*. The algorithms that we have described all assume a computational model where the transaction manager at the originating site of a transaction coordinates the execution of each database operation of that transaction. This is called *centralized execution*. It is also possible to consider a *distributed execution* model where a transaction is decomposed into a set of subtransactions each of which is allocated to one site where the transaction manager coordinates its execution. This is intuitively more attractive because it may permit load balancing across the multiple sites of a distributed database. However, the performance studies indicate that distributed computation performs better only under light load.

4. *Error types*. We have considered only failures that are attributable to errors. In other words, we assumed that every effort was made to design and implement the systems (hardware and software), but that because of various faults in the components, the design, or the operating environment, they failed to perform properly. Such failures are called *failures of omission*. There is another class of failures, called *failures of commission*, where the systems may not have been

designed and implemented so that they would work properly. The difference is that in the execution of the 2PC protocol, for example, if a participant receives a message from the coordinator, it treats this message as correct: the coordinator is operational and is sending the participant a correct message to go ahead and process. The only failure that the participant has to worry about is if the coordinator fails or if its messages get lost. These are failures of omission. If, on the other hand, the messages that a participant receives cannot be trusted, the participant also has to deal with failures of commission. For example, a participant site may pretend to be the coordinator and may send a malicious message. We have not discussed reliability measures that are necessary to cope with these types of failures. The techniques that address failures of commission are typically called *byzantine agreement*.

In addition to these issues there is quite a volume of recent work on transaction management in various environments (e.g., multicore, main-memory systems). We do not discuss those in this chapter, that is focused on the fundamentals, but we provide some pointers in the Bibliographic Notes.

5.7 Bibliographic Notes

Transaction management has been the topic of considerable study since DBMSs have become a significant research area. There are two excellent books on the subject: [Gray and Reuter 1993] and [Weikum and Vossen 2001]. Classical texts that focus on these topics are [Hadzilacos 1988] and [Bernstein et al. 1987]. An excellent companion to these is [Bernstein and Newcomer 1997] which provides an in-depth discussion of transaction processing principles. It also gives a view of transaction processing and transaction monitors which is more general than the database-centric view that we provide in this book. A very important work is a set of notes on database operating systems by Gray [1979]. These notes contain valuable information on transaction management, among other things.

Distributed concurrency control is extensively covered in [Bernstein and Goodman 1981], which is now out of print, but can be accessed online. The issues that are addressed In this chapter, are discussed in much more detail in [Cellary et al. 1988, Bernstein et al. 1987, Papadimitriou 1986] and [Gray and Reuter 1993].

For the fundamental techniques we have discussed in the paper, centralized 2PL was first proposed by Alsberg and Day [1976], hierarchical deadlock detection was discussed by Menasce and Muntz [1979] while distributed deadlock detection is due to Obermack [1982]. Our discussion of conservative TO algorithm is due to Herman and Verjus [1979]. The original multiversion TO algorithm was proposed by Reed [1978] with further formalization by Bernstein and Goodman [1983]. Lomet et al. [2012] discuss how to implement multiversioning on top of a concurrency layer that implements 2PL while Faleiro and Abadi [2015] do the same on top of one that implements TO. There are also approaches that implement versioning as a generic

framework on top of any concurrency control technique [Agrawal and Sengupta 1993]. Bernstein et al. [1987] discuss how to implement locking-based optimistic concurrency control algorithms while Thomas [1979] and Kung and Robinson [1981] discuss timestamp-based implementations. Our discussion in Sect. 5.2.4 is due to Ceri and Owicki [1982]. The original snapshot isolation proposal is by Berenson et al. [1995]. Our discussion of the snapshot isolation algorithm is due to Chairunnanda et al. [2014]. Binnig et al. [2014] discuss the optimization that we highlighted at the end of that section. The presumed abort and presumed commit protocols were proposed by Mohan and Lindsay [1983] and Mohan et al. [1986]. Site failures and recoverability from them is the topic of [Skeen and Stonebraker 1983] and [Skeen 1981], the latter also proposes the 3PC algorithm along with its analysis. Coordinator selection protocols that we discuss are due to Hammer and Shipman [1980] and Garcia-Molina [1982]. An early survey of consistency in the presence of network partitioning is by Davidson et al. [1985]. Thomas [1979] proposed the original majority voting technique, and the nonreplicated version of the protocol we discuss is due to Skeen [1982a]. Distributed transaction log idea in Sect. 5.4.6 is due to Bernstein et al. [1987] and Lampson and Sturgis [1976].

The transaction management in System R* discussion is due to Mohan et al. [1986]; NonStop SQL is presented in [Tandem 1987, 1988, Borr 1988]. Bernstein et al. [1980b] discusses SDD-1 in detail. The more modern systems Spanner and LeanXcale discussed in Sect. 5.5 are described in [Corbett et al. 2013] and [Jimenez-Peris and Patiño Martinez 2011], respectively.

Advanced transaction models are discussed and various examples are given in [Elmagarmid 1992]. Nested transactions are also covered in [Lynch et al. 1993]. Closed nested transactions are due to Moss [1985] while open nested transaction model sagas are proposed by Garcia-Molina and Salem [1987], Garcia-Molina et al. [1990] and split transactions by Pu [1988]. Nested transaction models and their specific concurrency control algorithms have been the subjects of some study. Specific results can be found in [Moss 1985, Lynch 1983b, Lynch and Merritt 1986, Fekete et al. 1987a,b, Goldman 1987, Beeri et al. 1989, Fekete et al. 1989] and in [Lynch et al. 1993]. A good introduction to workflow systems is given by Georgakopoulos et al. [1995] and the topic is covered in [Dogac et al. 1998] and [van Hee 2002].

The work on transaction management with semantic knowledge is presented in [Lynch 1983a, Garcia-Molina 1983], and [Farrag and Özsu 1989]. The processing of read-only transactions is discussed in [Garcia-Molina and Wiederhold 1982]. Transaction groups [Skarra et al. 1986, Skarra 1989] also exploit a correctness criterion called *semantic patterns* that is more relaxed than serializability. Furthermore, work on the ARIES system [Haderle et al. 1992] is also within this class of algorithms. In particular, [Rothermel and Mohan 1989] discusses ARIES within the context of nested transactions. Epsilon serializability [Ramamritham and Pu 1995, Wu et al. 1997] and NT/PV model [Kshemkalyani and Singhal 1994] are other "relaxed" correctness criteria. An algorithm based on ordering transactions using *serialization numbers* is discussed in [Halici and Dogac 1989].

Two books focus on the performance of concurrency control mechanisms with a focus on centralized systems [Kumar 1996, Thomasian 1996]. Kumar [1996] focuses on the performance of centralized DBMSs; the performance of distributed concurrency control methods are discussed in [Thomasian 1996] and [Cellary et al. 1988]. An early but comprehensive review of deadlock management is [Isloor and Marsland 1980]. Most of the work on distributed deadlock management has been on detection and resolution (see, e.g., [Obermack 1982, Elmagarmid et al. 1988]). Surveys of the important algorithms are included in [Elmagarmid 1986], [Knapp 1987], and [Singhal 1989].

Snapshot isolation has received significant attention in recent years. Although Oracle had implemented SI since its early versions, the concept was formally defined in [Berenson et al. 1995]. One line of work that we did not cover In this chapter, is to how to get serializable execution even when SI is used as the correctness criterion. This line of work modifies the concurrency control algorithm by detecting the anomalies that are caused by SI that lead to data inconsistency, and preventing them [Cahill et al. 2009, Alomari et al. 2009, Revilak et al. 2011, Alomari et al. 2008] and these techniques have started to be incorporated into systems, e.g., PostgreSQL [Ports and Grittner 2012]. The first SI concurrency control algorithm is due to Schenkel et al. [2000], focusing on concurrency control on data integration systems using SI. We based our discussion In this chapter, on the ConfluxDB system [Chairunnanda et al. 2014]; other work in this direction is by Binnig et al. [2014], where more refined techniques are developed.

Kohler [1981] presents a general discussion of the reliability issues in distributed database systems. Hadzilacos [1988] gives a formalization of the reliability concept. The reliability aspects of System R* are given in [Traiger et al. 1982], whereas Hammer and Shipman [1980] describe the same for the SDD-1 system.

More detailed material on the functions of the local recovery manager can be found in [Verhofstadt 1978, Härder and Reuter 1983]. Implementation of the local recovery functions in System R is described in [Gray et al. 1981].

The two-phase commit protocol is first described in [Gray 1979]. Modifications to it are presented in [Mohan and Lindsay 1983]. The definition of three-phase commit is due to Skeen [1981, 1982b]. Formal results on the existence of nonblocking termination protocols are due to Skeen and Stonebraker [1983].

Paxos was originally proposed by Lamport [1998]. This paper is considered hard to read, which has resulted in a number of different papers describing the protocol. Lamport [2001] gives a significantly simplified description, while Van Renesse and Altinbuken [2015] provide a description that is perhaps between these two points and is a good paper to study. The Paxos 2PC we briefly highlighted is proposed by Gray and Lamport [2006]. For a discussion of how to engineer a Paxos-based system, [Chandra et al. 2007] and [Kirsch and Amir 2008] are recommended. There are many different versions of Paxos—too many to list here—that has resulted in Paxos to be referred to it as a "family of protocols". We do not provide references to each of these. We also note that Paxos is not the only consensus algorithm; a number of alternatives have been proposed particularly as blockchains have become popular (see our discussion of blockchain in Chap. 9). This list is growing fast, which is why

we do not give references. One algorithm, Raft, has been proposed in response to complexity and perceived difficulty in understanding Paxos. The original proposal is by Ongaro and Ousterhout [2014] and it is described nicely in Chapter 23 of [Silberschatz et al. 2019].

As noted earlier, we do not address Byzantine failures in this chapter. The Paxos protocol also does not address these failures. A good description of how to deal with these types of failures is discussed by Castro and Liskov [1999].

Regarding more recent relevant work, Tu et al. [2013] discuss scale-up transaction processing on a single multicore machine. Kemper and Neumann [2011] discuss transaction management issues in a hybrid OLAP/OLTP environment within the context of the HyPer main-memory system. Similarly, Larson et al. [2011] discuss the same issue within the context of the Hekaton system. The E-store system that we discussed in Chap. 2 as part of adaptive data partitioning also addresses transaction management in partitioned distributed DBMSs. As noted there, E-store uses Squall [Elmore et al. 2015] that considers transactions in deciding data movement. Thomson and Abadi [2010] propose Calvin that combine a deadlock avoidance technique with concurrency control algorithms to obtain histories that are guaranteed to be equivalent to a predetermined serial ordering in replicated, distributed DBMSs.

Exercises

Problem 5.1 Which of the following histories are conflict equivalent?

$$H_1 = \{W_2(x), W_1(x), R_3(x), R_1(x), W_2(y), R_3(y), R_3(z), R_2(x)\}$$
$$H_2 = \{R_3(z), R_3(y), W_2(y), R_2(z), W_1(x), R_3(x), W_2(x), R_1(x)\}$$
$$H_3 = \{R_3(z), W_2(x), W_2(y), R_1(x), R_3(x), R_2(z), R_3(y), W_1(x)\}$$
$$H_4 = \{R_2(z), W_2(x), W_2(y), W_1(x), R_1(x), R_3(x), R_3(z), R_3(y)\}$$

Problem 5.2 Which of the above histories $H_1 - H_4$ are serializable?

Problem 5.3 Give a history of two complete transactions which is not allowed by a strict 2PL scheduler but is accepted by the basic 2PL scheduler.

Problem 5.4 (*) One says that history H is *recoverable* if, whenever transaction T_i reads (some item x) from transaction T_j ($i \neq j$) in H and C_i occurs in H, then $C_j \prec_S C_i$. T_i "reads x from" T_j in H if

1. $W_j(x) \prec_H R_i(x)$, and
2. $A_j$not $\prec_H R_i(x)$, and
3. if there is some $W_k(x)$ such that $W_j(x) \prec_H W_k(x) \prec_H R_i(x)$, then $A_k \prec_H R_i(x)$.

Which of the following histories are recoverable?

$$H_1 = \{W_2(x), W_1(x), R_3(x), R_1(x), C_1, W_2(y), R_3(y), R_3(z), C_3, R_2(x), C_2\}$$
$$H_2 = \{R_3(z), R_3(y), W_2(y), R_2(z), W_1(x), R_3(x), W_2(x), R_1(x), C_1, C_2, C_3\}$$
$$H_3 = \{R_3(z), W_2(x), W_2(y), R_1(x), R_3(x), R_2(z), R_3(y), C_3, W_1(x), C_2, C_1\}$$
$$H_4 = \{R_2(z), W_2(x), W_2(y), C_2, W_1(x), R_1(x), A_1, R_3(x), R_3(z), R_3(y), C_3\}$$

Problem 5.5 (*) Give the algorithms for the transaction managers and the lock managers for the distributed two-phase locking approach.

Problem 5.6 (**) Modify the centralized 2PL algorithm to handle phantom read. Phantom read occurs when two reads are executed within a transaction and the result returned by the second read contains tuples that do not exist in the first one. Consider the following example based on the airline reservation database discussed early in this chapter: Transaction T_1, during its execution, searches the FC table for the names of customers who have ordered a special meal. It gets a set of CNAME for customers who satisfy the search criteria. While T_1 is executing, transaction T_2 inserts new tuples into FC with the special meal request, and commits. If T_1 were to re-issue the same search query later in its execution, it will get back a set of CNAME that is different than the original set it had retrieved. Thus, "phantom" tuples have appeared in the database.

Problem 5.7 Timestamp ordering-based concurrency control algorithms depend on either an accurate clock at each site or a global clock that all sites can access (the clock can be a counter). Assume that each site has its own clock which "ticks" every 0.1 second. If all local clocks are resynchronized every 24 hours, what is the maximum drift in seconds per 24 hours permissible at any local site to ensure that a timestamp-based mechanism will successfully synchronize transactions?

Problem 5.8 (**) Incorporate the distributed deadlock strategy described In this chapter, into the distributed 2PL algorithms that you designed in Problem 5.5.

Problem 5.9 Explain the relationship between transaction manager storage requirement and transaction size (number of operations per transaction) for a transaction manager using an optimistic timestamp ordering for concurrency control.

Problem 5.10 (*) Give the scheduler and transaction manager algorithms for the distributed optimistic concurrency controller described in this chapter.

Problem 5.11 Recall from the discussion in Sect. 5.6 that the computational model that is used in our descriptions in this chapter is a centralized one. How would the distributed 2PL transaction manager and lock manager algorithms change if a distributed execution model were to be used?

Problem 5.12 It is sometimes claimed that serializability is quite a restrictive correctness criterion. Can you give examples of distributed histories that are correct (i.e., maintain the consistency of the local databases as well as their mutual consistency) but are not serializable?

Problem 5.13 (*) Discuss the site failure termination protocol for 2PC using a distributed communication topology.

Problem 5.14 (*)
Design a 3PC protocol using the linear communication topology.

Problem 5.15 (*) In our presentation of the centralized 3PC termination protocol, the first step involves sending the coordinator's state to all participants. The participants move to new states according to the coordinator's state. It is possible to design the termination protocol such that the coordinator, instead of sending its own state information to the participants, asks the participants to send their state information to the coordinator. Modify the termination protocol to function in this manner.

Problem 5.16 (**) In Sect. 5.4.6 we claimed that a scheduler which implements a strict concurrency control algorithm will always be ready to commit a transaction when it receives the coordinator's "prepare" message. Prove this claim.

Problem 5.17 (**) Assuming that the coordinator is implemented as part of the transaction manager and the participant as part of the scheduler, give the transaction manager, scheduler, and the local recovery manager algorithms for a nonreplicated distributed DBMS under the following assumptions.

(a) The scheduler implements a distributed (strict) two-phase locking concurrency control algorithm.
(b) The commit protocol log records are written to a central database log by the LRM when it is called by the scheduler.
(c) The LRM may implement any of the protocols that have been discussed (e.g., fix/no-flush or others). However, it is modified to support the distributed recovery procedures as we discussed in Sect. 5.4.6.

Problem 5.18 (*) Write the detailed algorithms for the no-fix/no-flush local recovery manager.

Problem 5.19 (**) Assume that

(a) The scheduler implements a centralized two-phase locking concurrency control,
(b) The LRM implements no-fix/no-flush protocol.

Give detailed algorithms for the transaction manager, scheduler, and local recovery managers.

Chapter 6
Data Replication

As we discussed in previous chapters, distributed databases are typically replicated. The purposes of replication are multiple:

1. **System availability.** As discussed in Chap. 1, distributed DBMSs may remove single points of failure by replicating data, so that data items are accessible from multiple sites. Consequently, even when some sites are down, data may be accessible from other sites.
2. **Performance.** As we have seen previously, one of the major contributors to response time is the communication overhead. Replication enables us to locate the data closer to their access points, thereby localizing most of the access that contributes to a reduction in response time.
3. **Scalability.** As systems grow geographically and in terms of the number of sites (consequently, in terms of the number of access requests), replication allows for a way to support this growth with acceptable response times.
4. **Application requirements.** Finally, replication may be dictated by the applications, which may wish to maintain multiple data copies as part of their operational specifications.

Although data replication has clear benefits, it poses the considerable challenge of keeping different copies synchronized. We will discuss this shortly, but let us first consider the execution model in replicated databases. Each replicated data item x has a number of copies $x_1, x_2, \ldots, x_n$. We will refer to x as the *logical data item* and to its copies (or *replicas*)[1] as *physical data items*. If replication transparency is to be provided, user transactions will issue read and write operations on the logical data item x. The replica control protocol is responsible for mapping these operations to reads and writes on the physical data items $x_1, \ldots, x_n$. Thus, the system behaves as if there is a single copy of each data item—referred to as *single system image* or *one-copy equivalence*. The specific implementation of the Read and Write interfaces

[1] In this chapter, we use the terms "replica," "copy," and "physical data item" interchangeably.

M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*,
https://doi.org/10.1007/978-3-030-26253-2_6

of the transaction monitor differs according to the specific replication protocol, and we will discuss these differences in the appropriate sections.

There are a number of decisions and factors that impact the design of replication protocols. Some of these were discussed in previous chapters, while others will be discussed here.

- **Database design.** As discussed in Chap. 2, a distributed database may be fully or partially replicated. In the case of a partially replicated database, the number of physical data items for each logical data item may vary, and some data items may even be nonreplicated. In this case, transactions that access only nonreplicated data items are *local transactions* (since they can be executed locally at one site) and their execution typically does not concern us here. Transactions that access replicated data items have to be executed at multiple sites and they are *global transactions*.
- **Database consistency.** When global transactions update copies of a data item at different sites, the values of these copies may be different at a given point in time. A replicated database is said to be *mutually consistent* if all the replicas of each of its data items have identical values. What differentiates different mutual consistency criteria is how tightly synchronized replicas have to be. Some ensure that replicas are mutually consistent when an update transaction commits; thus, they are usually called *strong consistency* criteria. Others take a more relaxed approach, and are referred to as *weak consistency* criteria.
- **Where updates are performed.** A fundamental design decision in designing a replication protocol is where the database updates are first performed. The techniques can be characterized as *centralized* if they perform updates first on a *master* copy, versus *distributed* if they allow updates over any replica. Centralized techniques can be further identified as *single master* when there is only one master database copy in the system, or *primary copy* where the master copy of each data item may be different.[2]
- **Update propagation.** Once updates are performed on a replica (master or otherwise), the next decision is how updates are propagated to the others. The alternatives are identified as *eager* versus *lazy*. Eager techniques perform all of the updates within the context of the global transaction that has initiated the write operations. Thus, when the transaction commits, its updates will have been applied to all of the copies. Lazy techniques, on the other hand, propagate the updates sometime after the initiating transaction has committed. Eager techniques are further identified according to when they push each write to the other replicas—some push each write operation individually, others batch the writes and propagate them at the commit point.

[2]Centralized techniques are referred to, in the literature, as *single master*, while distributed ones are referred to as *multimaster* or *update anywhere*. These terms, in particular "single master," are confusing, since they refer to alternative architectures for implementing centralized protocols (more on this in Sect. 6.2.3). Thus, we prefer the more descriptive terms "centralized" and "distributed."

- **Degree of replication transparency.** Certain replication protocols require each user application to know the master site where the transaction operations are to be submitted. These protocols provide only *limited replication transparency* to user applications. Other protocols provide *full replication transparency* by involving the TM at each site. In this case, user applications submit transactions to their local TMs rather than the master site.

We discuss consistency issues in replicated databases in Sect. 6.1, and analyze centralized versus distributed update application as well as update propagation alternatives in Sect. 6.2. This will lead us to a discussion of the specific protocols in Sect. 6.3. In Sect. 6.4, we discuss the use of group communication primitives in reducing the messaging overhead of replication protocols. In these sections, we will assume that no failures occur so that we can focus on the replication protocols. We will then introduce failures and investigate how protocols are revised to handle failures in Sect. 6.5.

6.1 Consistency of Replicated Databases

There are two issues related to consistency of a replicated database. One is mutual consistency, as discussed above, that deals with the convergence of the values of physical data items corresponding to one logical data item. The second is transaction consistency as we discussed in Chap. 5. Serializability, which we introduced as the transaction consistency criterion needs to be recast in the case of replicated databases. In addition, there are relationships between mutual consistency and transaction consistency. In this section, we first discuss mutual consistency approaches and then focus on the redefinition of transaction consistency and its relationship to mutual consistency.

6.1.1 Mutual Consistency

As indicated earlier, mutual consistency criteria for replicated databases can be either strong or weak. Each is suitable for different classes of applications with different consistency requirements.

Strong mutual consistency criteria require that all copies of a data item have the same value at the end of the execution of an update transaction. This is achieved by a variety of means, but the execution of 2PC at the commit point of an update transaction is a common way to achieve strong mutual consistency.

Weak mutual consistency criteria do not require the values of replicas of a data item to be identical when an update transaction terminates. What is required is that, if the update activity ceases for some time, the values *eventually* become identical. This is commonly referred to as *eventual consistency*, which refers to the fact that

replica values may diverge over time, but will eventually converge. It is hard to define this concept formally or precisely, although the following definition by Saito and Shapiro is probably as precise as one can hope to get:

> A replicated [data item] is *eventually consistent* when it meets the following conditions, assuming that all replicas start from the same initial state.
>
> - At any moment, for each replica, there is a prefix of the [history] that is equivalent to a prefix of the [history] of every other replica. We call this a *committed prefix* for the replica.
> - The committed prefix of each replica grows monotonically over time.
> - All nonaborted operations in the committed prefix satisfy their preconditions.
> - For every submitted operation α, either α or [its abort] will eventually be included in the committed prefix.

It should be noted that this definition of eventual consistency is rather strong—in particular the requirements that history prefixes are the same at any given moment and that the committed prefix grows monotonically. Many systems that claim to provide eventual consistency would violate these requirements.

Epsilon serializability (ESR) allows a query to see inconsistent data while replicas are being updated, but requires that the replicas converge to a one-copy equivalent state once the updates are propagated to all of the copies. It bounds the error on the read values by an epsilon (ϵ) value, which is defined in terms of the number of updates (write operations) that a query "misses." Given a read-only transaction (query) T_Q, let T_U be the set of all the update transactions that are executing concurrently with T_Q. If $RS(T_Q) \bigcap WS(T_U) \neq \emptyset$ (T_Q is reading some copy of some data items while a transaction in T_U is updating (possibly a different) copy of those data items), then there is a read–write conflict and T_Q may be reading inconsistent data. The inconsistency is bounded by the changes performed by T_U. Clearly, ESR does not sacrifice database consistency, but only allows read-only transactions (queries) to read inconsistent data. For this reason, it has been claimed that ESR does not weaken database consistency, but "stretches" it.

Other looser bounds have also been discussed. It has even been suggested that users should be allowed to specify *freshness constraints* that are suitable for particular applications and the replication protocols should enforce these. The types of freshness constraints that can be specified are the following:

- **Time-bound constraints.** Users may accept divergence of physical copy values up to a certain time interval: x_i may reflect the value of an update at time t, while x_j may reflect the value at $t - \Delta$ and this may be acceptable.
- **Value-bound constraints.** It may be acceptable to have values of all physical data items within a certain range of each other. The user may consider the database to be mutually consistent if the values do not diverge more than a certain amount (or percentage).
- **Drift constraints on multiple data items.** For transactions that read multiple data items, users may be satisfied if the time drift between the update timestamps of two data items is less than a threshold (i.e., they were updated within that threshold) or, in the case of aggregate computation, if the aggregate computed

over a data item is within a certain range of the most recent value (i.e., even if the individual physical copy values may be more out of sync than this range, as long as a particular aggregate computation is within range, it may be acceptable).

An important criterion in analyzing protocols that employ criteria that allow replicas to diverge is *degree of freshness*. The degree of freshness of a given replica x_i at time t is defined as the proportion of updates that have been applied at x_i at time t to the total number of updates.

6.1.2 Mutual Consistency Versus Transaction Consistency

Mutual consistency, as we have defined it here, and transactional consistency as we discussed in Chap. 5 are related, but different. Mutual consistency refers to the replicas converging to the same value, while transaction consistency requires that the global execution history be serializable. It is possible for a replicated DBMS to ensure that data items are mutually consistent when a transaction commits, but the execution history may not be globally serializable. This is demonstrated in the following example.

Example 6.1 Consider three sites (A, B, and C) and three data items (x, y, z) that are distributed as follows: site A hosts x, site B hosts x, y, site C hosts x, y, z. We will use site identifiers as subscripts on the data items to refer to a particular replica.

Now consider the following three transactions:

T_1:	$x \leftarrow 20$	T_2:	Read(x)	T_3:	Read(x)
	Write(x)		$y \leftarrow x + y$		Read(y)
	Commit		Write(y)		$z \leftarrow (x * y)/100$
			Commit		Write(z)
					Commit

Note that T_1's Write has to be executed at all three sites (since x is replicated at all three sites), T_2's Write has to be executed at B and C, and T_3's Write has to be executed only at C. We are assuming a transaction execution model where transactions can read their local replicas, but have to update all of the replicas.

Assume that the following three local histories are generated at the sites:

$$H_{\mathrm{A}} = \{W_1(x_{\mathrm{A}}), C_1\}$$

$$H_{\mathrm{B}} = \{W_1(x_{\mathrm{B}}), C_1, R_2(x_{\mathrm{B}}), W_2(y_{\mathrm{B}}), C_2\}$$

$$H_{\mathrm{C}} = \{W_2(y_{\mathrm{C}}), C_2, R_3(x_{\mathrm{C}}), R_3(y_{\mathrm{C}}), W_3(z_{\mathrm{C}}), C_3, W_1(x_{\mathrm{C}}), C_1\}$$

The serialization order in H_B is $T_1 \rightarrow T_2$, while in H_C it is $T_2 \rightarrow T_3 \rightarrow T_1$. Therefore, the global history is not serializable. However, the database is mutually consistent. Assume, for example, that initially $x_A = x_B = x_C = 10$, $y_B = y_C =$

15, and $z_C = 7$. With the above histories, the final values will be $x_A = x_B = x_C = 20$, $y_B = y_C = 35$, $z_C = 3.5$. All the physical copies (replicas) have indeed converged to the same value. ♦

Of course, it is possible for the database to be mutually inconsistent and the execution history to be globally nonserializable, as demonstrated in the following example.

Example 6.2 Consider two sites (A and B), and one data item (x) that is replicated at both sites (x_A and x_B). Further consider the following two transactions:

T_1:	Read(x)	T_2:	Read(x)
	$x \leftarrow x + 5$		$x \leftarrow x * 10$
	Write(x)		Write(x)
	Commit		Commit

Assume that the following two local histories are generated at the two sites (again using the execution model of the previous example):

$$H_{\mathrm{A}} = \{R_1(x_{\mathrm{A}}), W_1(x_{\mathrm{A}}), C_1, R_2(x_{\mathrm{A}}), W_2(x_{\mathrm{A}}), C_2\}$$
$$H_{\mathrm{B}} = \{R_2(x_{\mathrm{B}}), W_2(x_{\mathrm{B}}), C_2, R_1(x_{\mathrm{B}}), W_1(x_{\mathrm{B}}), C_1\}$$

Although both of these histories are serial, they serialize T_1 and T_2 in reverse order; thus, the global history is not serializable. Furthermore, the mutual consistency is violated as well. Assume that the value of x prior to the execution of these transactions was 1. At the end of the execution of these schedules, the value of x is 60 at site A, while it is 15 at site B. Thus, in this example, the global history is nonserializable, **and** the databases are mutually inconsistent. ♦

Given the above observation, the transaction consistency criterion given in Chap. 5 is extended in replicated databases to define *one-copy serializability*.One-copy serializability (1SR) states that the effects of transactions on replicated data items should be the same as if they had been performed one at-a-time on a single set of data items. In other words, the histories are equivalent to some serial execution over nonreplicated data items.

Snapshot isolation that we introduced in Chap. 5 has also been extended for replicated databases and used as an alternative transactional consistency criterion within the context of replicated databases. Similarly, a weaker form of serializability, called *relaxed concurrency (RC-)serializability* has been defined that corresponds to read-committed isolation level.

6.2 Update Management Strategies

As discussed earlier, the replication protocols can be classified according to when the updates are propagated to copies (eager versus lazy) and where updates are

allowed to occur (centralized versus distributed). These two decisions are generally referred to as *update management* strategies. In this section, we discuss these alternatives before we present protocols in the next section.

6.2.1 Eager Update Propagation

The eager update propagation approaches apply the changes to all the replicas within the context of the update transaction. Consequently, when the update transaction commits, all the copies have the same value. Typically, eager propagation techniques use 2PC at commit point, but, as we will see later, alternatives are possible to achieve agreement. Furthermore, eager propagation may use *synchronous* propagation of each update by applying it on all the replicas at the same time (when the Write is issued), or *deferred* propagation whereby the updates are applied to one replica when they are issued, but their application on the other replicas is batched and deferred to the end of the transaction. Deferred propagation can be implemented by including the updates in the "Prepare-to-Commit" message at the start of 2PC execution.

Eager techniques typically enforce strong mutual consistency criteria. Since all the replicas are mutually consistent at the end of an update transaction, a subsequent read can read from any copy (i.e., one can map a $R(x)$ to $R(x_i)$ for any x_i). However, a $W(x)$ has to be applied to all replicas (i.e., $W(x_i), \forall x_i$). Thus, protocols that follow eager update propagation are known as *read-one/write-all* (ROWA) protocols.

The advantages of eager update propagation are threefold. First, they typically ensure that mutual consistency is enforced using 1SR; therefore, there are no transactional inconsistencies. Second, a transaction can read a local copy of the data item (if a local copy is available) and be certain that an up-to-date value is read. Thus, there is no need to do a remote read. Finally, the changes to replicas are done atomically; thus, recovery from failures can be governed by the protocols we have already studied in the previous chapter.

The main disadvantage of eager update propagation is that a transaction has to update all the copies before it can terminate. This has two consequences. First, the response time performance of the update transaction suffers, since it typically has to participate in a 2PC execution, and because the update speed is restricted by the slowest machine. Second, if one of the copies is unavailable, then the transaction cannot terminate since all the copies need to be updated. As discussed in Chap. 5, if it is possible to differentiate between site failures and network failures, then one can terminate the transaction as long as only one replica is unavailable (recall that more than one site unavailability causes 2PC to be blocking), but it is generally not possible to differentiate between these two types of failures.

6.2.2 Lazy Update Propagation

In lazy update propagation the replica updates are not all performed within the context of the update transaction. In other words, the transaction does not wait until its updates are applied to all the copies before it commits—it commits as soon as one replica is updated. The propagation to other copies is done *asynchronously* from the original transaction, by means of *refresh transactions* that are sent to the replica sites some time after the update transaction commits. A refresh transaction carries the sequence of updates of the corresponding update transaction.

Lazy propagation is used in those applications for which strong mutual consistency may be unnecessary and too restrictive. These applications may be able to tolerate some inconsistency among the replicas in return for better performance. Examples of such applications are Domain Name Service (DNS), databases over geographically widely distributed sites, mobile databases, and personal digital assistant databases. In these cases, usually weak mutual consistency is enforced.

The primary advantage of lazy update propagation techniques is that they generally have lower response times for update transactions, since an update transaction can commit as soon as it has updated one copy. The disadvantages are that the replicas are not mutually consistent and some replicas may be out-of-date, and consequently, a local read may read stale data and does not guarantee to return the up-to-date value. Furthermore, under some scenarios that we will discuss later, transactions may not see their own writes, i.e., $R_i(x)$ of an update transaction T_i may not see the effects of $W_i(x)$ that was executed previously. This has been referred to as *transaction inversion*. Strong one-copy serializability (strong 1SR) and strong snapshot isolation (strong SI) prevent all transaction inversions at 1SR and SI isolation levels, respectively, but are expensive to provide. The weaker guarantees of 1SR and global SI, while being much less expensive to provide than their stronger counterparts, do not prevent transaction inversions. Session-level transactional guarantees at the 1SR and SI isolation levels have been proposed that address these shortcomings by preventing transaction inversions within a client session but not necessarily across sessions. These session-level guarantees are less costly to provide than their strong counterparts while preserving many of the desirable properties of the strong counterparts.

6.2.3 Centralized Techniques

Centralized update propagation techniques require that updates are first applied at a master copy and then propagated to other copies (which are called *slaves*). The site that hosts the master copy is similarly called the *master site*, while the sites that host the slave copies for that data item are called *slave sites*.

In some techniques, there is a single master for all replicated data. We refer to these as *single master* centralized techniques. In other protocols, the master copy

for each data item may be different (i.e., for data item x, the master copy may be x_i stored at site S_i, while for data item y, it may be y_j stored at site S_j). These are typically known as *primary copy* centralized techniques.

The advantages of centralized techniques are twofold. First, application of the updates is easy since they happen at only the master site, and they do not require synchronization among multiple replica sites. Second, there is the assurance that at least one site—the site that holds the master copy—has up-to-date values for a data item. These protocols are generally suitable in data warehouses and other applications where data processing is centralized at one or a few master sites.

The primary disadvantage is that, as in any centralized algorithm, if there is one central site that hosts all of the masters, this site can be overloaded and can become a bottleneck. Distributing the master site responsibility for each data item as in primary copy techniques is one way of reducing this overhead, but it raises consistency issues, in particular with respect to maintaining global serializability in lazy replication techniques, since the refresh transactions have to be executed at the replicas in the same serialization order. We discuss these further in relevant sections.

6.2.4 Distributed Techniques

Distributed techniques apply the update on the local copy at the site where the update transaction originates, and then the updates are propagated to the other replica sites. These are called distributed techniques since different transactions can update different copies of the same data item located at different sites. They are appropriate for collaborative applications with distributive decision/operation centers. They can more evenly distribute the load, and may provide the highest system availability if coupled with lazy propagation techniques.

A serious complication that arises in these systems is that different replicas of a data item may be updated at different sites (masters) concurrently. If distributed techniques are coupled by eager propagation methods, then the distributed concurrency control methods can adequately address the concurrent updates problem. However, if lazy propagation methods are used, then transactions may be executed in different orders at different sites causing non-1SR global history. Furthermore, various replicas will get out of sync. To manage these problems, a reconciliation method is applied involving undoing and redoing transactions in such a way that transaction execution is the same at each site. This is not an easy issue since reconciliation is generally application dependent.

6.3 Replication Protocols

In the previous section, we discussed two dimensions along which update management techniques can be classified. These dimensions are orthogonal; therefore, four

combinations are possible: eager centralized, eager distributed, lazy centralized, and lazy distributed. We discuss each of these alternatives in this section. For simplicity of exposition, we assume a fully replicated database, which means that all update transactions are global. We further assume that each site implements a 2PL-based concurrency control technique.

6.3.1 Eager Centralized Protocols

In eager centralized replica control, a master site controls the operations on a data item. These protocols are coupled with strong consistency techniques, so that updates to a logical data item are applied to all of its replicas within the context of the update transaction, which is committed using the 2PC protocol (although non-2PC alternatives exist as we discuss shortly). Consequently, once the update transaction completes, all replicas have the same values for the updated data items (i.e., mutually consistent), and the resulting global history is 1SR.

The two design parameters that we discussed earlier determine the specific implementation of eager centralized replica protocols: where updates are performed, and degree of replication transparency. The first parameter, which was discussed in Sect. 6.2.3, refers to whether there is a single master site for all data items (single master), or different master sites for each, or, more likely, for a group of data items (primary copy). The second parameter indicates whether each application knows the location of the master copy (limited application transparency) or whether it can rely on its local TM for determining the location of the master copy (full replication transparency).

6.3.1.1 Single Master with Limited Replication Transparency

The simplest case is to have a single master for the entire database (i.e., for all data items) with limited replication transparency so that user applications know the master site. In this case, global update transactions (i.e., those that contain at least one $W(x)$ operation, where x is a replicated data item) are submitted directly to the master site—more specifically, to the TM at the master site. At the master, each $R(x)$ operation is performed on the master copy (i.e., $R(x)$ is converted to $R(x_M)$, where M signifies master copy) and executed as follows: a read lock is obtained on x_M, the read is performed, and the result is returned to the user. Similarly, each $W(x)$ causes an update of the master copy [i.e., executed as $W(x_M)$] by first obtaining a write lock and then performing the write operation. The master TM then forwards the Write to the slave sites either synchronously or in a deferred fashion (Fig. 6.1). In either case, it is important to propagate updates such that conflicting updates are executed at the slaves in the same order they are executed at the master. This can be achieved by timestamping or by some other ordering scheme.

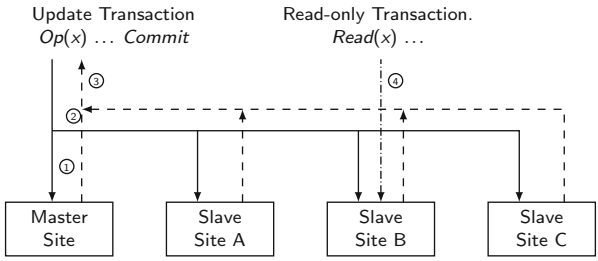

Fig. 6.1 Eager single master replication protocol actions. (1) A Write is applied on the master copy; (2) Write is then propagated to the other replicas; (3) Updates become permanent at commit time; (4) Read-only transaction's Read goes to any slave copy

The user application may submit a read-only transaction (i.e., all operations are Read) to any slave site. The execution of read-only transactions at the slaves can follow the process of centralized concurrency control algorithms, such as C2PL (Algorithms 5.1–5.3), where the centralized lock manager resides at the master replica site. Implementations within C2PL require minimal changes to the TM at the nonmaster sites, primarily to deal with the Write operations as described above, and its consequences (e.g., in the processing of Commit command). Thus, when a slave site receives a Read operation (from a read-only transaction), it forwards it to the master site to obtain a read lock. The Read can then be executed at the master and the result returned to the application, or the master can simply send a "lock granted" message to the originating site, which can then execute the Read on the local copy.

It is possible to reduce the load on the master by performing the Read on the local copy without obtaining a read lock from the master site. Whether synchronous or deferred propagation is used, the local concurrency control algorithm ensures that the local read–write conflicts are properly serialized, and since the Write operations can only be coming from the master as part of update propagation, local write–write conflicts will not occur as the propagation transactions are executed in each slave in the order dictated by the master. However, a Read may read data item values at a slave either before an update is installed or after. The fact that a Read from one transaction at one slave site may read the value of one replica before an update, while another Read from another transaction reads another replica at another slave after the same update is inconsequential from the perspective of ensuring global 1SR histories. This is demonstrated by the following example.

Example 6.3 Consider a data item x whose master site is at site A with slaves at sites B and C. Consider the following three transactions:

T_1:	Write(x)	T_2:	Read(x)	T_3:	Read(x)
	Commit		Commit		Commit

Assume that T_2 is sent to slave at site B and T_3 to slave at site C. Assume that T_2 reads x at B [$R_2(x_B)$] before T_1's update is applied at B, while T_3 reads x at C

$[R_3(x_C)]$ after T_1's update at C. Then the histories generated at the two slaves will be as follows:

$$H_B = \{R_2(x), C_2, W_1(x), C_1\}$$
$$H_C = \{W_1(x), C_1, R_3(x), C_3\}$$

The serialization order at site B is $T_2 \rightarrow T_1$, while at site C it is $T_1 \rightarrow T_3$. The global serialization order, therefore, is $T_2 \rightarrow T_1 \rightarrow T_3$, which is fine. Therefore the history is 1SR. ♦

Consequently, if this approach is followed, read transactions may read data that are concurrently updated at the master, but the global history will still be 1SR.

In this alternative protocol, when a slave site S_i receives a $R(x)$, it obtains a local read lock, reads from its local copy [i.e., $R(x_i)$], and returns the result to the user application; this can only come from a read-only transaction. When it receives a $W(x)$, if this is coming from the master site, then it performs it on the local copy [i.e., $W_i(x_i)$]. If it is from a user application, then it rejects $W(x)$, since this is obviously an error given that update transactions have to be submitted to the master site.

These alternatives of a single master eager centralized protocol are simple to implement. One important issue to address is how one recognizes a transaction as "update" or "read-only"—it may be possible to do this by explicit declaration within the Begin_transaction command.

6.3.1.2 Single Master with Full Replication Transparency

Single master eager centralized protocols require each user application to know the master site, and they put significant load on the master that has to deal with (at least) the Read operations within update transactions as well as acting as the coordinator for these transactions during 2PC execution. These issues can be addressed, to some extent, by involving, in the execution of the update transactions, the TM at the site where the application runs. Thus, the update transactions are not submitted to the master, but to the TM at the site where the application runs (since they do not need to know the master). This TM can act as the coordinating TM for both update and read-only transactions. Applications can simply submit their transactions to their local TM, providing full transparency.

There are alternatives to implementing full transparency—the coordinating TM may only act as a "router," forwarding each operation directly to the master site. The master site can then execute the operations locally (as described above) and return the results to the application. Although this alternative implementation provides full transparency and has the advantage of being simple to implement, it does not address the overloading problem at the master. An alternative implementation may be as follows:

1. The coordinating TM sends each operation, as it gets it, to the central (master) site. This requires no change to the C2PL-TM algorithm (Algorithm 5.1).
2. If the operation is a $R(x)$, then the centralized lock manager (C2PL-LM in Algorithm 5.2) can proceed by setting a read lock on its copy of x (call it x_M) on behalf of this transaction and informs the coordinating TM that the read lock is granted. The coordinating TM can then forward the $R(x)$ to any slave site that holds a replica of x [i.e., converts it to a $R(x_i)$]. The read can then be carried out by the data processor (DP) at that slave.
3. If the operation is a $W(x)$, then the centralized lock manager (master) proceeds as follows:

   (a) It first sets a write lock on its copy of x_M.
   (b) It then calls its local DP to perform $W(x_M)$ on its own copy.
   (c) Finally, it informs the coordinating TM that the write lock is granted.

   The coordinating TM, in this case, sends the $W(x)$ to all the slaves where a copy of x exists; the DPs at these slaves apply the Write to their local copies.

The fundamental difference in this case is that the master site does not deal with Read or with the coordination of the updates across replicas. These are left to the TM at the site where the user application runs.

It is straightforward to see that this algorithm guarantees that the histories are 1SR since the serialization orders are determined at a single master (similar to centralized concurrency control algorithms). It is also clear that the algorithm follows the ROWA protocol, as discussed above—since all the copies are ensured to be up-to-date when an update transaction completes, a Read can be performed on any copy.

To demonstrate how eager algorithms combine replica control and concurrency control, we show the Transaction Management algorithm for the coordinating TM (Algorithm 6.1) and the Lock Management algorithm for the master site (Algorithm 6.2). We show only the revisions to the centralized 2PL algorithms (Algorithms 5.1 and 5.2 in Chap. 5).

Note that in the algorithm fragments that we have given, the LM simply sends back a "Lock granted" message and not the result of the update operation. Consequently, when the update is forwarded to the slaves by the coordinating TM, they need to execute the update operation themselves. This is sometimes referred to as *operation transfer*. The alternative is for the "Lock granted" message to include the result of the update computation, which is then forwarded to the slaves who simply need to apply the result and update their logs. This is referred to as *state transfer*. The distinction may seem trivial if the operations are simply in the form $W(x)$, but recall that this Write operation is an abstraction; each update operation may require the execution of an SQL expression, in which case the distinction is quite important.

The above implementation of the protocol relieves some of the load on the master site and alleviates the need for user applications to know the master. However, its implementation is more complicated than the first alternative we discussed. In

Algorithm 6.1: Eager Single Master Modifications to C2PL-TM

```
begin
    ⋮
    if lock request granted then
        if op.Type = W then
            S ← set of all sites that are slaves for the data item
        else
            S ← any one site which has a copy of data item
        end if
        DP_S(op)                                   {send operation to all sites in set S}
    else
        inform user about the termination of transaction
    end if
    ⋮
end
```

Algorithm 6.2: Eager Single Master Modifications to C2PL-LM

```
begin
    ⋮
    switch op.Type do
        case R or W do                             {lock request; see if it can be granted}
            find the lock unit lu such that op.arg ⊆ lu ;
            if lu is unlocked or lock mode of lu is compatible with op.Type then
                set lock on lu in appropriate mode on behalf of transaction op.tid ;
                if op.Type = W then
                    DP_M(op)                       {call local DP (M for "master") with operation}
                send "Lock granted" to coordinating TM of transaction
            else
                put op on a queue for lu
            end if
        end case
        ⋮
    end switch
end
```

particular, now the TM at the site where transactions are submitted has to act as the 2PC coordinator and the master site becomes a participant. This requires some care in revising the algorithms at these sites.

6.3.1.3 Primary Copy with Full Replication Transparency

Let us now relax the requirement that there is one master for all data items; each data item can have a different master. In this case, for each replicated data item, one of the replicas is designated as the *primary copy*. Consequently, there is no single

master to determine the global serialization order, so more care is required. In the case of fully replicated databases, any replica can be primary copy for a data item; however, for partially replicated databases, limited replication transparency option only makes sense if an update transaction accesses only data items whose primary sites are at the same site. Otherwise, the application program cannot forward the update transactions to one master; it will have to do it operation-by-operation, and, furthermore, it is not clear which primary copy master would serve as the coordinator for 2PC execution. Therefore, the reasonable alternative is the full transparency support, where the TM at the application site acts as the coordinating TM and forwards each operation to the primary site of the data item that it acts on. Figure 6.2 depicts the sequence of operations in this case where we relax our previous assumption of fully replication. Site A is the master for data item x and sites B and C hold replicas (i.e., they are slaves); similarly data item y's master is site C with slave sites B and D.

Recall that this version still applies the updates to all the replicas within transactional boundaries, requiring integration with concurrency control techniques. A very early proposal is the *primary copy two-phase locking* (PC2PL) algorithm proposed for the prototype distributed version of INGRES. PC2PL is a straightforward extension of the single master protocol discussed above in an attempt to counter the latter's potential performance problems. Basically, it implements lock managers at a number of sites and makes each lock manager responsible for managing the locks for a given set of lock units for which it is the master site. The transaction managers then send their lock and unlock requests to the lock managers that are responsible for that specific lock unit. Thus the algorithm treats one copy of each data item as its primary copy.

As a combined replica control/concurrency control technique, primary copy approach demands a more sophisticated directory at each site, but it also improves the previously discussed approaches by reducing the load of the master site without causing a large amount of communication among the transaction managers and lock managers.

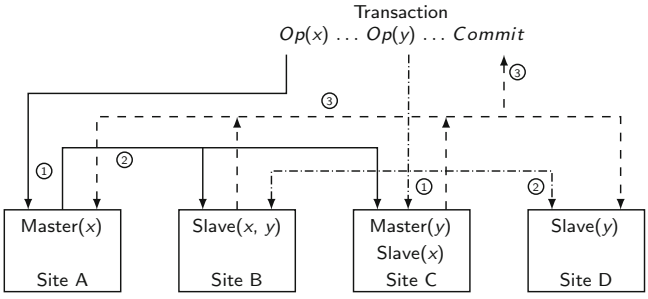

Fig. 6.2 Eager primary copy replication protocol actions. (1) Operations (Read or Write) for each data item are routed to that data item's master and a Write is first applied at the master; (2) Write is then propagated to the other replicas; (3) Updates become permanent at commit time

6.3.2 Eager Distributed Protocols

In eager distributed replica control, the updates can originate anywhere, and they are first applied on the local replica, then the updates are propagated to other replicas. If the update originates at a site where a replica of the data item does not exist, it is forwarded to one of the replica sites, which coordinates its execution. Again, all of these are done within the context of the update transaction, and when the transaction commits, the user is notified and the updates are made permanent. Figure 6.3 depicts the sequence of operations for one logical data item x with copies at sites A, B, C, and D, and where two transactions update two different copies (at sites A and D).

As can be clearly seen, the critical issue is to ensure that concurrent conflicting Write operations initiated at different sites are executed in the same order at every site where they execute together (of course, the local executions at each site also have to be serializable). This is achieved by means of the concurrency control techniques that are employed at each site. Consequently, read operations can be performed on any copy, but writes are performed on all copies within transactional boundaries (e.g., ROWA) using a concurrency control protocol.

6.3.3 Lazy Centralized Protocols

Lazy centralized replication algorithms are similar to eager centralized replication ones in that the updates are first applied to a master replica and then propagated to the slaves. The important difference is that the propagation does not take place within the update transaction, but after the transaction commits as a separate refresh transaction. Consequently, if a slave site performs a $R(x)$ operation on its local copy, it may read stale (nonfresh) data, since x may have been updated at the master, but the update may not have yet been propagated to the slaves.

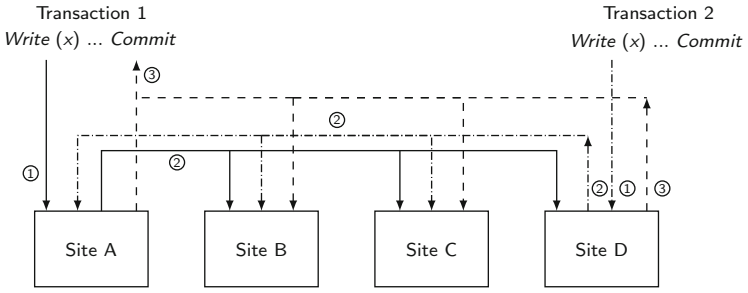

Fig. 6.3 Eager distributed replication protocol actions. (1) Two Write operations are applied on two local replicas of the same data item; (2) The Write operations are independently propagated to the other replicas; (3) Updates become permanent at commit time (shown only for Transaction 1)

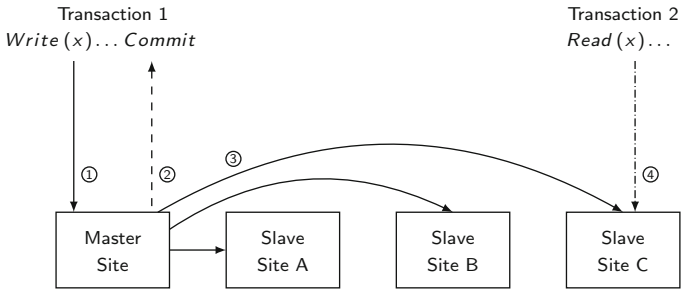

Fig. 6.4 Lazy single master replication protocol actions. (1) Update is applied on the local replica; (2) Transaction commit makes the updates permanent at the master; (3) Update is propagated to the other replicas in refresh transactions; (4) Transaction 2 reads from local copy

6.3.3.1 Single Master with Limited Transparency

In this case, the update transactions are submitted and executed directly at the master site (as in the eager single master); once the update transaction commits, the refresh transaction is sent to the slaves. The sequence of execution steps is as follows: (1) an update transaction is first applied to the master replica, (2) the transaction is committed at the master, and then (3) the refresh transaction is sent to the slaves (Fig. 6.4).

When a slave site receives a $R(x)$, it reads from its local copy and returns the result to the user. Notice that, as indicated above, its own copy may not be up-to-date if the master is being updated and the slave has not yet received and executed the corresponding refresh transaction. A $W(x)$ received by a slave is rejected (and the transaction aborted), as this should have been submitted directly to the master site. When a slave receives a refresh transaction from the master, it applies the updates to its local copy. When it receives a Commit or Abort (Abort can happen for only locally submitted read-only transactions), it locally performs these actions.

The case of primary copy with limited transparency is similar, so we do not discuss it in detail. Instead of going to a single master site, $W(x)$ is submitted to the primary copy of x; the rest is straightforward.

How can it be ensured that the refresh transactions can be applied at all of the slaves in the same order? In this architecture, since there is a single master copy for all data items, the ordering can be established by simply using timestamps. The master site would attach a timestamp to each refresh transaction according to the commit order of the actual update transaction, and the slaves would apply the refresh transactions in timestamp order.

A similar approach may be followed in the primary copy, limited transparency case. In this case, a site contains slave copies of a number of data items, causing it to get refresh transactions from multiple masters. The execution of these refresh transactions need to be ordered the same way at all of the involved slaves to ensure

that the database states eventually converge. There are a number of alternatives that can be followed.

One alternative is to assign timestamps such that refresh transactions issued from different masters have different timestamps (by appending the site identifier to a monotonic counter at each site). Then the refresh transactions at each site can be executed in their timestamp order. However, those that come out of order cause difficulty. In traditional timestamp-based techniques discussed in Chap. 5, these transactions would be aborted; however, in lazy replication, this is not possible since the transaction has already been committed at the primary copy site. The only possibility is to run a compensating transaction (which, effectively, aborts the transaction by rolling back its effects) or to perform update reconciliation that will be discussed shortly. The issue can be addressed by a more careful study of the resulting histories. An approach is to use a serialization graph approach that builds a *replication graph* whose nodes consist of transactions (T) and sites (S) and an edge $\langle T_i, S_j \rangle$ exists in the graph if and only if T_i performs a Write on a (replicated) physical copy that is stored at S_j. When an operation (op_k) is submitted, the appropriate nodes (T_k) and edges are inserted into the replication graph, which is checked for cycles. If there is no cycle, then the execution can proceed. If a cycle is detected and it involves a transaction that has committed at the master, but whose refresh transactions have not yet committed at all of the involved slaves, then the current transaction (T_k) is aborted (to be restarted later) since its execution would cause the history to be non-1SR. Otherwise, T_k can wait until the other transactions in the cycle are completed (i.e., they are committed at their masters and their refresh transactions are committed at all of the slaves). When a transaction is completed in this manner, the corresponding node and all of its incident edges are removed from the replication graph. This protocol is proven to produce 1SR histories. An important issue is the maintenance of the replication graph. If it is maintained by a single site, then this becomes a centralized algorithm. We leave the distributed construction and maintenance of the replication graph as an exercise.

Another alternative is to rely on the group communication mechanism provided by the underlying communication infrastructure (if it can provide it). We discuss this alternative in Sect. 6.4.

Recall from Sect. 6.3.1 that, in the case of partially replicated databases, eager primary copy with limited replication transparency approach makes sense if the update transactions access only data items whose master sites are the same, since the update transactions are run completely at a master. The same problem exists in the case of lazy primary copy, limited replication approach. The issue that arises in both cases is how to design the distributed database so that meaningful transactions can be executed. This problem has been studied within the context of lazy protocols and a primary site selection algorithm was proposed that, given a set of transactions, a set of sites, and a set of data items, finds a primary site assignment to these data items (if one exists) such that the set of transactions can be executed to produce a 1SR global history.

6.3.3.2 Single Master or Primary Copy with Full Replication Transparency

We now turn to alternatives that provide full transparency by allowing (both read and update) transactions to be submitted at any site and forwarding their operations to either the single master or to the appropriate primary master site. This is tricky and involves two problems: the first is that, unless one is careful, 1SR global history may not be guaranteed; the second problem is that a transaction may not see its own updates. The following two examples demonstrate these problems.

Example 6.4 Consider the single master scenario and two sites M and B, where M holds the master copies of x and y and B holds their slave copies. Now consider the following two transactions: T_1 submitted at site B, while transaction T_2 submitted at site M:

T_1:	Read(x)	T_2:	Write(x)
	Write(y)		Write(y)
	Commit		Commit

One way these would be executed under full transparency is as follows. T_2 would be executed at site M since it contains the master copies of both x and y. Sometime after it commits, refresh transactions for its Write operations are sent to site B to update the slave copies. On the other hand, T_1 would read the local copy of x at site B $[R_1(x_B)]$, but its $W_1(x)$ would be forwarded to x's master copy, which is at site M. Some time after $W_1(x)$ is executed at the master site and commits there, a refresh transaction would be sent back to site B to update the slave copy. The following is a possible sequence of steps of execution (Fig. 6.5):

1. $R_1(x)$ is submitted at site B, where it is performed $[R_1(x_B)]$;
2. $W_2(x)$ is submitted at site M, and it is executed $[W_2(x_M)]$;
3. $W_2(y)$ is submitted at site M, and it is executed $[W_2(y_B)]$;
4. T_2 submits its Commit at site M and commits there;
5. $W_1(x)$ is submitted at site B; since the master copy of x is at site M, the Write is forwarded to M;
6. $W_1(x)$ is executed at site M $[W_1(x_M)]$; and the confirmation is sent back to site B;
7. T_1 submits Commit at site B, which forwards it to site M; it is executed there and B is informed of the commit where T_1 also commits;
8. Site M now sends refresh transaction for T_2 to site B where it is executed and commits;
9. Site M finally sends refresh transaction for T_1 to site B (this is for T_1's Write that was executed at the master), it is executed at B and commits.

The following two histories are now generated at the two sites where the superscript r on operations indicates that they are part of a refresh transaction:

$$H_B = \{W_2(x_M), W_2(y_M), C_2, W_1(y_M), C_1\}$$
$$H_B = \{R_1(x_B), C_1, W_2^r(x_B), W_2^r(y_B), C_2^r, W_1^r(x_B), C_1^r\}$$

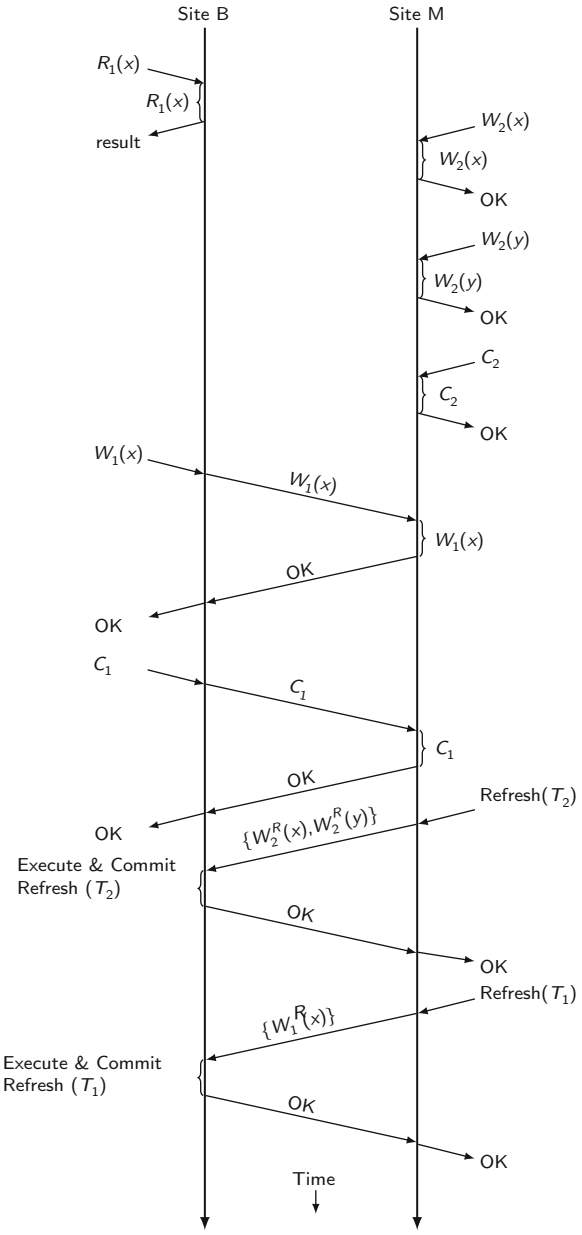

Fig. 6.5 Time sequence of executions of transactions

The resulting global history over the *logical* data items x and y is non-1SR. ♦

Example 6.5 Again consider a single master scenario, where site M holds the master copy of x and site D holds its slave. Consider the following simple transaction:

T_3: Write(x)
Read(x)
Commit

Following the same execution model as in Example 6.4, the sequence of steps would be as follows:

1. $W_3(x)$ is submitted at site D, which forwards it to site M for execution;
2. The Write is executed at M $[W_3(x_M)]$ and the confirmation is sent back to site D;
3. $R_3(x)$ is submitted at site D and is executed on the local copy $[R_3(x_D)]$;
4. T_3 submits commit at D, which is forwarded to M, executed there and a notification is sent back to site D, which also commits the transaction;
5. Site M sends a refresh transaction to site D for the $W_3(x)$ operation;
6. Site D executes the refresh transaction and commits it.

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 an algorithm 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 (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 6.4, master site M would not be able to assign an appropriate timestamp

[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.

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 and Write operations have to be executed at the master.

6.3.4 *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 (Fig. 6.6).

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.

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

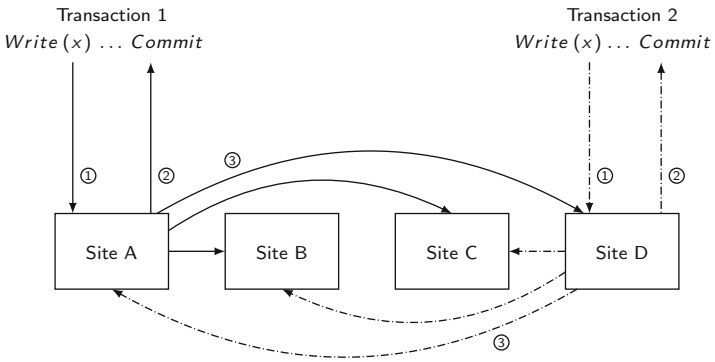

Fig. 6.6 Lazy distributed replication protocol actions. (1) Two updates are applied on two local replicas; (2) Transaction commit makes the updates permanent; (3) The updates are independently propagated to the other replicas

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.

6.4 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 together with nontraditional techniques for processing local transactions. 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, while the second one is a lazy centralized protocol.

The group communication-based eager distributed protocol 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 $R_i(x)$ operation is performed at the site where it is submitted (this is the master site for this transaction). A $W_i(x)$ operation 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 $W(x)$ 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

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

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.

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 Sect. 6.3.2. 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 protocol that 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 S_1 and S_2, respectively, then there are two pending queues, q_1 and q_2, corresponding to master sites S_1 and S_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 $delivery\_time = ts(new\_RT) + Max + \epsilon$. 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 holds: (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.

6.5 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.

6.5.1 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 Sect. 6.3.4. Before the merge, databases at multiple partitions diverge, but they are reconciled at merge time.

6.5.2 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, 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, two of which we will discuss. The first one is known as the *available copies protocol*. 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 that 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 nonserializable.
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 nonreplicated 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. This version is suitable wherever compensating transactions are acceptable (see Chap. 5).

The earliest voting algorithm (known as 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. This was revisited in Gifford's algorithm, which also works with partially replicated databases 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.

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 nonpartitioned 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.

6.6 Conclusion

In this chapter, we discussed different approaches to data replication and presented protocols that are appropriate under different circumstances. Each of the alternative protocols we have discussed has 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 multimaster 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 Chap. 8. We also defer to Chap. 7 replication issues that arise in multidatabase systems.

6.7 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 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].

Eventual consistency definition is from [Saito and Shapiro 2005], epsilon serializability is due to Pu and Leff [1991] and also discussed by Ramamritham and Pu [1995] and Wu et al. [1997]. 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, has been a topic of some study. Alternative techniques to ensure "better" freshness are discussed in [Pacitti et al. 1998, 1999, Pacitti and Simon 2000, Röhm et al. 2002, Pape et al. 2004, Akal et al. 2005, Bernstein et al. 2006].

Extension of snapshot isolation to replicated databases is due to Lin et al. [2005] and its use in replicated databases is discussed in [Plattner and Alonso 2004, Daudjee and Salem 2006]. RC-serializability as another weaker form of serializability is introduced by Bernstein et al. [2006]. Strong one-copy serializability is discussed in [Daudjee and Salem 2004] and strong snapshot isolation in [Daudjee and Salem 2006]—these prevent transaction inversion.

An early eager primary copy replication protocol has been implemented in distributed INGRES and described in [Stonebraker and Neuhold 1977].

In single master lazy replication approach, using a replication graph to dealing with ordering of refresh transactions is due to Breitbart and Korth [1997]. Dealing with deferred updates by finding appropriate primary site assignment for data items is due to Chundi et al. [1996].

Bernstein et al. [2006] propose a lazy replication algorithm with full transparency.

The use of group communication has been discussed in [Chockler et al. 2001, Stanoi et al. 1998, Kemme and Alonso 2000a,b, Patiño-Martínez et al. 2000, Jiménez-Peris et al. 2002]. The eager distributed protocol we discuss in Sect. 6.4 is due to Kemme and Alonso [2000b] and the lazy centralized one is due to Pacitti et al. [1999].

The available copies protocol in Sect. 6.5.2 is due to Bernstein and Goodman [1984] and Bernstein et al. [1987].

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]. 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]. The algorithm we present in Sect. 6.5.2 that overcomes the performance problems of Gifford's algorithm is by El Abbadi et al. [1985]. The comprehensive study we report in the same section that indicates the benefits of ROWA-A is [Jiménez-Peris et al. 2003]. 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 is 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 6.1 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 6.2 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 ordering of products, the payment of bills, and general information requests. There exist several tables to register the orders of a customer. Each order is registered 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) would 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 6.3 (*) 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 Sect. 6.3.3. Develop a method for distributed management of the replication graph.

Problem 6.4 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.

Chapter 7
Database Integration—Multidatabase Systems

Up to this point, we considered distributed DBMSs that are designed in a top-down fashion. In particular, Chap. 2 focuses on techniques for partitioning and allocating a database, while Chap. 4 focuses on distributed query processing over such a database. These techniques and approaches are suitable for tightly integrated, homogeneous distributed DBMSs. In this chapter, we focus on distributed databases that are designed in a bottom-up fashion—we referred to these as multidatabase systems in Chap. 1. In this case, a number of databases already exist, and the design task involves integrating them into one database. The starting point of bottom-up design is the set of individual local conceptual schemas (LCSs). The process consists of integrating local databases with their (local) schemas into a global database and generating a global conceptual schema (GCS) (also called the *mediated schema*). Querying over a multidatabase system is more complicated in that applications and users can either query using the GCS (or views defined on it) or through the LCSs since each existing local database may already have applications running on it. Therefore, the techniques required for query processing require adjustments to the approach we discussed in Chap. 4 although many of those techniques carry over.

Database integration, and the related problem of querying multidatabases, is only one part of the more general *interoperability* problem, which includes nondatabase data sources and interoperability at the application level in addition to the database level. We separate this discussion into three pieces: in this chapter, we focus on the database integration and querying issues, we discuss the concerns related to web data integration and access in Chap. 12, and we discuss the more general issue of integrating data from arbitrary data sources in Chap. 10 under the title *data lakes*.

This chapter consists of two main sections. In Sect. 7.1, we discuss database integration—the bottom-up design process. In Sect. 7.2 we discuss approaches to querying these systems.

M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*,
https://doi.org/10.1007/978-3-030-26253-2_7

7.1 Database Integration

Database integration can be either physical or logical. In the former, the source databases are integrated and the integrated database is *materialized*. These are known as *data warehouses*. The integration is aided by *extract–transform–load* (ETL) tools that enable extraction of data from sources, its transformation to match the GCS, and its loading (i.e., materialization). This process is depicted in Fig. 7.1. In logical integration, the global conceptual (or mediated) schema is entirely *virtual* and not materialized.

These two approaches are complementary and address differing needs. Data warehousing supports decision-support applications, which are commonly termed *Online Analytical Processing* (OLAP). Recall from Chap. 5 that OLAP applications analyze historical, summarized data coming from a number of operational databases through complex queries over potentially very large tables. Consequently, data warehouses gather data from a number of operational databases and materialize it. As updates happen on the operational databases, they are propagated to the data warehouse, which is known as *materialized view maintenance*.

By contrast, in logical data integration, the integration is only virtual and there is no materialized global database (see Fig. 1.13). The data resides in the operational databases and the GCS provides a virtual integration for querying over the multiple databases. In these systems, GCS may either be defined up-front and local databases (i.e., LCSs) mapped to it, or it may be defined bottom-up, by integrating parts of the LCSs of the local databases. Consequently, it is possible for the GCS not to capture

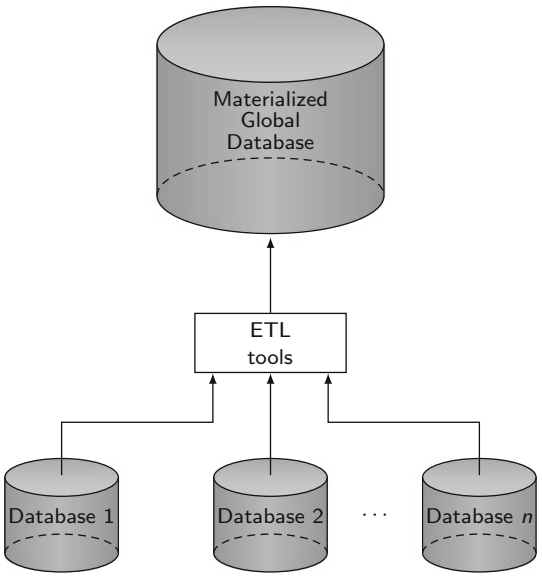

Fig. 7.1 Data warehouse approach

all of the information in each of the LCSs. User queries are posed over this global schema, which are then decomposed and shipped to the local operational databases for processing as is done in tightly integrated systems, with the main difference being the autonomy and potential heterogeneity of the local systems. These have important effects on query processing that we discuss in Sect. 7.2. Although there is ample work on transaction management in these systems, supporting global updates is quite difficult given the autonomy of the underlying operational DBMSs. Therefore, they are primarily read-only.

Logical data integration and the resulting systems are known by a variety of names; *data integration* and *information integration* are perhaps the most common terms used in literature although these generally refer to more than database integration and incorporate data from a variety of sources. In this chapter, we focus on the integration of autonomous and (possibly) heterogeneous databases; thus, we will use the term *database integration* or *multidatabase systems* (MDBSs).

7.1.1 Bottom-Up Design Methodology

Bottom-up design involves the process by which data from participating databases can be (physically or logically) integrated to form a single cohesive global database. As noted above, in some cases, the global conceptual (or mediated) schema is defined first, in which case the bottom-up design involves mapping LCSs to this schema. In other cases, the GCS is defined as an integration of parts of LCSs. In this case, the bottom-up design involves both the generation of the GCS and the mapping of individual LCSs to this GCS.

If the GCS is defined upfront, the relationship between the GCS and the LCSs can be of two fundamental types: local-as-view and global-as-view. In local-as-view (LAV) systems, the GCS definition exists, and each LCS is treated as a view definition over it. In global-as-view systems (GAV), on the other hand, the GCS is defined as a set of views over the LCSs. These views indicate how the elements of the GCS can be derived, when needed, from the elements of LCSs. One way to think of the difference between the two is in terms of the results that can be obtained from each system . In GAV, the query results are constrained to the set of objects that are defined in the GCS, although the local DBMSs may be considerably richer (Fig. 7.2a). In LAV, on the other hand, the results are constrained by the objects in the local DBMSs, while the GCS definition may be richer (Fig. 7.2b). Thus, in LAV systems, it may be necessary to deal with incomplete answers. A combination of these two approaches has also been proposed as global-local-as-view (GLAV) where the relationship between GCS and LCSs is specified using both LAV and GAV.

Bottom-up design occurs in two general steps (Fig. 7.3): *schema translation* (or simply *translation*) and *schema generation*. In the first step, the component database schemas are translated to a common intermediate canonical representation (InS_1, InS_2,. . ., InS_n). The use of a canonical representation facilitates the trans-

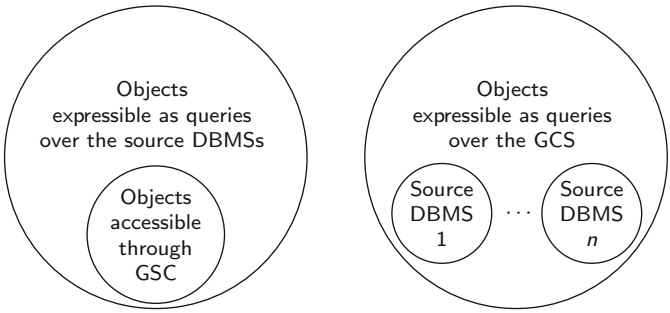

Fig. 7.2 GAV and LAV mappings (based on [Koch 2001])

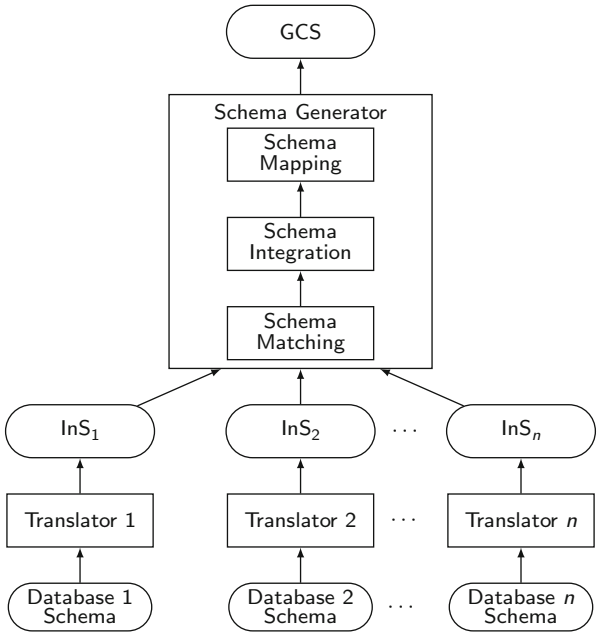

Fig. 7.3 Database integration process

lation process by reducing the number of translators that need to be written. The choice of the canonical model is important. As a principle, it should be one that is sufficiently expressive to incorporate the concepts available in all the databases that will later be integrated. Alternatives that have been used include the entity-relationship model, object-oriented model, or a graph that may be simplified to a trie or XML. In this chapter, we will simply use the relational model as our canonical data model despite its known deficiencies in representing rich semantic concepts. This choice does not affect in any fundamental way the discussion of the major

issues of data integration. In any case, we will not discuss the specifics of translating various data models to relational; this can be found in many database textbooks.

Clearly, the translation step is necessary only if the component databases are heterogeneous and local schemas are defined using different data models. There has been some work on the development of system federation, in which systems with similar data models are integrated together (e.g., relational systems are integrated into one conceptual schema and, perhaps, object databases are integrated to another schema) and these integrated schemas are "combined" at a later stage (e.g., AURORA project). In this case, the translation step is delayed, providing increased flexibility for applications to access underlying data sources in a manner that is suitable for their needs.

In the second step of bottom-up design, the intermediate schemas are used to generate a GCS. The schema generation process consists of the following steps:

1. Schema matching to determine the syntactic and semantic correspondences among the translated LCS elements or between individual LCS elements and the predefined GCS elements (Sect. 7.1.2).
2. Integration of the common schema elements into a global conceptual (mediated) schema if one has not yet been defined (Sect. 7.1.3).
3. Schema mapping that determines how to map the elements of each LCS to the other elements of the GCS (Sect. 7.1.4).

It is also possible that the schema mapping step be divided into two phases: mapping constraint generation and transformation generation. In the first phase, given correspondences between two schemas, a transformation function such as a query or view definition over the source schema is generated that would "populate" the target schema. In the second phase, an executable code is generated corresponding to this transformation function that would actually generate a target database consistent with these constraints. In some cases, the constraints are implicitly included in the correspondences, eliminating the need for the first phase.

Example 7.1 To facilitate our discussion of global schema design in multidatabase systems, we will use an example that is an extension of the engineering database we have been using throughout the book. To demonstrate both phases of the database integration process, we introduce some data model heterogeneity into our example.

Consider two organizations, each with their own database definitions. One is the (relational) database example that we introduced in Chap. 2. We repeat that definition in Fig. 7.4 for completeness. The second database also defines similar data, but is specified according to the entity-relationship (E-R) data model as depicted in Fig. 7.5.[1]

We assume that the reader is familiar with the entity-relationship data model. Therefore, we will not describe the formalism, except to make the following points regarding the semantics of Fig. 7.5. This database is similar to the relational

[1] In this chapter, we continue our notation of typesetting relation names in `typewriter` font, but we will use normal font for E-R model components to be able to easily differentiate them.

EMP(ENO, ENAME, TITLE)
PROJ(PNO, PNAME, BUDGET, LOC)
ASG(ENO, PNO, RESP, DUR)
PAY(TITLE, SAL)

Fig. 7.4 Relational engineering database representation

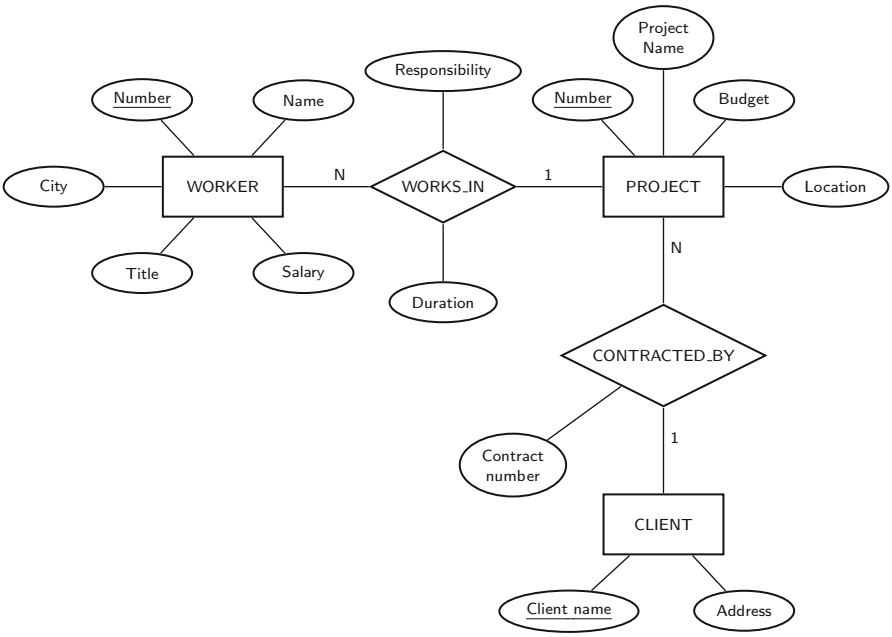

Fig. 7.5 Entity-relationship database

engineering database definition of Fig. 7.4, with one significant difference: it also maintains data about the clients for whom the projects are conducted. The rectangular boxes in Fig. 7.5 represent the entities modeled in the database, and the diamonds indicate a relationship between the entities to which they are connected. The relationship type is indicated around the diamonds. For example, the CONTRACTED-BY relation is a many-to-one from the PROJECT entity to the CLIENT entity (e.g., each project has a single client, but each client can have many projects). Similarly, the WORKS-IN relationship indicates a many-to-many relationship between the two connected relations. The attributes of entities and the relationships are shown as ellipses. ♦

Example 7.2 The mapping of the E-R model to the relational model is given in Fig. 7.6. Note that we have renamed some of the attributes in order to ensure name uniqueness. ♦

```
WORKER(WNUMBER, NAME, TITLE, SALARY,CITY)
PROJECT(PNUMBER, PNAME, BUDGET)
CLIENT(CNAME, ADDRESS)
WORKS_IN(WNUMBER, PNUMBER, RESPONSIBILITY, DURATION)
CONTRACTED_BY(PNUMBER, CNAME, CONTRACTNO)
```

Fig. 7.6 Relational mapping of E-R schema

7.1.2 Schema Matching

Given two schemas, schema matching determines for each concept in one schema what concept in the other matches it. As discussed earlier, if the GCS has already been defined, then one of these schemas is typically the GCS, and the task is to match each LCS to the GCS. Otherwise, matching is done over two LCSs. The matches that are determined in this phase are then used in schema mapping to produce a set of directed mappings, which, when applied to the source schema, would map its concepts to the target schema.

The matches that are defined or discovered during schema matching are specified as a set of rules where each rule (r) identifies a *correspondence* (c) between two elements, a *predicate* (p) that indicates when the correspondence may hold, and a *similarity value* (s) between the two elements identified in the correspondence. A correspondence may simply identify that two concepts are similar (which we will denote by $\approx$) or it may be a function that specifies that one concept may be derived by a computation over the other one (for example, if the budget value of one project is specified in US dollars, while the other one is specified in Euros, the correspondence may specify that one is obtained by multiplying the other one with the appropriate exchange rate). The predicate is a condition that qualifies the correspondence by specifying when it might hold. For example, in the budget example specified above, p may specify that the rule holds only if the location of one project is in US, while the other one is in the Euro zone. The similarity value for each rule can be specified or calculated. Similarity values are real values in the range [0,1]. Thus, a set of matches can be defined as $M = \{r\}$, where $r = \langle c, p, s \rangle$.

As indicated above, correspondences may either be discovered or specified. As much as it is desirable to automate this process, there are many complicating factors. The most important is schema heterogeneity, which refers to the differences in the way real-world phenomena are captured in different schemas. This is a critically important issue, and we devote a separate section to it (Sect. 7.1.2.1). Aside from schema heterogeneity, other issues that complicate the matching process are the following:

- *Insufficient schema and instance information:* Matching algorithms depend on the information that can be extracted from the schema and the existing data instances. In some cases there may be ambiguity due to the insufficient

information provided about these items. For example, using short names or ambiguous abbreviations for concepts, as we have done in our examples, can lead to incorrect matching.
- *Unavailability of schema documentation:* In most cases, the database schemas are not well documented or not documented at all. Quite often, the schema designer is no longer available to guide the process. The lack of these vital information sources adds to the difficulty of matching.
- *Subjectivity of matching:* Finally, it is important to recognize that matching schema elements can be highly subjective; two designers may not agree on a single "correct" mapping. This makes the evaluation of a given algorithm's accuracy significantly difficult.

Nevertheless, algorithmic approaches have been developed to the matching problem, which we discuss in this section. A number of issues affect the particular matching algorithm. The more important ones are the following:

- *Schema versus instance matching.* So far in this chapter, we have been focusing on schema integration; thus, our attention has naturally been on matching concepts of one schema to those of another. A large number of algorithms have been developed that work on schema elements. There are others, however, that have focused instead on the data instances or a combination of schema information and data instances. The argument is that considering data instances can help alleviate some of the semantic issues discussed above. For example, if an attribute name is ambiguous, as in "contact-info," then fetching its data may help identify its meaning; if its data instances have the phone number format, then obviously it is the phone number of the contact agent, while long strings may indicate that it is the contact agent name. Furthermore, there are a large number of attributes, such as postal codes, country names, email addresses, that can be defined easily through their data instances.

  Matching that relies solely on schema information may be more efficient, because it does not require a search over data instances to match the attributes. Furthermore, this approach is the only feasible one when few data instances are available in the matched databases, in which case learning may not be reliable. However, in some cases, e.g., peer-to-peer systems (see Chap. 9), there may not be a schema, in which case instance-based matching is the only appropriate approach.
- *Element-level vs. structure-level.* Some matching algorithms operate on individual schema elements, while others also consider the structural relationships between these elements. The basic concept of the element-level approach is that most of the schema semantics are captured by the elements' names. However, this may fail to find complex mappings that span multiple attributes. Match algorithms that also consider structure are based on the belief that, normally, the structures of matchable schemas tend to be similar.
- *Matching cardinality.* Matching algorithms exhibit various capabilities in terms of cardinality of mappings. The simplest approaches use 1:1 mapping, which means that each element in one schema is matched with exactly one element in

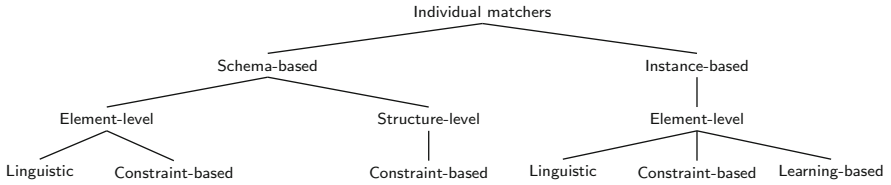

Fig. 7.7 Taxonomy of schema matching techniques

the other schema. The majority of proposed algorithms belong to this category, because problems are greatly simplified in this case. Of course there are many cases where this assumption is not valid. For example, an attribute named "Total price" could be mapped to the sum of two attributes in another schema named "Subtotal" and "Taxes." Such mappings require more complex matching algorithms that consider 1:M and N:M mappings.

These criteria, and others, can be used to come up with a taxonomy of matching approaches. According to this taxonomy (which we will follow in this chapter with some modifications), the first level of separation is between schema-based matchers versus instance-based matchers (Fig. 7.7). Schema-based matchers can be further classified as element-level and structure-level, while for instance-based approaches, only element-level techniques are meaningful. At the lowest level, the techniques are characterized as either linguistic or constraint-based. It is at this level that fundamental differences between matching algorithms are exhibited and we focus on these algorithms in the remainder, discussing linguistic approaches in Sect. 7.1.2.2, constraint-based approaches in Sect. 7.1.2.3, and learning-based techniques in Sect. 7.1.2.4. These are referred as *individual matcher* approaches, and their combinations are possible by developing either *hybrid matchers* or *composite matchers* (Sect. 7.1.2.5).

7.1.2.1 Schema Heterogeneity

Schema matching algorithms deal with both structural heterogeneity and semantic heterogeneity among the matched schemas. We discuss these in this section before presenting the different match algorithms.

Structural conflicts occur in four possible ways: as *type conflicts*, *dependency conflicts*, *key conflicts*, or *behavioral conflicts*. Type conflicts occur when the same object is represented by an attribute in one schema and an entity (relation) in another. Dependency conflicts occur when different relationship modes (e.g., one-to-one versus many-to-many) are used to represent the same thing in different schemas. Key conflicts occur when different candidate keys are available and different primary keys are selected in different schemas. Behavioral conflicts are implied by the modeling mechanism. For example, deleting the last item from one

database may cause the deletion of the containing entity (i.e., deletion of the last employee causes the dissolution of the department).

Example 7.3 We have two structural conflicts in the running example of this chapter. The first is a type conflict involving clients of projects. In the schema of Fig. 7.5, the client of a project is modeled as an entity. In the schema of Fig. 7.4, however, the client is included as an attribute of the PROJ entity.

The second structural conflict is a dependency conflict involving the WORKS_IN relationship in Fig. 7.5 and the ASG relation in Fig. 7.4. In the former, the relationship is many-to-one from the WORKER to the PROJECT, whereas in the latter, the relationship is many-to-many. ♦

Structural differences among schemas are important, but their identification and resolution is not sufficient. Schema matching has to take into account the (possibly different) semantics of the schema concepts. This is referred to as *semantic heterogeneity*, which is a fairly loaded term without a clear definition. It basically refers to the differences among the databases that relate to the meaning, interpretation, and intended use of data. There are attempts to formalize semantic heterogeneity and to establish its link to structural heterogeneity; we will take a more informal approach and discuss some of the semantic heterogeneity issues intuitively. The following are some of these problems that the match algorithms need to deal with.

- *Synonyms*, *homonyms*, *hypernyms*. Synonyms are multiple terms that all refer to the same concept. In our database example, PROJ relation and PROJECT entity refer to the same concept. Homonyms, on the other hand, occur when the same term is used to mean different things in different contexts. Again, in our example, BUDGET may refer to the gross budget in one database and it may refer to the net budget (after some overhead deduction) in another, making their simple comparison difficult. Hypernym is a term that is more generic than a similar word. Although there is no direct example of it in the databases we are considering, the concept of a Vehicle in one database is a hypernym for the concept of a Car in another (incidentally, in this case, Car is a *hyponym* of Vehicle). These problems can be addressed by the use of *domain ontologies* that define the organization of concepts and terms in a particular domain.
- *Different ontology:* Even if domain ontologies are used to deal with issues in one domain, it is quite often the case that schemas from different domains may need to be matched. In this case, one has to be careful of the meaning of terms across ontologies, as they can be highly domain dependent. For example, an attribute called LOAD may imply a measure of resistance in an electrical ontology, but in a mechanical ontology, it may represent a measure of weight.
- *Imprecise wording:* Schemas may contain ambiguous names. For example, the LOCATION (from E-R) and LOC (from relational) attributes in our example database may refer to the full address or just part of it. Similarly, an attribute named "contact-info" may imply that the attribute contains the name of the contact agent or his/her telephone number. These types of ambiguities are common.

7.1.2.2 Linguistic Matching Approaches

Linguistic matching approaches, as the name implies, use element names and other textual information (such as textual descriptions/annotations in schema definitions) to perform matches among elements. In many cases, they may use external sources, such as thesauri, to assist in the process.

Linguistic techniques can be applied in both schema-based approaches and instance-based ones. In the former case, similarities are established among schema elements, whereas in the latter, they are specified among elements of individual data instances. To focus our discussion, we will mostly consider schema-based linguistic matching approaches, briefly mentioning instance-based techniques. Consequently, we will use the notation $\langle$SC1.element-1 $\approx$ SC2.element-2, $p, s\rangle$ to represent that element-1 in schema SC1 corresponds to element-2 in schema SC2 if predicate p holds, with a similarity value of s. Matchers use these rules and similarity values to determine the similarity value of schema elements.

Linguistic matchers that operate at the schema element-level typically deal with the names of the schema elements and handle cases such as synonyms, homonyms, and hypernyms. In some cases, the schema definitions can have annotations (natural language comments) that may be exploited by the linguistic matchers. In the case of instance-based approaches, linguistic matchers focus on information retrieval techniques such as word frequencies, key terms, etc. In these cases, the matchers "deduce" similarities based on these information retrieval measures.

Schema linguistic matchers use a set of linguistic (also called terminological) rules that can be handcrafted or may be "discovered" using auxiliary data sources such as thesauri, e.g., WordNet. In the case of handcrafted rules, the designer needs to specify the predicate p and the similarity value s as well. For discovered rules, these may either be specified by an expert following the discovery, or they may be computed using one of the techniques we will discuss shortly.

The handcrafted linguistic rules may deal with issues such as capitalization, abbreviations, and concept relationships. In some systems, the handcrafted rules are specified for each schema individually (*intraschema rules*) by the designer, and *interschema rules* are then "discovered" by the matching algorithm. However, in most cases, the rule base contains both intra and interschema rules.

Example 7.4 In the relational database of Example 7.2, the set of rules may have been defined (quite intuitively) as follows where RelDB refers to the relational schema and ERDB refers to the translated E-R schema:

$\langle$uppercase names $\approx$ lower case names, $true$, 1.0$)\rangle$
$\langle$uppercase names $\approx$ capitalized names, $true$, 1.0$)\rangle$
$\langle$capitalized names $\approx$ lower case names, $true$, 1.0$)\rangle$
$\langle$RelDB.ASG $\approx$ ERDB.WORKS_IN, $true$, 0.8$\rangle$
...

The first three rules are generic ones specifying how to deal with capitalizations, while the fourth one specifies a similarity between the ASG of RelDB and the WORKS_IN of ERDB. Since these correspondences always hold, $p = true$. ♦

As indicated above, there are ways of determining the element name similarities automatically. For example, COMA uses the following techniques to determine similarity of two element names:

- The *affixes* which are the common prefixes and suffixes between the two element name strings are determined.
- The *n-grams* of the two element name strings are compared. An n-gram is a substring of length n and the similarity is higher if the two strings have more n-grams in common.
- The *edit distance* between two element name strings is computed. The edit distance (also called the Levenshtein metric) determines the number of character modifications (additions, deletions, insertions) that one has to perform on one string to convert it to the second string.
- The *soundex code* of the element names is computed. This gives the phonetic similarity between names based on their soundex codes. Soundex code of English words is obtained by hashing the word to a letter and three numbers. This hash value (roughly) corresponds to how the word would sound. The important aspect of this code in our context is that two words that sound similar will have close soundex codes.

Example 7.5 Consider matching the RESP and the RESPONSIBILITY attributes in the two example schemas we are considering. The rules defined in Example 7.4 take care of the capitalization differences, so we are left with matching RESP with RESPONSIBILITY. Let us consider how the similarity between the two strings can be computed using the edit distance and the n-gram approaches.

The number of editing changes that one needs to do to convert one of these strings to the other is 10 (either we add the characters "O," "N," "S," "I," "B," "I," "L," "I," "T," "Y," to string "RESP" or delete the same characters from string "RESPONSIBILITY"). Thus the ratio of the required changes is $10/14$, which defines the edit distance between these two strings; $1 - (10/14) = 4/14 = 0.29$ is then their similarity.

For n-gram computation, we need to first fix the value of n. For this example, let $n = 3$, so we are looking for 3-grams. The 3-grams of string "RESP" are "RES" and "ESP." Similarly, there are twelve 3-grams of "RESPONSIBILITY": "RES," "ESP," "SPO," "PON," "ONS," "NSI," "SIB," "IBI," "BIP," "ILI," "LIT," and "ITY." There are two matching 3-grams out of twelve, giving a 3-gram similarity of $2/12 = 0.17$. ♦

The examples we have covered in this section all fall into the category of 1:1 matches—we matched one element of a particular schema to an element of another schema. As discussed earlier, it is possible to have 1:N (e.g., Street address, City, and Country element values in one database can be extracted from a single Address element in another), N:1 (e.g., Total_price can be calculated from Subtotal and Taxes

elements), or N:M (e.g., Book_title, Rating information can be extracted via a join of two tables one of which holds book information and the other maintains reader reviews and ratings). 1:1, 1:N, and N:1 matchers are typically used in element-level matching, while schema-level matching can also use N:M matching, since, in the latter case the necessary schema information is available.

7.1.2.3 Constraint-Based Matching Approaches

Schema definitions almost always contain semantic information that constrain the values in the database. These are typically data type information, allowable ranges for data values, key constraints, etc. In the case of instance-based techniques, the existing ranges of the values can be extracted as well as some patterns that exist in the instance data. These can be used by matchers.

Consider data types that capture a large amount of semantic information. This information can be used to disambiguate concepts and also focus the match. For example, `RESP` and RESPONSIBILITY have relatively low similarity values according to calculations in Example 7.5. However, if they have the same data type definition, this may be used to increase their similarity value. Similarly, the data type comparison may differentiate between elements that have high lexical similarity. For example, `ENO` in Fig. 7.4 has the same edit distance and n-gram similarity values to the two NUMBER attributes in Fig. 7.5 (of course, we are referring to the *names* of these attributes). In this case, the data types may be of assistance—if the data type of both `ENO` and worker number (WORKER.NUMBER) is integer, while the data type of project number (PROJECT.NUMBER) is a string, the likelihood of `ENO` matching WORKER.NUMBER is significantly higher.

In structure-based approaches, the structural similarities in the two schemas can be exploited to determine the similarity of the schema elements. If two schema elements are structurally similar, this enhances our confidence that they indeed represent the same concept. For example, if two elements have very different names and we have not been able to establish their similarity through element matchers, but they have the same properties (e.g., same attributes) that have the same data types, then we can be more confident that these two elements may be representing the same concept.

The determination of structural similarity involves checking the similarity of the "neighborhoods" of the two concepts under consideration. Definition of the neighborhood is typically done using a graph representation of the schemas where each concept (relation, entity, attribute) is a vertex and there is a directed edge between two vertices if and only if the two concepts are related (e.g., there is an edge from a relation vertex to each of its attributes, or there is an edge from a foreign key attribute vertex to the primary key attribute vertex it is referencing). In this case, the neighborhood can be defined in terms of the vertices that can be reached within a certain path length of each concept, and the problem reduces to checking the similarity of the subgraphs in this neighborhood. Many of these algorithms consider the trie rooted at the concept that is being examined and compute the

similarity of the concepts represented by the root vertices in the two trees. The fundamental idea is that if the subgraphs (subtrees) are similar, this increases the similarity of the concepts represented by the "root" vertex in the two graphs. The similarity of the subgraphs is typically determined in a bottom-up process, starting at the leaves whose similarity is determined using element matching (e.g., name similarity to the level of synonyms or data type compatibility). The similarity of the two subtrees is recursively determined based on the similarity of the vertices in the subtree. The similarity of two subgraphs (subtrees) is then defined as the fraction of leaves in the two subtrees that are strongly linked. This is based on the assumption that leaf vertices carry more information and that the structural similarity of two nonleaf schema elements is determined by the similarity of the leaf vertices in their respective subtrees, even if their immediate children are not similar. These are heuristic rules and it is possible to define others.

Another interesting approach to considering neighborhood in directed graphs while computing similarity of vertices is *similarity flooding*. It starts from an initial graph where the vertex similarities are already determined by means of an element matcher, and propagates, iteratively, to determine the similarity of each vertex to its neighbors. Hence, whenever any two elements in two schemas are found to be similar, the similarity of their adjacent vertices increases. The iterative process stops when the vertex similarities stabilize. At each iteration, to reduce the amount of work, a subset of the vertices are selected as the "most plausible" matches, which are then considered in the subsequent iteration.

Both of these approaches are agnostic to the edge semantics. In some graph representations, there is additional semantics attached to these edges. For example, *containment edges* from a relation or entity vertex to its attributes may be distinguished from *referential edges* from a foreign key attribute vertex to the corresponding primary key attribute vertex. Some systems (e.g., DIKE) exploit these edge semantics.

7.1.2.4 Learning-Based Matching

A third alternative approach that has been proposed is to use machine learning techniques to determine schema matches. Learning-based approaches formulate the problem as one of classification where concepts from various schemas are classified into classes according to their similarity. The similarity is determined by checking the features of the data instances of the databases that correspond to these schemas. How to classify concepts according to their features is learned by studying the data instances in a training dataset.

The process is as follows (Fig. 7.8). A training set (τ) is prepared that consists of instances of example correspondences between the concepts of two databases D_i and D_j. This training set can be generated after manual identification of the schema correspondences between two databases followed by extraction of example training data instances or by the specification of a query expression that converts data from one database to another. The learner uses this training data to acquire probabilistic

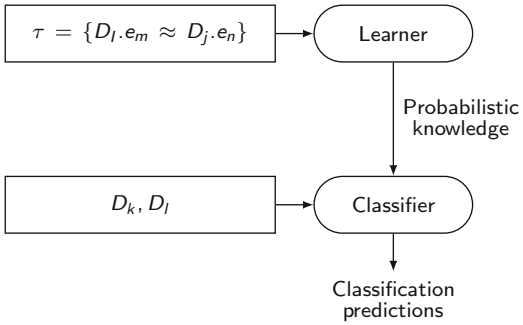

Fig. 7.8 Learning-based matching approach

information about the features of the datasets. The classifier, when given two other database instances (D_k and D_l), then uses this knowledge to go through the data instances in D_k and D_l and make predictions about classifying the elements of D_k and D_l.

This general approach applies to all of the proposed learning-based schema matching approaches. Where they differ is the type of learner that they use and how they adjust this learner's behavior for schema matching. Some have used neural networks (e.g., SEMINT), others have used Naïve Bayesian learner/classifier (Autoplex , LSD) and decision trees. We do not discuss the details of these learning techniques.

7.1.2.5 Combined Matching Approaches

The individual matching techniques that we have considered so far have their strong points and their weaknesses. Each may be more suitable for matching certain cases. Therefore, a "complete" matching algorithm or methodology usually needs to make use of more than one individual matcher.

There are two possible ways in which matchers can be combined: hybrid and composite. *Hybrid* algorithms combine multiple matchers within one algorithm. In other words, elements from two schemas can be compared using a number of element matchers (e.g., string matching as well as data type matching) and/or structural matchers within one algorithm to determine their overall similarity. Careful readers will have noted that in discussing the constraint-based matching algorithms that focused on structural matching, we followed a hybrid approach since they were based on an initial similarity determination of, for example, the leaf nodes using an element matcher, and these similarity values were then used in structural matching. *Composite* algorithms, on the other hand, apply each matcher to the elements of the two schemas (or two instances) individually, obtaining individual similarity scores, and then they apply a method for combining these similarity scores. More precisely, if $s_i(C_j^k, C_l^m)$ is the similarity score using matcher

i ($i = 1, \ldots, q$) over two concepts C_j from schema k and C_l from schema m, then the composite similarity of the two concepts is given by $s(C_j^k, C_l^m) = f(s_1, \ldots, s_q)$, where f is the function that is used to combine the similarity scores. This function can be as simple as average, max, or min, or it can be an adaptation of more complicated ranking aggregation functions that we will discuss further in Sect. 7.2. Composite approach has been proposed in the LSD and iMAP systems for handling 1:1 and N:M matches, respectively.

7.1.3 Schema Integration

Once schema matching is done, the correspondences between the various LCSs have been identified. The next step is to create the GCS, and this is referred to as *schema integration*. As indicated earlier, this step is only necessary if a GCS has not already been defined and matching was performed on individual LCSs. If the GCS was defined upfront, then the matching step would determine correspondences between it and each of the LCSs and there would be no need for the integration step. If the GCS is created as a result of the integration of LCSs based on correspondences identified during schema matching, then, as part of integration, it is important to identify the correspondences between the GCS and the LCSs. Although tools have been developed to aid in the integration process, human involvement is clearly essential.

Example 7.6 There are a number of possible integrations of the two example LCSs we have been discussing. Figure 7.9 shows one possible GCS that can be generated as a result of schema integration. We use this in the remainder of this chapter. ♦

Integration methodologies can be classified as binary or n-ary mechanisms based on the manner in which the local schemas are handled in the first phase (Fig. 7.10). Binary integration methodologies involve the manipulation of two schemas at a time. These can occur in a stepwise (ladder) fashion (Fig. 7.11a) where intermediate schemas are created for integration with subsequent schemas, or in a purely binary fashion (Fig. 7.11b), where each schema is integrated with one other, creating an intermediate schema for integration with other intermediate schemas.

EMP(E#, ENAME, TITLE, CITY)
PAY(TITLE, SAL)
PR(P#, PNAME, BUDGET, LOC)
CL(CNAME, ADDR, CT#, P#)
WORKS(E#, P#, RESP, DUR)

Fig. 7.9 Example integrated GCS (EMP is employee, PR is project, CL is client)

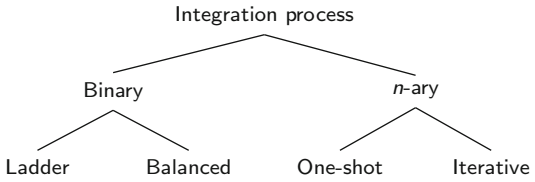

Fig. 7.10 Taxonomy of integration methodologies

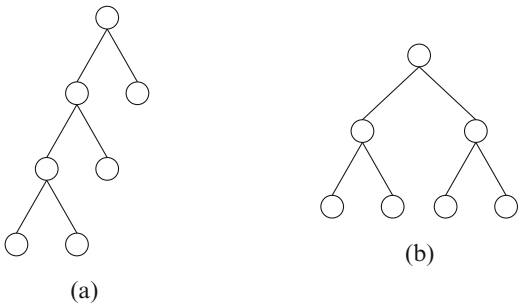

Fig. 7.11 Binary integration methods. (**a**) Stepwise. (**b**) Pure binary

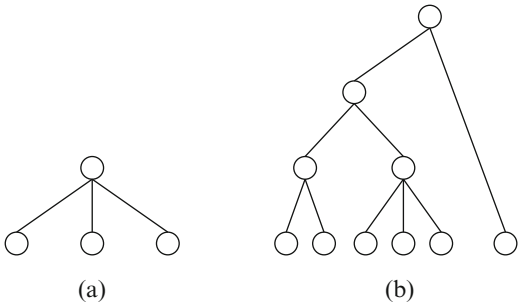

Fig. 7.12 *N*-ary integration methods. (**a**) One-pass. (**b**) Iterative

N-ary integration mechanisms integrate more than two schemas at each iteration. One-pass integration (Fig. 7.12a) occurs when all schemas are integrated at once, producing the global conceptual schema after one iteration. Benefits of this approach include the availability of complete information about all databases at integration time. There is no implied priority for the integration order of schemas, and the trade-offs, such as the best representation for data items or the most understandable structure, can be made between all schemas rather than between a few. Difficulties with this approach include increased complexity and difficulty of automation.

Iterative *n*-ary integration (Fig. 7.12b) offers more flexibility (typically, more information is available) and is more general (the number of schemas can be varied depending on the integrator's preferences). Binary approaches are a special case of

iterative n-ary. They decrease the potential integration complexity and lead towards automation techniques, since the number of schemas to be considered at each step is more manageable. Integration by an n-ary process enables the integrator to perform the operations on more than two schemas. For practical reasons, the majority of systems utilize binary methodology, but a number of researchers prefer the n-ary approach because complete information is available.

7.1.4 Schema Mapping

Once a GCS (or mediated schema) is defined, it is necessary to identify how the data from each of the local databases (source) can be mapped to GCS (target) while preserving semantic consistency (as defined by both the source and the target). Although schema matching has identified the correspondences between the LCSs and the GCS, it may not have identified explicitly how to obtain the global database from the local ones. This is what schema mapping is about.

In the case of data warehouses, schema mappings are used to explicitly extract data from the sources, and translate them to the data warehouse schema for populating it. In the case of data integration systems, these mappings are used in query processing phase by both the query processor and the wrappers (see Sect. 7.2).

There are two issues related to schema mapping that we will study: *mapping creation* and *mapping maintenance*. Mapping creation is the process of creating explicit queries that map data from a local database to the global one. Mapping maintenance is the detection and correction of mapping inconsistencies resulting from schema evolution. Source schemas may undergo structural or semantic changes that invalidate mappings. Mapping maintenance is concerned with the detection of broken mappings and the (automatic) rewriting of mappings such that semantic consistency with the new schema and semantic equivalence with the current mapping are achieved.

7.1.4.1 Mapping Creation

Mapping creation starts with a source LCS, the target GCS, and a set of schema matches M and produces a set of queries that, when executed, will create GCS data instances from the source data. In data warehouses, these queries are actually executed to create the data warehouse (global database), while in data integration systems, they are used in the reverse direction during query processing (Sect. 7.2).

Let us make this more concrete by referring to the canonical relational representation that we have adopted. The source LCS under consideration consists of a set of relations $Source = \{\mathtt{O}_1, \ldots, \mathtt{O}_m\}$, the GCS consists of a set of global (or target) relations $Target = \{\mathtt{T}_1, \ldots, \mathtt{T}_n\}$, and M consists of a set of schema match rules as defined in Sect. 7.1.2. We are looking for a way to generate, for each $\mathtt{T}_k$, a query

Q_k that is defined on a (possibly proper) subset of the relations in *Source* such that, when executed, it will generate data for $\mathtt{T}_k$ from the source relations.

This can be accomplished iteratively by considering each $\mathtt{T}_k$ in turn. It starts with $M_k \subseteq M$ (M_k is the set of rules that only apply to the attributes of $\mathtt{T}_k$) and divides it into subsets $\{M_k^1, \ldots, M_k^s\}$ such that each M_k^j specifies one possible way that values of $\mathtt{T}_k$ can be computed. Each M_k^j can be mapped to a query q_k^j that, when executed, would generate *some* of $\mathtt{T}_k$'s data. The union of all of these queries gives $Q_k (= \cup_j q_k^j)$ that we are looking for.

The algorithm proceeds in four steps that we discuss below. It does not consider the similarity values in the rules. It can be argued that the similarity values would be used in the final stages of the matching process to finalize correspondences, so that their use during mapping is unnecessary. Furthermore, by the time this phase of the integration process is reached, the concern is how to map source relation (LCS) data to target relation (GCS) data. Consequently, correspondences are not symmetric equivalences ($\approx$), but mappings ($\mapsto$): attribute(s) from (possibly multiple) source relations are mapped to an attribute of a target relation (i.e., $(\mathtt{O}_i.attribute_k, \mathtt{O}_j.attribute_l) \mapsto \mathtt{T}_w.attribute_z)$).

Example 7.7 To demonstrate the algorithm, we will use a different example database than what we have been working with, because it does not incorporate all the complexities that we wish to demonstrate. Instead, we will use the following abstract example.

Source relations (LCS):

$\mathtt{O}_1(\mathtt{A}_1, \mathtt{A}_2)$
$\mathtt{O}_2(\mathtt{B}_1, \mathtt{B}_2, \mathtt{B}_3)$
$\mathtt{O}_3(\mathtt{C}_1, \mathtt{C}_2, \mathtt{C}_3)$
$\mathtt{O}_4(\mathtt{D}_1, \mathtt{D}_2)$

Target relation (GCS):

$\mathtt{T}(\mathtt{W}_1, \mathtt{W}_2, \mathtt{W}_3, \mathtt{W}_4)$

We consider only one relation in GCS since the algorithm iterates over target relations one-at-a-time; this is sufficient to demonstrate the operation of the algorithm.

The foreign key relationships between the attributes are as follows:

Foreign key	Refers to
$\mathtt{A}_1$	$\mathtt{B}_1$
$\mathtt{A}_2$	$\mathtt{B}_1$
$\mathtt{C}_1$	$\mathtt{B}_1$

Assume that the following matches have been discovered for attributes of relation T (these make up $M_{\mathtt{T}}$). In the subsequent examples, we will not be concerned with the predicates, so they are not explicitly specified.

$$r_1 = \langle \mathtt{A}_1 \mapsto \mathtt{W}_1, p \rangle$$
$$r_2 = \langle \mathtt{A}_2 \mapsto \mathtt{W}_2, p \rangle$$
$$r_3 = \langle \mathtt{B}_2 \mapsto \mathtt{W}_4, p \rangle$$
$$r_4 = \langle \mathtt{B}_3 \mapsto \mathtt{W}_3, p \rangle$$
$$r_5 = \langle \mathtt{C}_1 \mapsto \mathtt{W}_1, p \rangle$$
$$r_6 = \langle \mathtt{C}_2 \mapsto \mathtt{W}_2, p \rangle$$
$$r_7 = \langle \mathtt{D}_1 \mapsto \mathtt{W}_4, p \rangle$$

♦

In the first step, M_k (corresponding to $\mathtt{T}_k$) is partitioned into its subsets $\{M_k^1, \ldots, M_k^n\}$ such that each M_k^j contains at most one match for each attribute of $\mathtt{T}_k$. These are called *potential candidate sets*, some of which may be *complete* in that they include a match for every attribute of $\mathtt{T}_k$, but others may not be. The reasons for considering incomplete sets are twofold. First, it may be the case that no match is found for one or more attributes of the target relation (i.e., none of the match sets is complete). Second, for large and complex database schemas, it may make sense to build the mapping iteratively so that the designer specifies the mappings incrementally.

Example 7.8 $M_{\mathtt{T}}$ is partitioned into fifty-three subsets (i.e., potential candidate sets). The first eight of these are complete, while the rest are not. We show some of these below. To make it easier to read, the complete rules are listed in the order of the target attributes to which they map (e.g., the third rule in $M_{\mathtt{T}}^1$ is r_4, because this rule maps to attribute $\mathtt{W}_3$):

$$M_{\mathtt{T}}^1 = \{r_1, r_2, r_4, r_3\} \quad M_{\mathtt{T}}^2 = \{r_1, r_2, r_4, r_7\}$$
$$M_{\mathtt{T}}^3 = \{r_1, r_6, r_4, r_3\} \quad M_{\mathtt{T}}^4 = \{r_1, r_6, r_4, r_7\}$$
$$M_{\mathtt{T}}^5 = \{r_5, r_2, r_4, r_3\} \quad M_{\mathtt{T}}^6 = \{r_5, r_2, r_4, r_7\}$$
$$M_{\mathtt{T}}^7 = \{r_5, r_6, r_4, r_3\} \quad M_{\mathtt{T}}^8 = \{r_5, r_6, r_4, r_7\}$$
$$M_{\mathtt{T}}^9 = \{r_1, r_2, r_3\} \qquad M_{\mathtt{T}}^{10} = \{r_1, r_2, r_4\}$$
$$M_{\mathtt{T}}^{11} = \{r_1, r_3, r_4\} \qquad M_{\mathtt{T}}^{12} = \{r_2, r_3, r_4\}$$
$$M_{\mathtt{T}}^{13} = \{r_1, r_3, r_6\} \qquad M_{\mathtt{T}}^{14} = \{r_3, r_4, r_6\}$$
$$\ldots \qquad \ldots$$
$$M_{\mathtt{T}}^{47} = \{r_1\} \qquad M_{\mathtt{T}}^{48} = \{r_2\}$$
$$M_{\mathtt{T}}^{49} = \{r_3\} \qquad M_{\mathtt{T}}^{50} = \{r_4\}$$
$$M_{\mathtt{T}}^{51} = \{r_5\} \qquad M_{\mathtt{T}}^{52} = \{r_6\}$$
$$M_{\mathtt{T}}^{53} = \{r_7\}$$

♦

In the second step, the algorithm analyzes each potential candidate set M_k^j to see if a "good" query can be produced for it. If all the matches in M_k^j map values from a single source relation to $\mathtt{T}_k$, then it is easy to generate a query corresponding to M_k^j. Of particular concern are matches that require access to multiple source relations. In this case the algorithm checks to see if there is a referential connection between these relations through foreign keys (i.e., whether there is a join path through the source relations). If there is not, then the potential candidate set is eliminated from further consideration. In case there are multiple join paths through foreign key relationships, the algorithm looks for those paths that will produce the most number of tuples (i.e., the estimated difference in size of the outer and inner joins is the smallest). If there are multiple such paths, then the database designer needs to be involved in selecting one (tools such as Clio, OntoBuilder, and others facilitate this process and provide mechanisms for designers to view and specify correspondences). The result of this step is a set $\overline{M_k} \subseteq M_k$ of *candidate sets*.

Example 7.9 In this example, there is no M_k^j where the values of all of $\mathtt{T}$'s attributes are mapped from a single source relation. Among those that involve multiple source relations, rules that involve $\mathtt{O}_1$, $\mathtt{O}_2$, and $\mathtt{O}_3$ can be mapped to "good" queries since there are foreign key relationships between them. However, the rules that involve $\mathtt{O}_4$ (i.e., those that include rule r_7) cannot be mapped to a "good" query since there is no join path from $\mathtt{O}_4$ to the other relations (i.e., any query would involve a cross product, which is expensive). Thus, these rules are eliminated from the potential candidate set. Considering only the complete sets, M_k^2, M_k^4, M_k^6, and M_k^8 are pruned from the set. In the end, the candidate set ($\overline{M_k}$) contains thirty-five rules (the readers are encouraged to verify this to better understand the algorithm). ♦

In the third step, the algorithm looks for a cover of the candidate sets $\overline{M_k}$. The cover $C_k \subseteq \overline{M_k}$ is a set of candidate sets such that each match in $\overline{M_k}$ appears in C_k at least once. The point of determining a cover is that it accounts for all of the matches and is, therefore, sufficient to generate the target relation $\mathtt{T}_k$. If there are multiple covers (a match can participate in multiple covers), then they are ranked in increasing number of the candidate sets in the cover. The fewer the number of candidate sets in the cover, the fewer are the number of queries that will be generated in the next step; this improves the efficiency of the mappings that are generated. If there are multiple covers with the same ranking, then they are further ranked in decreasing order of the total number of unique target attributes that are used in the candidate sets constituting the cover. The point of this ranking is that covers with higher number of attributes generate fewer null values in the result. At this stage, the designer may need to be consulted to choose from among the ranked covers.

Example 7.10 First note that we have six rules that define matches in $\overline{M_k}$ that we need to consider, since M_k^j that include rule r_7 have been eliminated. There are a large number of possible covers; let us start with those that involve M_k^1 to demonstrate the algorithm:

$$C_{\mathrm{T}}^1 = \{\underbrace{\{r_1, r_2, r_4, r_3\}}_{M_{\mathrm{T}}^1}, \underbrace{\{r_1, r_6, r_4, r_3\}}_{M_{\mathrm{T}}^3}, \underbrace{\{r_2\}}_{M_{\mathrm{T}}^{48}}\}$$

$$C_{\mathrm{T}}^2 = \{\underbrace{\{r_1, r_2, r_4, r_3\}}_{M_{\mathrm{T}}^1}, \underbrace{\{r_5, r_2, r_4, r_3\}}_{M_{\mathrm{T}}^5}, \underbrace{\{r_6\}}_{M_{\mathrm{T}}^{50}}\}$$

$$C_{\mathrm{T}}^3 = \{\underbrace{\{r_1, r_2, r_4, r_3\}}_{M_{\mathrm{T}}^1}, \underbrace{\{r_5, r_6, r_4, r_3\}}_{M_{\mathrm{T}}^7}\}$$

$$C_{\mathrm{T}}^4 = \{\underbrace{\{r_1, r_2, r_4, r_3\}}_{M_{\mathrm{T}}^1}, \underbrace{\{r_5, r_6, r_4\}}_{M_{\mathrm{T}}^{12}}\}$$

$$C_{\mathrm{T}}^5 = \{\underbrace{\{r_1, r_2, r_4, r_3\}}_{M_{\mathrm{T}}^1}, \underbrace{\{r_5, r_6, r_3\}}_{M_{\mathrm{T}}^{19}}\}$$

$$C_{\mathrm{T}}^6 = \{\underbrace{\{r_1, r_2, r_4, r_3\}}_{M_{\mathrm{T}}^1}, \underbrace{\{r_5, r_6\}}_{M_{\mathrm{T}}^{32}}\}$$

At this point we observe that the covers consist of either two or three candidate sets. Since the algorithm prefers those with fewer candidate sets, we only need to focus on those involving two sets. Furthermore, among these covers, we note that the number of target attributes in the candidate sets differ. Since the algorithm prefers covers with the largest number of target attributes in each candidate set, C_{T}^3 is the preferred cover.

Note that due to the two heuristics employed by the algorithm, the only covers we need to consider are those that involve M_{T}^1, M_{T}^3, M_{T}^5, and M_{T}^7. Similar covers can be defined involving M_{T}^3, M_{T}^5, and M_{T}^7; we leave that as an exercise. In the remainder, we will assume that the designer has chosen to use C_{T}^3 as the preferred cover. ♦

The final step of the algorithm builds a query q_k^j for each of the candidate sets in the cover selected in the previous step. The union of all of these queries (`UNION ALL`) results in the final mapping for relation $\mathtt{T}_k$ in the GCS.

Query q_k^j is built as follows:

- **`SELECT`** clause includes all correspondences (c) in each of the rules (r_k^i) in M_k^j.
- **`FROM`** clause includes all source relations mentioned in r_k^i and in the join paths determined in Step 2 of the algorithm.
- **`WHERE`** clause includes conjunct of all predicates (p) in r_k^i and all join predicates determined in Step 2 of the algorithm.
- If r_k^i contains an aggregate function either in c or in p
  - **`GROUP BY`** is used over attributes (or functions on attributes) in the **`SELECT`** clause that are not within the aggregate;

- If aggregate is in the correspondence c, it is added to **SELECT**, else (i.e., aggregate is in the predicate p) a **HAVING** clause is created with the aggregate.

Example 7.11 Since in Example 7.10 we have decided to use cover C_{T}^3 for the final mapping, we need to generate two queries: q_{T}^1 and q_{T}^7 corresponding to M_{T}^1 and M_{T}^7, respectively. For ease of presentation, we list the rules here again:

$$r_1 = \langle \mathrm{A}_1 \mapsto \mathrm{W}_1, p \rangle$$
$$r_2 = \langle \mathrm{A}_2 \mapsto \mathrm{W}_2, p \rangle$$
$$r_3 = \langle \mathrm{B}_2 \mapsto \mathrm{W}_4, p \rangle$$
$$r_4 = \langle \mathrm{B}_3 \mapsto \mathrm{W}_3, p \rangle$$
$$r_5 = \langle \mathrm{C}_1 \mapsto \mathrm{W}_1, p \rangle$$
$$r_6 = \langle \mathrm{C}_2 \mapsto \mathrm{W}_2, p \rangle$$

The two queries are as follows:

q_k^1 :
SELECT $\mathrm{A}_1, \mathrm{A}_2, \mathrm{B}_2, \mathrm{B}_3$
FROM $\mathrm{O}_1, \mathrm{O}_2$
WHERE p_1 **AND** $\mathrm{O}_1.\mathrm{A}_2 = \mathrm{O}_2.\mathrm{B}_1$

q_k^7 :
SELECT $\mathrm{B}_2, \mathrm{B}_3, \mathrm{C}_1, \mathrm{C}_2$
FROM $\mathrm{O}_2, \mathrm{O}_3$
WHERE p_3 **AND** p_4 **AND** p_5 **AND** p_6
AND $\mathrm{O}_3.c_1 = \mathrm{O}_2.\mathrm{B}_1$

Thus, the final query Q_k for target relation T becomes q_k^1 **UNION ALL** q_k^7. ♦

The output of this algorithm after it is iteratively applied to each target relation T_k is a set of queries $Q = \{Q_k\}$ that, when executed, produce data for the GCS relations. Thus, the algorithm produces GAV mappings between relational schemas—recall that GAV defines a GCS as a view over the LCSs and that is exactly what the set of mapping queries do. The algorithm takes into account the semantics of the source schema since it considers foreign key relationships in determining which queries to generate. However, it does not consider the semantics of the target, so that the tuples that are generated by the execution of the mapping queries are not guaranteed to satisfy target semantics. This is not a major issue in the case when the GCS is integrated from the LCSs; however, if the GCS is defined independent of the LCSs, then this is problematic.

It is possible to extend the algorithm to deal with target semantics as well as source semantics. This requires that interschema tuple-generating dependencies be considered. In other words, it is necessary to produce GLAV mappings. A GLAV

mapping, by definition, is not simply a query over the source relations; it is a relationship between a query over the source (i.e., LCS) relations and a query over the target (i.e., GCS) relations. Let us be more precise. Consider a schema match v that specifies a correspondence between attribute A of a source LCS relation R and attribute B of a target GCS relation T (in the notation we used in this section we have $v = \langle \mathtt{R.A} \approx \mathtt{T.B}, p, s \rangle$). Then the source query specifies how to retrieve R.A and the target query specifies how to obtain T.B. The GLAV mapping, then, is a relationship between these two queries.

This can be accomplished by starting with a source schema, a target schema, and M, and "discovering" mappings that satisfy both the source and the target schema semantics. This algorithm is also more powerful than the one we discussed in this section in that it can handle nested structures that are common in XML, object databases, and nested relational systems.

The first step in discovering all of the mappings based on schema match correspondences is *semantic translation*, which seeks to interpret schema matches in M in a way that is consistent with the semantics of both the source and target schemas as captured by the schema structure and the referential (foreign key) constraints. The result is a set of *logical mappings* each of which captures the design choices (semantics) made in both source and target schemas. Each logical mapping corresponds to one target schema relation. The second step is *data translation* that implements each logical mapping as a rule that can be translated into a query that would create an instance of the target element when executed.

Semantic translation takes as inputs the source $Source$ and target schemas $Target$, and M and performs the following two steps:

- It examines intraschema semantics within the $Source$ and $Target$ separately and produces for each a set of *logical relations* that are semantically consistent.
- It then interprets interschema correspondences M in the context of logical relations generated in Step 1 and produces a set of queries into Q that are semantically consistent with $Target$.

7.1.4.2 Mapping Maintenance

In dynamic environments where schemas evolve over time, schema mappings can be made invalid as the result of structural or constraint changes of the schemas. Thus, the detection of invalid/inconsistent schema mappings and the adaptation of such schema mappings to new schema structures/constraints are important.

In general, automatic detection of invalid/inconsistent schema mappings is desirable as the complexity of the schemas and the number of schema mappings used in database applications increase. Likewise, (semi-)automatic adaptation of mappings to schema changes is also a goal. It should be noted that automatic adaptation of schema mappings is not the same as automatic schema matching. Schema adaptation aims to resolve semantic correspondences using known changes in intraschema semantics, semantics in existing mappings, and detected semantic

inconsistencies (resulting from schema changes). Schema matching must take a much more "from scratch" approach at generating schema mappings and does not have the ability (or luxury) of incorporating such contextual knowledge.

Detecting Invalid Mappings

In general, detection of invalid mappings resulting from schema change can either happen proactively or reactively. In proactive detection environments, schema mappings are tested for inconsistencies as soon as schema changes are made by a user. The assumption (or requirement) is that the mapping maintenance system is completely aware of any and all schema changes, as soon as they are made. The ToMAS system, for example, expects users to make schema changes through its own schema editors, making the system immediately aware of any schema changes. Once schema changes have been detected, invalid mappings can be detected by doing a semantic translation of the existing mappings using the logical relations of the updated schema.

In reactive detection environments, the mapping maintenance system is unaware of when and what schema changes are made. To detect invalid schema mappings in this setting, mappings are tested at regular intervals by performing queries against the data sources and translating the resulting data using the existing mappings. Invalid mappings are then determined based on the results of these mapping tests.

An alternative method that has been proposed is to use machine learning techniques to detect invalid mappings (as in the Maveric system). What has been proposed is to build an ensemble of trained *sensors* (similar to multiple learners in schema matching) to detect invalid mappings. Examples of such sensors include value sensors for monitoring distribution characteristics of target instance values, trend sensors for monitoring the average rate of data modification, and layout and constraint sensors that monitor translated data against expected target schema syntax and semantics. A weighted combination of the findings of the individual sensors is then calculated where the weights are also learned. If the combined result indicates changes and follow-up tests suggest that this may indeed be the case, an alert is generated.

Adapting Invalid Mappings

Once invalid schema mappings are detected, they must be adapted to schema changes and made valid once again. Various high-level mapping adaptation approaches have been proposed. These can be broadly described as *fixed rule approaches* that define a remapping rule for every type of expected schema change, *map bridging approaches* that compare original schema S and the updated schema S', and generate new mapping from S to S' in addition to existing mappings, and *semantic rewriting approaches*, which exploit semantic information encoded in existing mappings, schemas, and semantic changes made to schemas to propose map

rewritings that produce semantically consistent target data. In most cases, multiple such rewritings are possible, requiring a ranking of the candidates for presentation to users who make the final decision (based on scenario- or business-level semantics not encoded in schemas or mappings).

Arguably, a complete remapping of schemas (i.e., from scratch, using schema matching techniques) is another alternative to mapping adaption. However, in most cases, map rewriting is cheaper than map regeneration as rewriting can exploit knowledge encoded in existing mappings to avoid computation of mappings that would be rejected by the user anyway (and to avoid redundant mappings).

7.1.5 Data Cleaning

Errors in source databases can always occur, requiring cleaning in order to correctly answer user queries. Data cleaning is a problem that arises in both data warehouses and data integration systems, but in different contexts. In data warehouses where data is actually extracted from local operational databases and materialized as a global database, cleaning is performed as the global database is created. In the case of data integration systems, data cleaning is a process that needs to be performed during query processing when data is returned from the source databases.

The errors that are subject to data cleaning can generally be broken down into either schema-level or instance-level concerns. Schema-level problems can arise in each individual LCS due to violations of explicit and implicit constraints. For example, values of attributes may be outside the range of their domains (e.g., 14th month or negative salary value), attribute values may violate implicit dependencies (e.g., the age attribute value may not correspond to the value that is computed as the difference between the current date and the birth date), uniqueness of attribute values may not hold, and referential integrity constraints may be violated. Furthermore, in the environment that we are considering in this chapter, the schema-level heterogeneities (both structural and semantic) among the LCSs that we discussed earlier can all be considered problems that need to be resolved. At the schema level, it is clear that the problems need to be identified at the schema match stage and fixed during schema integration.

Instance level errors are those that exist at the data level. For example, the values of some attributes may be missing although they were required, there could be misspellings and word transpositions (e.g., “M.D. Mary Smith” versus “Mary Smith, M.D.”) or differences in abbreviations (e.g., “J. Doe” in one source database, while “J.N. Doe” in another), embedded values (e.g., an aggregate address attribute that includes street name, value, province name, and postal code), values that were erroneously placed in other fields, duplicate values, and contradicting values (the salary value appearing as one value in one database and another value in another database). For instance-level cleaning, the issue is clearly one of generating the mappings such that the data is cleaned through the execution of the mapping functions (queries).

The popular approach to data cleaning has been to define a number of operators that operate either on schemas or on individual data. The operators can be composed into a data cleaning plan. Example schema operators add or drop columns from table, restructure a table by combining columns or splitting a column into two, or define more complicated schema transformation through a generic "map" operator that takes a single relation and produces one or more relations. Example data level operators include those that apply a function to every value of one attribute, merging values of two attributes into the value of a single attribute and its converse split operator, a matching operator that computes an approximate join between tuples of two relations, clustering operator that groups tuples of a relation into clusters, and a tuple merge operator that partitions the tuples of a relation into groups and collapses the tuples in each group into a single tuple through some aggregation over them, as well as basic operators to find duplicates and eliminate them. Many of the data level operators compare individual tuples of two relations (from the same or different schemas) and decide whether or not they represent the same fact. This is similar to what is done in schema matching, except that it is done at the individual data level and what is considered are not individual attribute values, but entire tuples. However, the same techniques we studied under schema matching (e.g., use of edit distance or soundex value) can be used in this context. There have been proposals for special techniques for handling this efficiently within the context of data cleaning such as fuzzy matching that computes a similarity function to determine whether the two tuples are identical or reasonably similar.

Given the large amount of data that needs to be handled, data level cleaning is expensive and efficiency is a significant issue. The physical implementation of each of the operators we discussed above is a considerable concern. Although cleaning can be done off-line as a batch process in the case of data warehouses, for data integration systems, cleaning may need to be done online as data is retrieved from the sources. The performance of data cleaning is, of course, more critical in the latter case.

7.2 Multidatabase Query Processing

We now turn our attention to querying and accessing an integrated database obtained through the techniques discussed in the previous section—this is known as the multidatabase querying problem. As previously noted, many of the distributed query processing and optimization techniques that we discussed in Chap. 4 carry over to multidatabase systems, but there are important differences. Recall from that chapter that we characterized distributed query processing in four steps: query decomposition, data localization, global optimization, and local optimization. The nature of multidatabase systems requires slightly different steps and different techniques. The component DBMSs may be autonomous and have different database languages and query processing capabilities. Thus, an MDBS layer (see Fig. 1.12) is necessary to communicate with component DBMSs in an effective way, and this requires

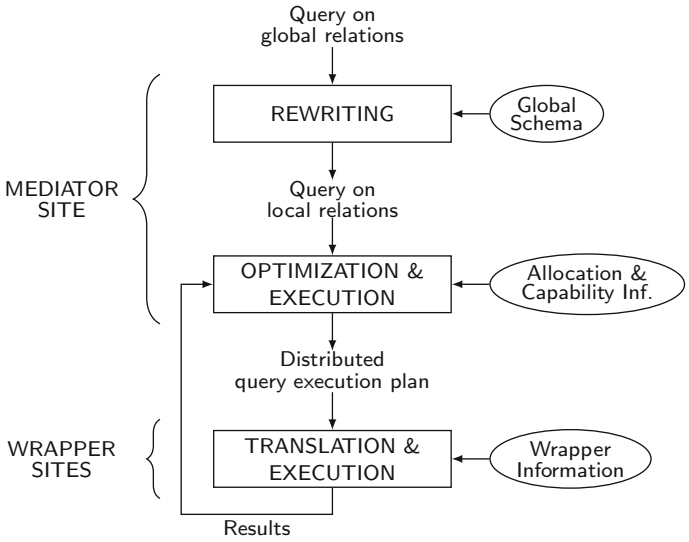

Fig. 7.13 Generic layering scheme for multidatabase query processing

additional query processing steps (Fig. 7.13). Furthermore, there may be many component DBMSs, each of which may exhibit different behavior, thereby posing new requirements for more adaptive query processing techniques.

7.2.1 Issues in Multidatabase Query Processing

Query processing in a multidatabase system is more complex than in a distributed DBMS for the following reasons:

1. The computing capabilities of the component DBMSs may be different, which prevents uniform treatment of queries across multiple DBMSs. For example, some DBMSs may be able to support complex SQL queries with join and aggregation, while some others cannot. Thus the multidatabase query processor should consider the various DBMS capabilities. The capabilities of each component is recorded in the directory along with data allocation information.
2. Similarly, the cost of processing queries may be different on different DBMSs, and the local optimization capability of each DBMS may be quite different. This increases the complexity of the cost functions that need to be evaluated.
3. The data models and languages of the component DBMSs may be quite different, for instance, relational, object-oriented, semi-structured, etc. This creates difficulties in translating multidatabase queries to component DBMS and in integrating heterogeneous results.

4. Since a multidatabase system enables access to very different DBMSs that may have different performance and behavior, distributed query processing techniques need to adapt to these variations.

The autonomy of the component DBMSs poses problems. DBMS autonomy can be defined along three main dimensions: communication, design, and execution. Communication autonomy means that a component DBMS communicates with others at its own discretion, and, in particular, it may terminate its services at any time. This requires query processing techniques that are tolerant to system unavailability. The question is how the system answers queries when a component system is either unavailable from the beginning or shuts down in the middle of query execution. Design autonomy may restrict the availability and accuracy of cost information that is needed for query optimization. The difficulty of determining local cost functions is an important issue. The execution autonomy of multidatabase systems makes it difficult to apply some of the query optimization strategies we discussed in previous chapters. For example, semijoin-based optimization of distributed joins may be difficult if the source and target relations reside in different component DBMSs, since, in this case, the semijoin execution of a join translates into three queries: one to retrieve the join attribute values of the target relation and to ship it to the source relation's DBMS, the second to perform the join at the source relation, and the third to perform the join at the target relation's DBMS. The problem arises because communication with component DBMSs occurs at a high level of the DBMS API.

In addition to these difficulties, the architecture of a distributed multidatabase system poses certain challenges. The architecture depicted in Fig. 1.12 points to an additional complexity. In distributed DBMSs, query processors have to deal only with data distribution across multiple sites. In a distributed multidatabase environment, on the other hand, data is distributed not only across sites but also across multiple databases, each managed by an autonomous DBMS. Thus, while there are two parties that cooperate in the processing of queries in a distributed DBMS (the control site and local sites), the number of parties increases to three in the case of a distributed MDBS: the MDBS layer at the control site (i.e., the mediator) receives the global query, the MDBS layers at the sites (i.e., the wrappers) participate in processing the query, and the component DBMSs ultimately optimize and execute the query.

7.2.2 Multidatabase Query Processing Architecture

Most of the work on multidatabase query processing has been done in the context of the mediator/wrapper architecture (see Fig. 1.13). In this architecture, each component database has an associated wrapper that exports information about the source schema, data, and query processing capabilities. A mediator centralizes the information provided by the wrappers in a unified view of all available data (stored

in a global data dictionary) and performs query processing using the wrappers to access the component DBMSs. The data model used by the mediator can be relational, object-oriented, or even semi-structured. In this chapter, for consistency with the previous chapters on distributed query processing, we continue to use the relational model, which is quite sufficient to explain the multidatabase query processing techniques.

The mediator/wrapper architecture has several advantages. First, the specialized components of the architecture allow the various concerns of different kinds of users to be handled separately. Second, mediators typically specialize in a related set of component databases with "similar" data, and thus export schemas and semantics related to a particular domain. The specialization of the components leads to a flexible and extensible distributed system. In particular, it allows seamless integration of different data stored in very different components, ranging from full-fledged relational DBMSs to simple files.

Assuming the mediator/wrapper architecture, we can now discuss the various layers involved in query processing in distributed multidatabase systems as shown in Fig. 7.13. As before, we assume the input is a query on global relations expressed in relational calculus. This query is posed on global (distributed) relations, meaning that data distribution and heterogeneity are hidden. Three main layers are involved in multidatabase query processing. This layering is similar to that of query processing in homogeneous distributed DBMSs (see Fig. 4.2). However, since there is no fragmentation, there is no need for the data localization layer.

The first two layers map the input query into an optimized distributed query execution plan (QEP). They perform the functions of query rewriting, query optimization, and some query execution. The first two layers are performed by the mediator and use metainformation stored in the global directory (global schema, allocation, and capability information). Query rewriting transforms the input query into a query on local relations, using the global schema. Recall that there are two main approaches for database integration: global-as-view (GAV) and local-as-view (LAV). Thus, the global schema provides the view definitions (i.e., mappings between the global relations and the local relations stored in the component databases) and the query is rewritten using the views.

Rewriting can be done at the relational calculus or algebra levels. In this chapter, we will use a generalized form of relational calculus called Datalog that is well-suited for such rewriting. Thus, there is an additional step of calculus to algebra translation that is similar to the decomposition step in homogeneous distributed DBMSs.

The second layer performs query optimization and (some) execution by considering the allocation of the local relations and the different query processing capabilities of the component DBMSs exported by the wrappers. The allocation and capability information used by this layer may also contain heterogeneous cost information. The distributed QEP produced by this layer groups within subqueries the operations that can be performed by the component DBMSs and wrappers. Similar to distributed DBMSs, query optimization can be static or dynamic. However, the lack of homogeneity in multidatabase systems (e.g., some component

DBMSs may have unexpectedly long delays in answering) makes dynamic query optimization more critical. In the case of dynamic optimization, there may be subsequent calls to this layer after execution by the subsequent layer as illustrated by the arrow showing results coming from the translation and execution layer. Finally, this layer integrates the results coming from the different wrappers to provide a unified answer to the user's query. This requires the capability of executing some operations on data coming from the wrappers. Since the wrappers may provide very limited execution capabilities, e.g., in the case of very simple component DBMSs, the mediator must provide the full execution capabilities to support the mediator interface.

The third layer performs *query translation and execution* using the wrappers. Then it returns the results to the mediator that can perform result integration from different wrappers and subsequent execution. Each wrapper maintains a *wrapper schema* that includes the local export schema and mapping information to facilitate the translation of the input subquery (a subset of the QEP) expressed in a common language into the language of the component DBMS. After the subquery is translated, it is executed by the component DBMS and the local result is translated back to the common format.

The wrapper information describes how mappings from/to participating local schemas and global schema can be performed. It enables conversions between components of the database in different ways. For example, if the global schema represents temperatures in Fahrenheit degrees, but a participating database uses Celsius degrees, the wrapper information must contain a conversion formula to provide the proper presentation to the global user and the local databases. If the conversion is across types and simple formulas cannot perform the translation, complete mapping tables could be used in the wrapper information stored in the directory.

7.2.3 Query Rewriting Using Views

Query rewriting reformulates the input query expressed on global relations into a query on local relations. It uses the global schema, which describes in terms of views the correspondences between the global relations and the local relations. Thus, the query must be rewritten using views. The techniques for query rewriting differ in major ways depending on the database integration approach that is used, i.e., GAV or LAV. In particular, the techniques for LAV (and its extension GLAV) are much more involved. Most of the work on query rewriting using views has been done using Datalog, which is a logic-based database language. Datalog is more concise than relational calculus and thus more convenient for describing complex query rewriting algorithms. In this section, we first introduce Datalog terminology. Then, we describe the main techniques and algorithms for query rewriting in the GAV and LAV approaches.

7.2.3.1 Datalog Terminology

Datalog can be viewed as an in-line version of domain relational calculus. Let us first define *conjunctive queries*, i.e., select-project-join queries, which are the basis for more complex queries. A conjunctive query in Datalog is expressed as a rule of the form:

$$Q(t) : -\mathtt{R}_1(t_1), \ldots, \mathtt{R}_n(t_n)$$

The atom $Q(t)$ is the *head* of the query and denotes the result relation. The atoms $\mathtt{R}_1(t_1), \ldots, \mathtt{R}_n(t_n)$ are the *subgoals* in the body of the query and denote database relations. Q and $\mathtt{R}_1, \ldots, \mathtt{R}_n$ are predicate names and correspond to relation names. $t, t_1, \ldots, t_n$ refer to the relation tuples and contain variables or constants. The variables are similar to domain variables in domain relational calculus. Thus, the use of the same variable name in multiple predicates expresses equijoin predicates. Constants correspond to equality predicates. More complex comparison predicates (e.g., using comparators such as $\neq$, $\leq$, and $<$) must be expressed as other subgoals. We consider queries that are *safe*, i.e., those where each variable in the head also appears in the body. Disjunctive queries can also be expressed in Datalog using unions, by having several conjunctive queries with the same head predicate.

Example 7.12 Let us consider GCS relations EMP and WORKS defined in Fig. 7.9. Consider the following SQL query:

```
SELECT E#, TITLE, P#
FROM   EMP NATURAL JOIN WORKS
WHERE  TITLE = "Programmer" OR DUR = 24
```

The corresponding query in Datalog can be expressed as:

$$Q(\mathtt{E\#}, \mathtt{TITLE}, \mathtt{P\#}) : -\mathtt{EMP}(\mathtt{E\#}, \mathtt{ENAME}, \text{“Programmer”}, \mathtt{CITY}),$$
$$\mathtt{WORKS}(\mathtt{E\#}, \mathtt{P\#}, \mathtt{RESP}, \mathtt{DUR})$$
$$Q(\mathtt{E\#}, \mathtt{TITLE}, \mathtt{P\#}) : -\mathtt{EMP}(\mathtt{E\#}, \mathtt{ENAME}, \mathtt{TITLE}, \mathtt{CITY}),$$
$$\mathtt{WORKS}(\mathtt{E\#}, \mathtt{P\#}, \mathtt{RESP}, 24)$$

♦

7.2.3.2 Rewriting in GAV

In the GAV approach, the global schema is expressed in terms of the data sources and each global relation is defined as a view over the local relations. This is similar to the global schema definition in tightly integrated distributed DBMS. In particular,

the local relations (i.e., relations in a component DBMS) can correspond to fragments. However, since the local databases preexist and are autonomous, it may happen that tuples in a global relation do not exist in local relations or that a tuple in a global relation appears in different local relations. Thus, the properties of completeness and disjointness of fragmentation cannot be guaranteed. The lack of completeness may yield incomplete answers to queries. The lack of disjointness may yield duplicate results that may still be useful information and may not need to be eliminated. Similar to queries, view definitions can use Datalog notation.

Example 7.13 Let us consider the global relations EMP and WORKS in Fig. 7.9, with a slight modification: the default responsibility of an employee in a project corresponds to its title, so that attribute TITLE is present in relation WORKS but absent in relation EMP. Let us consider the local relations EMP1 and EMP2 each with attributes E#, ENAME, TITLE, and CITY, and local relation WORKS1 with attributes E#, P#, and DUR. The global relations EMP and WORKS can be simply defined with the following Datalog rules:

$$\texttt{EMP(E\#, ENAME, CITY)} : -\texttt{EMP1(E\#, ENAME, TITLE, CITY)} \quad (d_1)$$

$$\texttt{EMP(E\#, ENAME, TITLE, CITY)} : -\texttt{EMP2(E\#, ENAME, TITLE, CITY)} \quad (d_2)$$

$$\texttt{WORKS(E\#, P\#, TITLE, DUR)} : -\texttt{EMP1(E\#, ENAME, TITLE, CITY)}, \texttt{WORKS1(E\#, P\#, DUR)} \quad (d_3)$$

$$\texttt{WORKS(E\#, P\#, TITLE, DUR)} : -\texttt{EMP2(E\#, ENAME, TITLE, CITY))}, \texttt{WORKS1(E\#, P\#, DUR)} \quad (d_4)$$

♦

Rewriting a query expressed on the global schema into an equivalent query on the local relations is relatively simple and similar to data localization in tightly integrated distributed DBMS (see Sect. 4.2). The rewriting technique using views is called *unfolding*, and it replaces each global relation invoked in the query with its corresponding view. This is done by applying the view definition rules to the query and producing a union of conjunctive queries, one for each rule application. Since a global relation may be defined by several rules (see Example 7.13), unfolding can generate redundant queries that need to be eliminated.

Example 7.14 Let us consider the global schema in Example 7.13 and the following query q that asks for assignment information about the employees living in Paris:

$$Q(e, p) : -\texttt{EMP}(e, \texttt{ENAME}, \text{“Paris”}), \texttt{WORKS}(e, p, \texttt{TITLE}, \texttt{DUR}).$$

Unfolding q produces q' as follows:

$$Q'(e, p) : -\texttt{EMP1}(e, \texttt{ENAME}, \texttt{TITLE}, \text{“Paris”}), \texttt{WORKS1}(e, p, \texttt{DUR}). \quad (q_1)$$

$$Q'(e, p) : -\texttt{EMP2}(e, \texttt{ENAME}, \texttt{TITLE}, \text{“Paris”}), \texttt{WORKS1}(e, p, \texttt{DUR}). \quad (q_2)$$

Q' is the union of two conjunctive queries labeled as q_1 and q_2. q_1 is obtained by applying GAV rule d_3 or both rules d_1 and d_3. In the latter case, the query obtained is redundant with respect to that obtained with d_3 only. Similarly, q_2 is obtained by applying rule d_4 or both rules d_2 and d_4. ♦

Although the basic technique is simple, rewriting in GAV becomes difficult when local databases have limited access patterns. This is the case for databases accessed over the web where relations can be only accessed using certain binding patterns for their attributes. In this case, simply substituting the global relations with their views is not sufficient, and query rewriting requires the use of recursive Datalog queries.

7.2.3.3 Rewriting in LAV

In the LAV approach, the global schema is expressed independent of the local databases and each local relation is defined as a view over the global relations. This enables considerable flexibility for defining local relations.

Example 7.15 To facilitate comparison with GAV, we develop an example that is symmetric to Example 7.13 with EMP and WORKS defined in that example as global relations as. In the LAV approach, the local relations EMP1, EMP2, and WORKS1 can be defined with the following Datalog rules:

$$\texttt{EMP1}(\texttt{E\#}, \texttt{ENAME}, \texttt{TITLE}, \texttt{CITY}) : -\texttt{EMP}(\texttt{E\#}, \texttt{ENAME}, \texttt{CITY}), \texttt{WORKS}(\texttt{E\#}, \texttt{P\#}, \texttt{TITLE}, \texttt{DUR}) \quad (d_5)$$

$$\texttt{EMP2}(\texttt{E\#}, \texttt{ENAME}, \texttt{TITLE}, \texttt{CITY}) : -\texttt{EMP}(\texttt{E\#}, \texttt{ENAME}, \texttt{CITY}), \texttt{WORKS}(\texttt{E\#}, \texttt{P\#}, \texttt{TITLE}, \texttt{DUR}) \quad (d_6)$$

$$\texttt{WORKS1}(\texttt{E\#}, \texttt{P\#}, \texttt{DUR}) : -\texttt{WORKS}(\texttt{E\#}, \texttt{P\#}, \texttt{TITLE}, \texttt{DUR}) \quad (d_7)$$

♦

Rewriting a query expressed on the global schema into an equivalent query on the views describing the local relations is difficult for three reasons. First, unlike in the GAV approach, there is no direct correspondence between the terms used in the global schema, (e.g., EMP, ENAME) and those used in the views (e.g., EMP1, EMP2, ENAME). Finding the correspondences requires comparison with each view. Second, there may be many more views than global relations, thus making view comparison time consuming. Third, view definitions may contain complex predicates to reflect the specific contents of the local relations, e.g., view EMP3

containing only programmers. Thus, it is not always possible to find an equivalent rewriting of the query. In this case, the best that can be done is to find a *maximally contained* query, i.e., a query that produces the maximum subset of the answer. For instance, EMP3 could only return a subset of all employees, those who are programmers.

Rewriting queries using views has received much attention because of its relevance to both logical and physical data integration problems. In the context of physical integration (i.e., data warehousing), using materialized views may be much more efficient than accessing base relations. However, the problem of finding a rewriting using views is NP-complete in the number of views and the number of subgoals in the query. Thus, algorithms for rewriting a query using views essentially try to reduce the numbers of rewritings that need to be considered. Three main algorithms have been proposed for this purpose: the bucket algorithm, the inverse rule algorithm, and the MinCon algorithm. The bucket algorithm and the inverse rule algorithm have similar limitations that are addressed by the MinCon algorithm.

The bucket algorithm considers each predicate of the query independently to select only the views that are relevant to that predicate. Given a query Q, the algorithm proceeds in two steps. In the first step, it builds a bucket b for each subgoal q of Q that is not a comparison predicate and inserts in b the heads of the views that are relevant to answer q. To determine whether a view V should be in b, there must be a mapping that unifies q with one subgoal v in V.

For instance, consider query Q in Example 7.14 and the views in Example 7.15. The following mapping unifies the subgoal EMP(e, ENAME, "Paris") of Q with the subgoal EMP(E#, ENAME, CITY) in view EMP1:

$$e \rightarrow \text{E\#}, \text{“Paris”} \rightarrow \text{CITY}$$

In the second step, for each view V of the Cartesian product of the nonempty buckets (i.e., some subset of the buckets), the algorithm produces a conjunctive query and checks whether it is contained in Q. If it is, the conjunctive query is kept as it represents one way to answer part of Q from V. Thus, the rewritten query is a union of conjunctive queries.

Example 7.16 Let us consider query Q in Example 7.14 and the views in Example 7.15. In the first step, the bucket algorithm creates two buckets, one for each subgoal of Q. Let us denote by b_1 the bucket for the subgoal EMP(e, ENAME, "Paris") and by b_2 the bucket for the subgoal WORKS(e, p, TITLE, DUR). Since the algorithm inserts only the view heads in a bucket, there may be variables in a view head that are not in the unifying mapping. Such variables are simply primed. We obtain the following buckets:

$$\begin{aligned} b_1 = \{&\text{EMP1(E\#, ENAME, TITLE}', \text{CITY}), \\ &\text{EMP2(E\#, ENAME, TITLE}', \text{CITY})\} \\ b_2 = \{&\text{WORKS1(E\#, P\#, DUR}')\} \end{aligned}$$

In the second step, the algorithm combines the elements from the buckets, which produces a union of two conjunctive queries:

$$Q'(e, p) : -\texttt{EMP1}(e, \texttt{ENAME}, \texttt{TITLE}, \text{“Paris”}), \texttt{WORKS1}(e, p, \texttt{DUR}) \qquad (q_1)$$

$$Q'(e, p) : -\texttt{EMP2}(e, \texttt{ENAME}, \texttt{TITLE}, \text{“Paris”}), \texttt{WORKS1}(e, p, \texttt{DUR}) \qquad (q_2)$$

♦

The main advantage of the bucket algorithm is that, by considering the predicates in the query, it can significantly reduce the number of rewritings that need to be considered. However, considering the predicates in the query in isolation may yield the addition of a view in a bucket that is irrelevant when considering the join with other views. Furthermore, the second step of the algorithm may still generate a large number of rewritings as a result of the Cartesian product of the buckets.

Example 7.17 Let us consider query Q in Example 7.14 and the views in Example 7.15 with the addition of the following view that gives the projects for which there are employees who live in Paris.

$$\texttt{PROJ1(P\#)} : -\texttt{EMP1(E\#, ENAME, “Paris”)}, \texttt{WORKS(E\#, P\#, TITLE, DUR)} \qquad (d_8)$$

Now, the following mapping unifies the subgoal $\texttt{WORKS}(e, p, \texttt{TITLE}, \texttt{DUR})$ of Q with the subgoal $\texttt{WORKS(E\#, P\#, TITLE, DUR)}$ in view PROJ1:

$$p \rightarrow \texttt{PNAME}$$

Thus, in the first step of the bucket algorithm, PROJ1 is added to bucket b_2. However, PROJ1 cannot be useful in a rewriting of Q since the variable ENAME is not in the head of PROJ1 and thus makes it impossible to join PROJ1 on the variable e of Q. This can be discovered only in the second step when building the conjunctive queries. ♦

The MinCon algorithm addresses the limitations of the bucket algorithm (and the inverse rule algorithm) by considering the query globally and considering how each predicate in the query interacts with the views. It proceeds in two steps like the bucket algorithm. The first step starts by selecting the views that contain subgoals corresponding to subgoals of query Q. However, upon finding a mapping that unifies a subgoal q of Q with a subgoal v in view V, it considers the join predicates in Q and finds the minimum set of additional subgoals of Q that must be mapped to subgoals in V. This set of subgoals of Q is captured by a *MinCon description* (MCD) associated with V. The second step of the algorithm produces a rewritten query by combining the different MCDs. In this second step, unlike in the bucket algorithm, it is not necessary to check that the proposed rewritings are contained in the query

because the way the MCDs are created guarantees that the resulting rewritings will be contained in the original query.

Applied to Example 7.17, the algorithm would create 3 MCDs: two for the views EMP1 and EMP2 containing the subgoal EMP of Q and one for ASG1 containing the subgoal ASG. However, the algorithm cannot create an MCD for PROJ1 because it cannot apply the join predicate in Q. Thus, the algorithm would produce the rewritten query Q' of Example 7.16. Compared with the bucket algorithm, the second step of the MinCon algorithm is much more efficient since it performs fewer combinations of MCDs than buckets.

7.2.4 Query Optimization and Execution

The three main problems of query optimization in multidatabase systems are heterogeneous cost modeling, heterogeneous query optimization (to deal with different capabilities of component DBMSs), and adaptive query processing (to deal with strong variations in the environment—failures, unpredictable delays, etc.). In this section, we describe the techniques for the first two problems. In Sect. 4.6, we presented the techniques for adaptive query processing. These techniques can be used in multidatabase systems as well, provided that the wrappers are able to collect information regarding execution within the component DBMSs.

7.2.4.1 Heterogeneous Cost Modeling

Global cost function definition, and the associated problem of obtaining cost-related information from component DBMSs, is perhaps the most-studied of the three problems. A number of possible solutions have emerged, which we discuss below.

The first thing to note is that we are primarily interested in determining the cost of the lower levels of a query execution trie that correspond to the parts of the query executed at component DBMSs. If we assume that all local processing is "pushed down" in the trie, then we can modify the query plan such that the leaves of the trie correspond to subqueries that will be executed at individual component DBMSs. In this case, we are talking about the determination of the costs of these subqueries that are input to the first level (from the bottom) operators. Cost for higher levels of the query execution trie may be calculated recursively, based on the leaf node costs.

Three alternative approaches exist for determining the cost of executing queries at component DBMSs:

1. **Black-Box Approach.** This approach treats each component DBMS as a black box, running some test queries on it, and from these determines the necessary cost information.

2. **Customized Approach.** This approach uses previous knowledge about the component DBMSs, as well as their external characteristics, to subjectively determine the cost information.
3. **Dynamic Approach.** This approach monitors the runtime behavior of component DBMSs, and dynamically collects the cost information.

We discuss each approach, focusing on the proposals that have attracted the most attention.

Black-Box Approach

In the black-box approach, the cost functions are expressed logically (e.g., aggregate CPU and I/O costs, selectivity factors), rather than on the basis of physical characteristics (e.g., relation cardinalities, number of pages, number of distinct values for each column). Thus, the cost functions for component DBMSs are expressed as

$$\begin{aligned} Cost = {} & initialization\ cost + cost\ to\ find\ qualifying\ tuples \\ & + cost\ to\ process\ selected\ tuples \end{aligned}$$

The individual terms of this formula will differ for different operators. However, these differences are not difficult to specify a priori. The fundamental difficulty is the determination of the term coefficients in these formulas, which change with different component DBMSs. One way to deal with this is to construct a synthetic database (called a *calibrating database*), run queries against it in isolation, and measure the elapsed time to deduce the coefficients.

A problem with this approach is that the results obtained by using a synthetic database may not apply well to real DBMSs. An alternative is based on running probing queries on component DBMSs to determine cost information. Probing queries can, in fact, be used to gather a number of cost information factors. For example, probing queries can be issued to retrieve data from component DBMSs to construct and update the multidatabase catalog. Statistical probing queries can be issued that, for example, count the number of tuples of a relation. Finally, performance measuring probing queries can be issued to measure the elapsed time for determining cost function coefficients.

A special case of probing queries is sample queries. In this case, queries are classified according to a number of criteria, and sample queries from each class are issued and measured to derive component cost information. Query classification can be performed according to query characteristics (e.g., unary operation queries, two-way join queries), characteristics of the operand relations (e.g., cardinality, number of attributes, information on indexed attributes), and characteristics of the underlying component DBMSs (e.g., the access methods that are supported and the policies for choosing access methods).

Classification rules are defined to identify queries that execute similarly, and thus could share the same cost formula. For example, one may consider that two queries that have similar algebraic expressions (i.e., the same algebraic trie shape), but different operand relations, attributes, or constants, are executed the same way if their attributes have the same physical properties. Another example is to assume that join order of a query has no effect on execution since the underlying query optimizer applies reordering techniques to choose an efficient join ordering. Thus, two queries that join the same set of relations belong to the same class, whatever ordering is expressed by the user. Classification rules are combined to define query classes. The classification is performed either top-down by dividing a class into more specific ones or bottom-up by merging two classes into a larger one. In practice, an efficient classification is obtained by mixing both approaches. The global cost function consists of three components: initialization cost, cost of retrieving a tuple, and cost of processing a tuple. The difference is in the way the parameters of this function are determined. Instead of using a calibrating database, sample queries are executed and costs are measured. The global cost equation is treated as a regression equation, and the regression coefficients are calculated using the measured costs of sample queries. The regression coefficients are the cost function parameters. Eventually, the cost model quality is controlled through statistical tests (e.g., F-test): if the tests fail, the query classification is refined until quality is sufficient.

The above approaches require a preliminary step to instantiate the cost model (either by calibration or sampling). This may not be always appropriate, because it would slow down the system each time a new DBMS component is added. One way to address this problem is to progressively learn the cost model from queries. The assumption is that the mediator invokes the underlying component DBMSs by a function call. The cost of a call is composed of three values: the response time to access the first tuple, the whole result response time, and the result cardinality. This allows the query optimizer to minimize either the time to receive the first tuple or the time to process the whole query, depending on end-user requirements. Initially the query processor does not know any statistics about component DBMSs. Then it monitors ongoing queries: it collects processing time of every call and stores it for future estimation. To manage the large amount of collected statistics, the cost manager summarizes them, either without loss of precision or with less precision at the benefit of lower space use and faster cost estimation. Summarization consists of aggregating statistics: the average response time is computed for all the calls that match the same pattern, i.e., those with identical function name and zero or more identical argument values. The cost estimator module is implemented in a declarative language. This allows adding new cost formulas describing the behavior of a particular component DBMS. However, the burden of extending the mediator cost model remains with the mediator developer.

The major drawback of the black-box approach is that the cost model, although adjusted by calibration, is common for all component DBMSs and may not capture their individual specifics. Thus it might fail to estimate accurately the cost of a query executed at a component DBMS that exposes unforeseen behavior.

Customized Approach

The basis of this approach is that the query processors of the component DBMSs are too different to be represented by a unique cost model as used in the black-box approach. It also assumes that the ability to accurately estimate the cost of local subqueries will improve global query optimization. The approach provides a framework to integrate the component DBMSs' cost model into the mediator query optimizer. The solution is to extend the wrapper interface such that the mediator gets some specific cost information from each wrapper. The wrapper developer is free to provide a cost model, partially or entirely. Then, the challenge is to integrate this (potentially partial) cost description into the mediator query optimizer. There are two main solutions.

A first solution is to provide the logic within the wrapper to compute three cost estimates: the time to initiate the query process and receive the first result item (called $reset\_cost$), the time to get the next item (called $advance\_cost$), and the result cardinality. Thus, the total query cost is

$$Total\_access\_cost = reset\_cost + (cardinality - 1) * advance\_cost$$

This solution can be extended to estimate the cost of database procedure calls. In that case, the wrapper provides a cost formula that is a linear equation depending on the procedure parameters. This solution has been successfully implemented to model a wide range of heterogeneous components DBMSs, ranging from a relational DBMS to an image server. It shows that a little effort is sufficient to implement a rather simple cost model and this significantly improves distributed query processing over heterogeneous sources.

A second solution is to use a hierarchical generic cost model. As shown in Fig. 7.14, each node represents a cost rule that associates a query pattern with a cost function for various cost parameters.

The node hierarchy is divided into five levels depending on the genericity of the cost rules (in Fig. 7.14, the increasing width of the boxes shows the increased focus of the rules). At the top level, cost rules apply by default to any DBMS. At the underlying levels, the cost rules are increasingly focused on: specific DBMS, relation, predicate, or query. At the time of wrapper registration, the mediator receives wrapper metadata including cost information, and completes its built-in cost model by adding new nodes at the appropriate level of the hierarchy. This framework is sufficiently general to capture and integrate both general cost knowledge declared as rules given by wrapper developers and specific information derived from recorded past queries that were previously executed. Thus, through an inheritance hierarchy, the mediator cost-based optimizer can support a wide variety of data sources. The mediator benefits from specialized cost information about each component DBMS, to accurately estimate the cost of queries and choose a more efficient QEP.

Example 7.18 Consider the GCS relations `EMP` and `WORKS` (Fig. 7.9). `EMP` is stored at component DBMS db_1 and contains 1,000 tuples. `ASG` is stored at component DBMS db_2 and contains 10,000 tuples. We assume uniform distribution

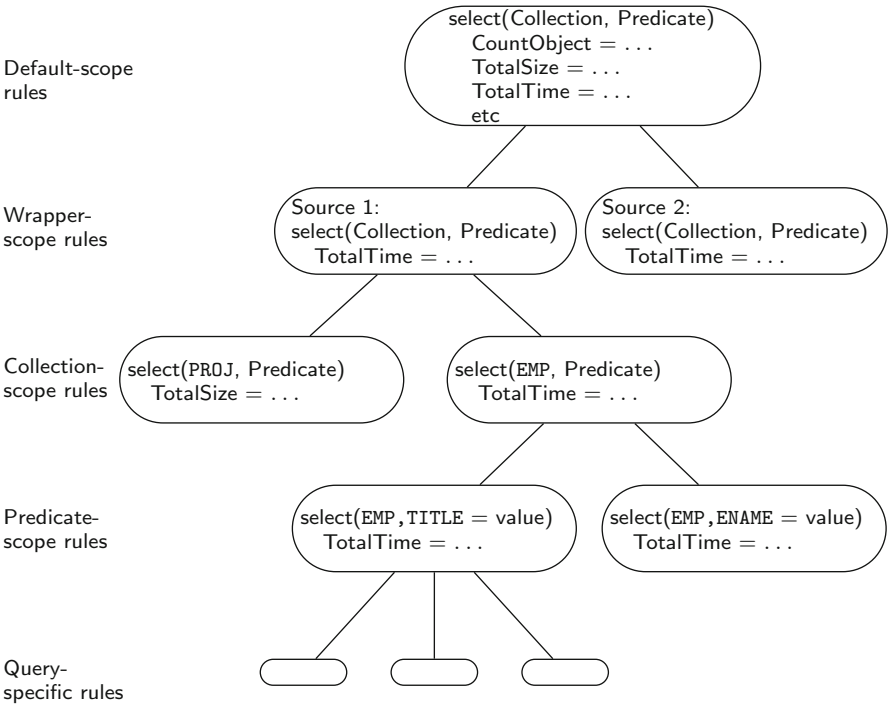

Fig. 7.14 Hierarchical cost formula trie

of attribute values. Half of the WORKS tuples have a duration greater than 6. We detail below some parts of the mediator generic cost model where R and S are two relations, A is the join attribute and we use superscripts to indicate the access method:

$cost(\mathtt{R}) = |\mathtt{R}|$

$cost(\sigma_{predicate}(\mathtt{R})) = cost(\mathtt{R})$ (access to R by sequential scan—by default)

$cost(\mathtt{R} \bowtie_{\mathtt{A}}^{ind} \mathtt{S}) = cost(\mathtt{R}) + |\mathtt{R}| * cost(\sigma_{\mathtt{A}=v}(\mathtt{S}))$ (using an index based (ind) join with the index on S.A)

$cost(\mathtt{R} \bowtie_{\mathtt{A}}^{nl} \mathtt{S}) = cost(\mathtt{R}) + |\mathtt{R}| * cost(\mathtt{S})$ (using a nested-loop (nl) join)

Consider the following global query Q:

```
SELECT *
FROM   EMP NATURAL JOIN WORKS
WHERE  WORKS.DUR>6
```

The cost-based query optimizer generates the following plans to process Q:

$$P_1 = \sigma_{\texttt{DUR}>6}(\texttt{EMP} \bowtie^{ind}_{\texttt{E\#}} \texttt{WORKS})$$

$$P_2 = \texttt{EMP} \bowtie^{nl}_{\texttt{E\#}} \sigma_{\texttt{DUR}>6}(\texttt{WORKS})$$

$$P_3 = \sigma_{\texttt{DUR}>6}(\texttt{WORKS}) \bowtie^{ind}_{\texttt{E\#}} \texttt{EMP}$$

$$P_4 = \sigma_{\texttt{DUR}>6}(\texttt{WORKS}) \bowtie^{nl}_{\texttt{E\#}} \texttt{EMP}$$

Based on the generic cost model, we compute their cost as:

$$\begin{aligned}
cost(P_1) &= cost(\sigma_{\texttt{DUR}>6}(\texttt{EMP} \bowtie^{ind}_{\texttt{E\#}} \texttt{WORKS}) \\
&= cost(\texttt{EMP} \bowtie^{ind}_{\texttt{E\#}} \texttt{WORKS}) \\
&= cost(\texttt{EMP}) + |\texttt{EMP}| * cost(\sigma_{\texttt{E\#}=v}(\texttt{WORKS})) \\
&= |\texttt{EMP}| + |\texttt{EMP}| * |\texttt{WORKS}| = 10,001,000 \\
cost(P_2) &= cost(\texttt{EMP}) + |\texttt{EMP}| * cost(\sigma_{\texttt{DUR}>6}(\texttt{WORKS})) \\
&= cost(\texttt{EMP}) + |\texttt{EMP}| * cost(\texttt{WORKS}) \\
&= |\texttt{EMP}\ | + |\texttt{EMP}| * |\texttt{WORKS}| = 10,001,000 \\
cost(P_3) &= cost(P_4) = |\texttt{WORKS}| + \frac{|\texttt{WORKS}|}{2} * |\texttt{EMP}| \\
&= 5,010,000
\end{aligned}$$

Thus, the optimizer discards plans P_1 and P_2 to keep either P_3 or P_4 for processing Q. Let us assume now that the mediator imports specific cost information about component DBMSs. db_1 exports the cost of accessing EMP tuples as:

$$cost(\sigma_{\texttt{A}=v}(\texttt{R})) = |\sigma_{\texttt{A}=v}(\texttt{R})|$$

db_2 exports the specific cost of selecting WORKS tuples that have a given E# as:

$$cost(\sigma_{\texttt{E\#}=v}(\texttt{WORKS})) = |\sigma_{\texttt{E\#}=v}(\texttt{WORKS})|$$

The mediator integrates these cost functions in its hierarchical cost model, and can now estimate more accurately the cost of the QEPs:

$$\begin{aligned}
cost(P_1) &= |\texttt{EMP}| + |\texttt{EMP}| * |\sigma_{\texttt{E\#}=v}(\texttt{WORKS})| \\
&= 1,000 + 1,000 * 10
\end{aligned}$$

$$
\begin{aligned}
&= 11,000 \\
cost(P_2) &= |\texttt{EMP}| + |\texttt{EMP}| * |\sigma_{\texttt{DUR}>6}(\texttt{WORKS})| \\
&= |\texttt{EMP}| + |\texttt{EMP}| * \frac{|\texttt{ASG}|}{2} \\
&= 5,001,000 \\
cost(P_3) &= |\texttt{WORKS}| + \frac{|\texttt{WORKS}|}{2} * |\sigma_{\texttt{E\#}=v}(\texttt{EMP})| \\
&= 10,000 + 5,000 * 1 \\
&= 15,000 \\
cost(P_4) &= |\texttt{WORKS}| + \frac{|\texttt{WORKS}|}{2} * |\texttt{EMP}| \\
&= 10,000 + 5,000 * 1,000 \\
&= 5,010,000
\end{aligned}
$$

The best QEP is now P_1 which was previously discarded because of lack of cost information about component DBMSs. In many situations P_1 is actually the best alternative to process Q_1. ♦

The two solutions just presented are well-suited to the mediator/wrapper architecture and offer a good trade-off between the overhead of providing specific cost information for diverse component DBMSs and the benefit of faster heterogeneous query processing.

Dynamic Approach

The above approaches assume that the execution environment is stable over time. However, in most cases, the execution environment factors are frequently changing. Three classes of environmental factors can be identified based on their dynamicity. The first class for frequently changing factors (every second to every minute) includes CPU load, I/O throughput, and available memory. The second class for slowly changing factors (every hour to every day) includes DBMS configuration parameters, physical data organization on disks, and database schema. The third class for almost stable factors (every month to every year) includes DBMS type, database location, and CPU speed. We focus on solutions that deal with the first two classes.

One way to deal with dynamic environments where network contention, data storage, or available memory changes over time is to extend the sampling method and consider user queries as new samples. Query response time is measured

to adjust the cost model parameters at runtime for subsequent queries. This avoids the overhead of processing sample queries periodically, but still requires heavy computation to solve the cost model equations and does not guarantee that cost model precision improves over time. A better solution, called qualitative, defines the system contention level as the combined effect of frequently changing factors on query cost. The system contention level is divided into several discrete categories: high, medium, low, or no system contention. This allows for defining a multicategory cost model that provides accurate cost estimates, while dynamic factors are varying. The cost model is initially calibrated using probing queries. The current system contention level is computed over time, based on the most significant system parameters. This approach assumes that query executions are short, so the environment factors remain rather constant during query execution. However, this solution does not apply to long running queries, since the environment factors may change rapidly during query execution.

To manage the case where the environment factor variation is predictable (e.g., the daily DBMS load variation is the same every day), the query cost is computed for successive date ranges. Then, the total cost is the sum of the costs for each range. Furthermore, it may be possible to learn the pattern of the available network bandwidth between the MDBS query processor and the component DBMS. This allows adjusting the query cost depending on the actual date.

7.2.4.2 Heterogeneous Query Optimization

In addition to heterogeneous cost modeling, multidatabase query optimization must deal with the issue of the heterogeneous computing capabilities of component DBMSs. For instance, one component DBMS may support only simple select operations, while another may support complex queries involving join and aggregate. Thus, depending on how the wrappers export such capabilities, query processing at the mediator level can be more or less complex. There are two main approaches to deal with this issue depending on the kind of interface between mediator and wrapper: query-based and operator-based.

1. **Query-based.** In this approach, the wrappers support the same query capability, e.g., a subset of SQL, which is translated to the capability of the component DBMS. This approach typically relies on a standard DBMS interface such as Open Database Connectivity (ODBC) and its extensions for the wrappers or SQL Management of External Data (SQL/MED). Thus, since the component DBMSs appear homogeneous to the mediator, query processing techniques designed for homogeneous distributed DBMS can be reused. However, if the component DBMSs have limited capabilities, the additional capabilities must be implemented in the wrappers, e.g., join queries may need to be handled at the wrapper, if the component DBMS does not support join.
2. **Operator-based.** In this approach, the wrappers export the capabilities of the component DBMSs through compositions of relational operators. Thus, there is

more flexibility in defining the level of functionality between the mediator and the wrapper. In particular, the different capabilities of the component DBMSs can be made available to the mediator. This makes wrapper construction easier at the expense of more complex query processing in the mediator. In particular, any functionality that may not be supported by component DBMSs (e.g., join) will need to be implemented at the mediator.

In the rest of this section, we present, in more detail, the approaches to query optimization in these systems.

Query-Based Approach

Since the component DBMSs appear homogeneous to the mediator, one approach is to use a distributed cost-based query optimization algorithm (see Chap. 4) with a heterogeneous cost model (see Sect. 7.2.4.1). However, extensions are needed to convert the distributed execution plan into subqueries to be executed by the component DBMSs and into subqueries to be executed by the mediator. The hybrid two-step optimization technique is useful in this case (see Sect. 4.5.3): in the first step, a static plan is produced by a centralized cost-based query optimizer; in the second step, at startup time, an execution plan is produced by carrying out site selection and allocating the subqueries to the sites. However, centralized optimizers restrict their search space by eliminating bushy join trees from consideration. Almost all the systems use left linear join orders. Consideration of only left linear join trees gives good results in centralized DBMSs for two reasons: it reduces the need to estimate statistics for at least one operand, and indexes can still be exploited for one of the operands. However, in multidatabase systems, these types of join execution plans are not necessarily the preferred ones as they do not allow any parallelism in join execution. As we discussed in earlier chapters, this is also a problem in homogeneous distributed DBMSs, but the issue is more serious in the case of multidatabase systems, because we wish to push as much processing as possible to the component DBMSs.

A way to resolve this problem is to somehow generate bushy join trees and consider them at the expense of left linear ones. One way to achieve this is to apply a cost-based query optimizer to first generate a left linear join trie, and then convert it to a bushy trie. In this case, the left linear join execution plan can be optimal with respect to total time, and the transformation improves the query response time without severely impacting the total time. A hybrid algorithm that concurrently performs a bottom-up and top-down sweep of the left linear join execution trie, transforming it, step-by-step, to a bushy one is possible. The algorithm maintains two pointers, called *upper anchor nodes* (UAN) on the trie. At the beginning, one of these, called the bottom UAN (UAN_B), is set to the grandparent of the leftmost root node (join with R_3 in Fig. 4.9), while the second one, called the top UAN (UAN_T), is set to the root (join with R_5). For each UAN the algorithm selects a *lower anchor node* (LAN). This is the node closest to the UAN and whose right

child subtree's response time is within a designer-specified range, relative to that of the UAN's right child subtree. Intuitively, the LAN is chosen such that its right child subtree's response time is **close** to the corresponding UAN's right child subtree's response time. As we will see shortly, this helps in keeping the transformed bushy trie balanced, which reduces the response time.

At each step, the algorithm picks one of the UAN/LAN pairs (strictly speaking, it picks the UAN and selects the appropriate LAN, as discussed above), and performs the following translation for the segment between that LAN and UAN pair:

1. The left child of UAN becomes the new UAN of the transformed segment.
2. The LAN remains unchanged, but its right child vertex is replaced with a new join node of two subtrees, which were the right child subtrees of the input UAN and LAN.

The UAN mode that will be considered in that particular iteration is chosen according to the following heuristic: choose UAN_B if the response time of its left child subtree is smaller than that of UAN_T's subtree; otherwise, choose UAN_T. If the response times are the same, choose the one with the more unbalanced child subtree.

At the end of each transformation step, the UAN_B and UAN_T are adjusted. The algorithm terminates when $UAN_B = UAN_T$, since this indicates that no further transformations are possible. The resulting join execution trie will be almost balanced, producing an execution plan whose response time is reduced due to parallel execution of the joins.

The algorithm described above starts with a left linear join execution trie that is generated by a centralized DBMS optimizer. These optimizers are able to generate very good plans, but the initial linear execution plan may not fully account for the peculiarities of the distributed multidatabase characteristics, such as data replication. A special global query optimization algorithm can take these into consideration. One proposed algorithm starts from an initial plan and checks for different parenthesizations of this linear join execution order to produce a parenthesized order that is optimal with respect to response time. The result is an (almost) balanced join execution trie. This approach is likely to produce better quality plans at the expense of longer optimization time.

Operator-Based Approach

Expressing the capabilities of the component DBMSs through relational operators allows tight integration of query processing between the mediator and the wrappers. In particular, the mediator/wrapper communication can be in terms of subplans. We illustrate the operator-based approach via the planning functions proposed in the Garlic project. In this approach, the capabilities of the component DBMSs are expressed by the wrappers as planning functions that can be directly called by a centralized query optimizer. It extends a rule-based optimizer with operators to create temporary relations and retrieve locally stored data. It also creates the

PushDown operator that pushes a portion of the work to the component DBMSs where it will be executed. The execution plans are represented, as usual, as operator trees, but the operator nodes are annotated with additional information that specifies the source(s) of the operand(s), whether the results are materialized, and so on. The Garlic operator trees are then translated into operators that can be directly executed by the execution engine.

Planning functions are considered by the optimizer as enumeration rules. They are called by the optimizer to construct subplans using two main functions: accessPlan to access a relation, and joinPlan to join two relations using the access plans. These functions precisely reflect the capabilities of the component DBMSs with a common formalism.

Example 7.19 We consider three component databases, each at a different site. Component database db_1 stores relation EMP(ENO, ENAME, CITY) and component database db_2 stores relation WORKS(ENO, PNAME, DUR). Component database db_3 stores only employee information with a single relation of schema EMPASG(ENAME, CITY, PNAME, DUR), whose primary key is (ENAME, PNAME). Component databases db_1 and db_2 have the same wrapper w_1, whereas db_3 has a different wrapper w_2.

Wrapper w_1 provides two planning functions typical of a relational DBMS. The accessPlan rule

accessPlan(R: relation, A: attribute list, P: select predicate) =
    scan(R, A, P, db(R))

produces a scan operator that accesses tuples of R from its component database db(R) (here we can have db(R) = db_1 or db(R) = db_2), applies select predicate P, and projects on the attribute list A. The joinPlan rule

joinPlan(R_1, R_2: relations, A: attribute list, P: join predicate) =
    join(R_1, R_2, A, P)
condition: $db(R_1) \neq db(R_2)$

produces a join operator that accesses tuples of relations R_1 and R_2 and applies join predicate P and projects on attribute list A. The condition expresses that R_1 and R_2 are stored in different component databases (i.e., db_1 and db_2). Thus, the join operator is implemented by the wrapper.

Wrapper w_2 also provides two planning functions. The accessPlan rule

accessPlan(R: relation, A: attribute list, P: select predicate) =
    fetch(CITY="c")
condition: (CITY="c") $\subseteq P$

produces a fetch operator that directly accesses (entire) employee tuples in component database db_3 whose CITY value is "c." The accessPlan rule

accessPlan(R: relation, A: attribute list, P: select predicate) =
    scan(R, A, P)

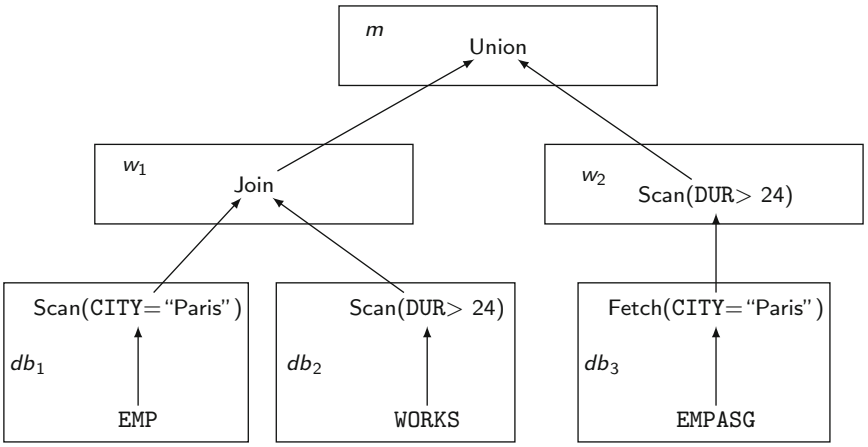

Fig. 7.15 Heterogeneous query execution plan

produces a scan operator that accesses tuples of relation R in the wrapper and applies select predicate P and attribute project list A. Thus, the scan operator is implemented by the wrapper, not the component DBMS.

Consider the following SQL query submitted to mediator m:

```
SELECT ENAME, PNAME, DUR
FROM   EMPASG
WHERE  CITY = "Paris" AND DUR > 24
```

Assuming the GAV approach, the global view EMPASG(ENAME, CITY, PNAME, DUR) can be defined as follows (for simplicity, we prefix each relation by its component database name):

$$\texttt{EMPASG} = (db_1.\texttt{EMP} \bowtie db_2.\texttt{WORKS}) \cup db_3.\texttt{EMPASG}$$

After query rewriting in GAV and query optimization, the operator-based approach could produce the QEP shown in Fig. 7.15. This plan shows that the operators that are not supported by the component DBMS are to be implemented by the wrappers or the mediator. ♦

Using planning functions for heterogeneous query optimization has several advantages in MDBSs. First, planning functions provide a flexible way to express precisely the capabilities of component data sources. In particular, they can be used to model nonrelational data sources such as web sites. Second, since these rules are declarative, so they make wrapper development easier. The only important development for wrappers is the implementation of specific operators, e.g., the scan operator of db_3 in Example 7.19. Finally, this approach can be easily incorporated in an existing, centralized query optimizer.

The operator-based approach has also been successfully used in DIMDBS, an MDBS designed to access multiple databases over the web. DISCO uses the

GAV approach and supports an object data model to represent both mediator and component database schemas and data types. This allows easy introduction of new component databases, easily handling potential type mismatches. The component DBMS capabilities are defined as a subset of an algebraic machine (with the usual operators such as scan, join, and union) that can be partially or entirely supported by the wrappers or the mediator. This gives much flexibility for the wrapper implementors in deciding where to support component DBMS capabilities (in the wrapper or in the mediator). Furthermore, compositions of operators, including specific datasets, can be specified to reflect component DBMS limitations. However, query processing is more complicated because of the use of an algebraic machine and compositions of operators. After query rewriting on the component schemas, there are three main steps:

1. **Search space generation.** The query is decomposed into a number of QEPs, which constitutes the search space for query optimization. The search space is generated using a traditional search strategy such as dynamic programming.
2. **QEP decomposition.** Each QEP is decomposed into a forest of *n wrapper QEPs* and a *composition QEP*. Each wrapper QEP is the largest part of the initial QEP that can be entirely executed by the wrapper. Operators that cannot be performed by a wrapper are moved up to the composition QEP. The composition QEP combines the results of the wrapper QEPs in the final answer, typically through unions and joins of the intermediate results produced by the wrappers.
3. **Cost evaluation.** The cost of each QEP is evaluated using a hierarchical cost model discussed in Sect. 7.2.4.1.

7.2.5 Query Translation and Execution

Query translation and execution is performed by the wrappers using the component DBMSs. A wrapper encapsulates the details of one or more component databases, each supported by the same DBMS (or file system). It also exports to the mediator the component DBMS capabilities and cost functions in a common interface. One of the major practical uses of wrappers has been to allow an SQL-based DBMS to access non-SQL databases.

The main function of a wrapper is conversion between the common interface and the DBMS-dependent interface. Figure 7.16 shows these different levels of interfaces between the mediator, the wrapper, and the component DBMSs. Note that, depending on the level of autonomy of the component DBMSs, these three components can be located differently. For instance, in the case of strong autonomy, the wrapper should be at the mediator site, possibly on the same server. Thus, communication between a wrapper and its component DBMS incurs network cost. However, in the case of a cooperative component database (e.g., within the same organization), the wrapper could be installed at the component DBMS site, much

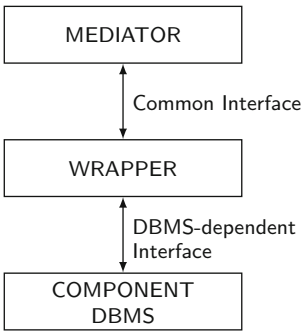

Fig. 7.16 Wrapper interfaces

like an ODBC driver. Thus, communication between the wrapper and the component DBMS is much more efficient.

The information necessary to perform conversion is stored in the wrapper schema that includes the local schema exported to the mediator in the common interface (e.g., relational) and the schema mappings to transform data between the local schema and the component database schema and vice versa. We discussed schema mappings in Sect. 7.1.4. Two kinds of conversion are needed. First, the wrapper must translate the input QEP generated by the mediator and expressed in a common interface into calls to the component DBMS using its DBMS-dependent interface. These calls yield query execution by the component DBMS that return results expressed in the DBMS-dependent interface. Second, the wrapper must translate the results to the common interface format so that they can be returned to the mediator for integration. In addition, the wrapper can execute operations that are not supported by the component DBMS (e.g., the scan operation by wrapper w_2 in Fig. 7.15).

As discussed in Sect. 7.2.4.2, the common interface to the wrappers can be query-based or operator-based. The problem of translation is similar in both approaches. To illustrate query translation in the following example, we use the query-based approach with the SQL/MED standard that allows a relational DBMS to access external data represented as foreign relations in the wrapper's local schema. This example illustrates how a very simple data source can be wrapped to be accessed through SQL.

Example 7.20 We consider relation EMP(ENO, ENAME, CITY) stored in a very simple component database, in server *ComponentDB*, built with Unix text files. Each EMP tuple can then be stored as a line in a file, e.g., with the attributes separated by ":". In SQL/MED, the definition of the local schema for this relation together with the mapping to a Unix file can be declared as a foreign relation with the following statement:

```
CREATE  FOREIGN TABLE EMP
        ENO INTEGER,
        ENAME VARCHAR(30),
```

```
        CITY VARCHAR(20)
SERVER  ComponentDB
OPTIONS (Filename '/usr/EngDB/emp.txt',
         Delimiter ':')
```

Then, the mediator can send SQL statements to the wrapper that supports access to this relation. For instance, the query

```
SELECT ENAME
FROM   EMP
```

can be translated by the wrapper using the following Unix shell command to extract the relevant attribute:

```
cut -d: -f2 /usr/EngDB/emp
```

Additional processing, e.g., for type conversion, can then be done using programming code. ♦

Wrappers are mostly used for read-only queries, which makes query translation and wrapper construction relatively easy. Wrapper construction typically relies on tools with reusable components to generate most of the wrapper code. Furthermore, DBMS vendors provide wrappers for transparently accessing their DBMS using standard interfaces. However, wrapper construction is much more difficult if updates to component databases are to be supported through wrappers (as opposed to directly updating the component databases through their DBMS). A major problem is due to the heterogeneity of integrity constraints between the common interface and the DBMS-dependent interface. As discussed in Chap. 3, integrity constraints are used to reject updates that violate database consistency. In modern DBMSs, integrity constraints are explicit and specified as rules that are part of the database schema. However, in older DBMSs or simpler data sources (e.g., files), integrity constraints are implicit and implemented by specific code in the applications. For instance, in Example 7.20, there could be applications with some embedded code that rejects insertions of new lines with an existing ENO in the EMP text file. This code corresponds to a unique key constraint on ENO in relation EMP but is not readily available to the wrapper. Thus, the main problem of updating through a wrapper is guaranteeing component database consistency by rejecting all updates that violate integrity constraints, whether they are explicit or implicit. A software engineering solution to this problem uses a tool with reverse engineering techniques to identify within application code the implicit integrity constraints that are then translated into validation code in the wrappers.

Another major problem is wrapper maintenance. Query translation relies heavily on the mappings between the component database schema and the local schema. If the component database schema is changed to reflect the evolution of the component database, then the mappings can become invalid. For instance, in Example 7.20, the administrator may switch the order of the fields in the EMP file. Using invalid mappings may prevent the wrapper from producing correct results. Since the

component databases are autonomous, detecting and correcting invalid mappings is important. The techniques to do so are those for mapping maintenance that we discussed in this chapter.

7.3 Conclusion

In this chapter, we discussed the bottom-up database design process, which we called database integration and how to execute queries over databases constructed in this manner. Database integration is the process of creating a GCS (or a mediated schema) and determining how each LCS maps to it. A fundamental separation is between data warehouses where the GCS is instantiated and materialized, and data integration systems where the GCS is merely a virtual view.

Although the topic of database integration has been studied extensively for a long time, almost all of the work has been fragmented. Individual projects focus either on schema matching, or data cleaning, or schema mapping. What is needed is an end-to-end methodology for database integration that is semiautomatic with sufficient hooks for expert involvement. One approach to such a methodology is the work of Bernstein and Melnik [2007], which provides the beginnings of a comprehensive "end-to-end" methodology.

A related concept that has received considerable discussion in literature is *data exchange*, which is defined as "the problem of taking data structured under a source schema and creating an instance of a target schema that reflects the source data as accurately as possible" [Fagin et al. 2005]. This is very similar to the physical integration (i.e., materialized) data integration, such as data warehouses, that we discussed in this chapter. A difference between data warehouses and the materialization approaches as addressed in data exchange environments is that data warehouse data typically belongs to one organization and can be structured according to a well-defined schema, while in data exchange environments data may come from different sources and contain heterogeneity.

Our focus in this chapter has been on integrating *databases*. Increasingly, however, the data that are used in distributed applications involve those that are not in a database. An interesting new topic of discussion among researchers is the integration of *structured* data that is stored in databases and *unstructured* data that is maintained in other systems (web servers, multimedia systems, digital libraries, etc.). We discuss these in Chap. 12 where we focus on the integration of data from different web repositories and introduce the recent concept of *data lakes*.

Another issue that we ignored in this chapter is data integration when a GCS does not exist or cannot be specified. The issue arises particularly in the peer-to-peer systems where the scale and the variety of data sources make it quite difficult (if not impossible) to design a GCS. We will discuss data integration in peer-to-peer systems in Chap. 9.

The second part of this chapter focused on query processing in multidatabase systems, which is significantly more complex than in tightly integrated and homogeneous distributed DBMSs. In addition to being distributed, component databases may be autonomous, have different database languages and query processing capabilities, and exhibit varying behavior. In particular, component databases may range from full-fledged SQL databases to very simple data sources (e.g., text files).

In this chapter, we addressed these issues by extending and modifying the distributed query processing architecture presented in Chap. 4. Assuming the popular mediator/wrapper architecture, we isolated the three main layers by which a query is successively rewritten (to bear on local relations) and optimized by the mediator, and then translated and executed by the wrappers and component DBMSs. We also discussed how to support OLAP queries in a multidatabase, an important requirement of decision-support applications. This requires an additional layer of translation from OLAP multidimensional queries to relational queries. This layered architecture for multidatabase query processing is general enough to capture very different variations. This has been useful to describe various query processing techniques, typically designed with different objectives and assumptions.

The main techniques for multidatabase query processing are query rewriting using multidatabase views, multidatabase query optimization and execution, and query translation and execution. The techniques for query rewriting using multidatabase views differ in major ways depending on whether the GAV or LAV integration approach is used. Query rewriting in GAV is similar to data localization in homogeneous distributed database systems. But the techniques for LAV (and its extension GLAV) are much more involved and it is often not possible to find an equivalent rewriting for a query, in which case a query that produces a maximum subset of the answer is necessary. The techniques for multidatabase query optimization include cost modeling and query optimization for component databases with different computing capabilities. These techniques extend traditional distributed query processing by focusing on heterogeneity. Besides heterogeneity, an important problem is to deal with the dynamic behavior of the component DBMSs. Adaptive query processing addresses this problem with a dynamic approach whereby the query optimizer communicates at runtime with the execution environment in order to react to unforeseen variations of runtime conditions. Finally, we discussed the techniques for translating queries for execution by the components DBMSs and for generating and managing wrappers.

The data model used by the mediator can be relational, object-oriented, or others. In this chapter, for simplicity, we assumed a mediator with a relational model that is sufficient to explain the multidatabase query processing techniques. However, when dealing with data sources on the Web, a richer mediator model such as object-oriented or semistructured (e.g., XML- or RDF-based) may be preferred. This requires significant extensions to query processing techniques.

7.4 Bibliographic Notes

A large volume of literature exists on the topic of this chapter. The work goes back to early 1980s and which is nicely surveyed by Batini et al. [1986]. Subsequent work is nicely covered by Elmagarmid et al. [1999] and Sheth and Larson [1990]. Another more recent good review of the field is by Jhingran et al. [2002].

The book by Doan et al. [2012] provides the broadest coverage of the subject. There are a number of overview papers on the topic. Bernstein and Melnik [2007] provide a very nice discussion of the integration methodology. It goes further by comparing the model management work with some of the data integration research. Halevy et al. [2006] review the data integration work in the 1990s, focusing on the Information Manifold system [Levy et al. 1996a], that uses a LAV approach. The paper provides a large bibliography and discusses the research areas that have been opened in the intervening years. Haas [2007] takes a comprehensive approach to the entire integration process and divides it into four phases: understanding that involves discovering relevant information (keys, constraints, data types, etc.), analyzing it to assess quality, and to determine statistical properties; standardization whereby the best way to represent the integrated information is determined; specification that involves the configuration of the integration process; and execution, which is the actual integration. The specification phase includes the techniques defined in this paper.

The LAV and GAV approaches are introduced and discussed by Lenzerini [2002], Koch [2001], and Calì and Calvanese [2002]. The GLAV approach is discussed in [Friedman et al. 1999] and [Halevy 2001]. A large number of systems have been developed that have tested the LAV versus GAV approaches. Many of these focus on querying over integrated systems. Examples of LAV approaches are described in the papers [Duschka and Genesereth 1997, Levy et al. 1996b, Manolescu et al. 2001], while examples of GAV are presented in papers [Adali et al. 1996a, Garcia-Molina et al. 1997, Haas et al. 1997b].

Topics of structural and semantic heterogeneity have occupied researchers for quite some time. While the literature on this topic is quite extensive, some of the interesting publications that discuss structural heterogeneity are [Dayal and Hwang 1984, Kim and Seo 1991, Breitbart et al. 1986, Krishnamurthy et al. 1991, Batini et al. 1986] (Batini et al. [1986] also discuss the structural conflicts introduced in this chapter) and those that focus on semantic heterogeneity are [Sheth and Kashyap 1992, Hull 1997, Ouksel and Sheth 1999, Kashyap and Sheth 1996, Bright et al. 1994, Ceri and Widom 1993, Vermeer 1997]. We should note that this list is seriously incomplete.

Various proposals for the canonical model for the GCS exist. The ones we discussed in this chapter and their sources are the ER model [Palopoli et al. 1998, Palopoli 2003, He and Ling 2006], object-oriented model [Castano and Antonellis 1999, Bergamaschi 2001], graph model (which is also used for determining structural similarity) [Palopoli et al. 1999, Milo and Zohar 1998, Melnik et al. 2002,

Do and Rahm 2002, Madhavan et al. 2001], trie model [Madhavan et al. 2001], and XML [Yang et al. 2003].

Doan and Halevy [2005] provide a very good overview of the various schema matching techniques, proposing a different, and simpler, classification of the techniques as rule-based, learning-based, and combined. More works in schema matching are surveyed by Rahm and Bernstein [2001], which gives a very nice comparison of various proposals. The interschema rules we discussed in this chapter are due to Palopoli et al. [1999]. The classical source for the ranking aggregation functions used in matching is [Fagin 2002].

A number of systems have been developed demonstrating the feasibility of various schema matching approaches. Among rule-based techniques, one can cite DIKE [Palopoli et al. 1998, Palopoli 2003, Palopoli et al. 2003], DIPE, which is an earlier version of this system [Palopoli et al. 1999], TranSCM [Milo and Zohar 1998], ARTEMIS [Bergamaschi 2001], similarity flooding [Melnik et al. 2002], CUPID [Madhavan et al. 2001], and COMA [Do and Rahm 2002]. For learning-based matching, Autoplex [Berlin and Motro 2001] implements a naïve Bayesian classifier, which is also the approach proposed by Doan et al. [2001, 2003a] and Naumann et al. [2002]. In the same class, decision trees are discussed in [Embley et al. 2001, 2002], and iMAP in [Dhamankar et al. 2004].

Roth and Schwartz [1997], Tomasic et al. [1997], and Thiran et al. [2006] focus on various aspects of wrapper technology. A software engineering solution to the problem of wrapper creation and maintenance, considering integrity control, is proposed in [Thiran et al. 2006].

Some sources for binary integration are [Batini et al. 1986, Pu 1988, Batini and Lenzirini 1984, Dayal and Hwang 1984, Melnik et al. 2002], while n-ary mechanisms are discussed in [Elmasri et al. 1987, Yao et al. 1982, He et al. 2004]. For some database integration tools the readers can consult [Sheth et al. 1988a], [Miller et al. 2001] that discuss Clio, and [Roitman and Gal 2006] that describes OntoBuilder.

Mapping creation algorithm in Sect. 7.1.4.1 is due to Miller et al. [2000], Yan et al. [2001], and [Popa et al. 2002]. Mapping maintenance is discussed by Velegrakis et al. [2004].

Data cleaning has gained significant interest in recent years as the integration efforts opened up to data sources more widely. The literature is rich on this topic and is well discussed in the book by Ilyas and Chu [2019]. In this context, the distinction between schema-level and instance-level cleaning is due to Rahm and Do [2000]. The data cleaning operators we discussed are column splitting [Raman and Hellerstein 2001], map operator [Galhardas et al. 2001], and fuzzy match [Chaudhuri et al. 2003].

Work on multidatabase query processing started in the early 1980s with the first multidatabase systems (e.g., [Brill et al. 1984, Dayal and Hwang 1984] and [Landers and Rosenberg 1982]). The objective then was to access different databases within an organization. In the 1990s, the increasing use of the Web for accessing all kinds of data sources triggered renewed interest and much more work in multidatabase query processing, following the popular mediator/wrapper architecture [Wiederhold

1992]. A brief overview of multidatabase query optimization issues can be found in [Meng et al. 1993]. Good discussions of multidatabase query processing can be found in [Lu et al. 1992, 1993], in Chapter 4 of [Yu and Meng 1998] and in [Kossmann 2000].

Query rewriting using views is discussed in [Levy et al. 1995] and surveyed in [Halevy 2001]. In [Levy et al. 1995], the general problem of finding a rewriting using views is shown to be NP-complete in the number of views and the number of subgoals in the query. The unfolding technique for rewriting a query expressed in Datalog in GAV was proposed in [Ullman 1997]. The main techniques for query rewriting using views in LAV are the bucket algorithm [Levy et al. 1996b], the inverse rule algorithm [Duschka and Genesereth 1997], and the MinCon algorithm [Pottinger and Levy 2000].

The three main approaches for heterogeneous cost modeling are discussed in [Zhu and Larson 1998]. The black-box approach is used in [Du et al. 1992, Zhu and Larson 1994]; the techniques in this group are probing queries: [Zhu and Larson 1996a], sample queries (which are a special case of probing) [Zhu and Larson 1998], and learning the cost over time as queries are posed and answered [Adali et al. 1996b]. The customized approach is developed in [Zhu and Larson 1996b, Roth et al. 1999, Naacke et al. 1999]; in particular cost computation can be done within the wrapper (as in Garlic) [Roth et al. 1999] or a hierarchical cost model can be developed (as in Disco) [Naacke et al. 1999]. The dynamic approach is used in [Zhu et al. 2000], [Zhu et al. 2003], and [Rahal et al. 2004] and also discussed by Lu et al. [1992]. Zhu [1995] discusses a dynamic approaching sampling and Zhu et al. [2000] present a qualitative approach.

The algorithm we described to illustrate the query-based approach to heterogeneous query optimization (Sect. 7.2.4.2) has been proposed in [Du et al. 1995] and also discussed in [Evrendilek et al. 1997]. To illustrate the operator-based approach, we described the popular solution with planning functions proposed in the Garlic project [Haas et al. 1997a]. The operator-based approach has been also used in DISCO, a multidatabase system to access component databases over the web [Tomasic et al. 1996, 1998].

The case for adaptive query processing is made by a number of researchers in a number of environments. Amsaleg et al. [1996] show why static plans cannot cope with unpredictability of data sources; the problem exists in continuous queries [Madden et al. 2002b], expensive predicates [Porto et al. 2003], and data skew [Shah et al. 2003]. The adaptive approach is surveyed in [Hellerstein et al. 2000, Gounaris et al. 2002]. The best-known dynamic approach is eddy (see Chap. 4), which is discussed in [Avnur and Hellerstein 2000]. Other important techniques for adaptive query processing are query scrambling [Amsaleg et al. 1996, Urhan et al. 1998], Ripple joins [Haas and Hellerstein 1999b], adaptive partitioning [Shah et al. 2003], and Cherry picking [Porto et al. 2003]. Major extensions to eddy are state modules [Raman et al. 2003] and distributed Eddies [Tian and DeWitt 2003].

In this chapter, we focused on the integration of structured data captured in databases. The more general problem of integrating both structured and unstructured data is discussed by Halevy et al. [2003] and Somani et al. [2002]. A different

generality direction is investigated by Bernstein and Melnik [2007], who propose a model management engine that "supports operations to match schemas, compose mappings, diff schemas, merge schemas, translate schemas into different data models, and generate data transformations from mappings."

In addition to the systems we noted above, in this chapter we referred to a number of other systems. These and their main sources are the following: SEMINT [Li and Clifton 2000, Li et al. 2000], ToMAS [Velegrakis et al. 2004], Maveric [McCann et al. 2005], and Aurora [Yan 1997, Yan et al. 1997].

Exercises

Problem 7.1 Distributed database systems and distributed multidatabase systems represent two different approaches to systems design. Find three real-life applications for which each of these approaches would be more appropriate. Discuss the features of these applications that make them more favorable for one approach or the other.

Problem 7.2 Some architectural models favor the definition of a global conceptual schema, whereas others do not. What do you think? Justify your selection with detailed technical arguments.

Problem 7.3 (*) Give an algorithm to convert a relational schema to an entity-relationship one.

Problem 7.4 (**) Consider the two databases given in Figs. 7.17 and 7.18 and described below. Design a global conceptual schema as a union of the two databases by first translating them into the E-R model.

Figure 7.17 describes a relational race database used by organizers of road races and Fig. 7.18 describes an entity-relationship database used by a shoe manufacturer. The semantics of each of these database schemas is discussed below. Figure 7.17 describes a relational road race database with the following semantics:

DIRECTOR is a relation that defines race directors who organize races; we assume that each race director has a unique name (to be used as the key), a phone number, and an address.

```
DIRECTOR(NAME, PHONE_NO, ADDRESS)
LICENSES(LIC_NO, CITY, DATE, ISSUES, COST, DEPT, CONTACT)
RACER(NAME, ADDRESS, MEM_NUM)
SPONSOR(SP_NAME, CONTACT)
RACE(R_NO, LIC_NO, DIR, MAL_WIN, FRM_WIN, SP_NAME)
```

Fig. 7.17 Road race database

LICENSES is required because all races require a governmental license, which is issued by a CONTACT in a department who is the ISSUER, possibly contained within another government department DEPT; each license has a unique LIC_NO (the key), which is issued for use in a specific CITY on a specific DATE with a certain COST.

RACER is a relation that describes people who participate in a race. Each person is identified by NAME, which is not sufficient to identify them uniquely, so a compound key formed with the ADDRESS is required. Finally, each racer may have a MEM_NUM to identify him or her as a member of the racing fraternity, but not all competitors have membership numbers.

SPONSOR indicates which sponsor is funding a given race. Typically, one sponsor funds a number of races through a specific person (CONTACT), and a number of races may have different sponsors.

RACE uniquely identifies a single race which has a license number (LIC_NO) and race number (R_NO) (to be used as a key, since a race may be planned without acquiring a license yet); each race has a winner in the male and female groups (MAL_WIN and FEM_WIN) and a race director (DIR).

Figure 7.18 illustrates an entity-relationship schema used by the sponsor's database system with the following semantics:

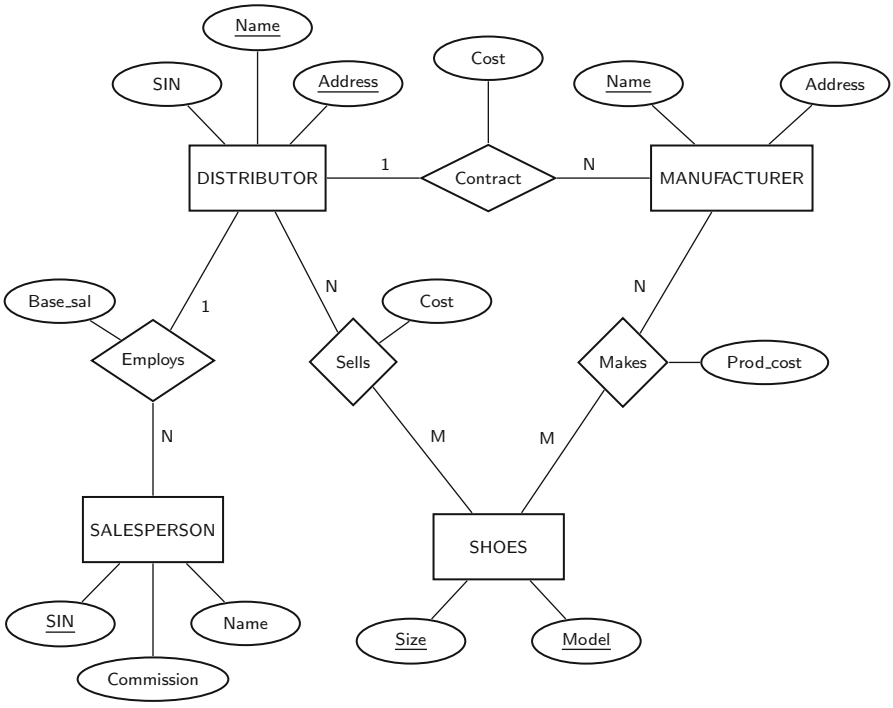

Fig. 7.18 Sponsor database

SHOES are produced by sponsors of a certain MODEL and SIZE, which forms the key to the entity.

MANUFACTURER is identified uniquely by NAME and resides at a certain ADDRESS.

DISTRIBUTOR is a person that has a NAME and ADDRESS (which are necessary to form the key) and a SIN number for tax purposes.

SALESPERSON is a person (entity) who has a NAME, earns a COMMISSION, and is uniquely identified by his or her SIN number (the key).

Makes is a relationship that has a certain fixed production cost (PROD_COST). It indicates that a number of different shoes are made by a manufacturer, and that different manufacturers produce the same shoe.

Sells is a relationship that indicates the wholesale COST to a distributor of shoes. It indicates that each distributor sells more than one type of shoe, and that each type of shoe is sold by more than one distributor.

Contract is a relationship whereby a distributor purchases, for a COST, exclusive rights to represent a manufacturer. Note that this does not preclude the distributor from selling different manufacturers' shoes.

Employs indicates that each distributor hires a number of salespeople to sell the shoes; each earns a BASE_SALARY.

Problem 7.5 (*) Consider three sources:

- Database 1 has one relation `Area(Id, Field)` providing areas of specialization of employees; the Id field identifies an employee.
- Database 2 has two relations, `Teach(Professor, Course)` and `In(Course, Field)`; `Teach` indicates the courses that each professor teaches and `In` specifies possible fields that a course can belong to.
- Database 3 has two relations, `Grant(Researcher, GrantNo)` for grants given to researchers, and `For(GrantNo, Field)` indicating which fields the grants are for.

The objective is to build a GCS with two relations: `Works(Id, Project)` stating that an employee works for a particular project, and `Area(Project, Field)` associating projects with one or more fields.

(a) Provide a LAV mapping between Database 1 and the GCS.
(b) Provide a GLAV mapping between the GCS and the local schemas.
(c) Suppose one extra relation, `Funds(GrantNo, Project)`, is added to Database 3. Provide a GAV mapping in this case.

Problem 7.6 Consider a GCS with the following relation: `Person(Name, Age, Gender)`. This relation is defined as a view over three LCSs as follows:

```
CREATE VIEW Person AS
SELECT Name, Age, "male" AS Gender
FROM   SoccerPlayer
UNION
SELECT Name, NULL AS Age, Gender
```

```
FROM   Actor
UNION
SELECT Name, Age, Gender
FROM   Politician
WHERE  Age > 30
```

For each of the following queries, discuss which of the three local schemas (`SoccerPlayer`, `Actor`, and `Politician`) contributes to the global query result.

(a) `SELECT Name FROM Person`
(b) `SELECT Name FROM Person WHERE Gender = "female"`
(c) `SELECT Name FROM Person WHERE Age > 25`
(d) `SELECT Name FROM Person WHERE Age < 25`
(e) `SELECT Name FROM Person WHERE Gender = "male" AND Age = 40`

Problem 7.7 A GCS with the relation `Country(Name, Continent, Population, HasCoast)` describes countries of the world. The attribute `HasCoast` indicates if the country has direct access to the sea. Three LCSs are connected to the global schema using the LAV approach as follows:

```
CREATE VIEW EuropeanCountry AS
SELECT Name, Continent, Population, HasCoast
FROM   Country
WHERE  Continent = "Europe"

CREATE VIEW BigCountry AS
SELECT Name, Continent, Population, HasCoast
FROM   Country
WHERE  Population >= 30000000

CREATE VIEW MidsizeOceanCountry AS
SELECT Name, Continent, Population, HasCoast
FROM   Country
WHERE  HasCoast = true AND Population > 10000000
```

(a) For each of the following queries, discuss the results with respect to their completeness, i.e., verify if the (combination of the) local sources cover all relevant results.

1. `SELECT Name FROM Country`
2. `SELECT Name FROM Country WHERE Population > 40`
3. `SELECT Name FROM Country WHERE Population > 20`

(b) For each of the following queries, discuss which of the three LCSs are necessary for the global query result.

1. `SELECT Name FROM Country`

2. `SELECT Name FROM Country WHERE Population > 30 AND Continent = "Europe"`
3. `SELECT Name FROM Country WHERE Population < 30`
4. `SELECT Name FROM Country WHERE Population > 30 AND HasCoast = true`

Problem 7.8 Consider the following two relations `PRODUCT` and `ARTICLE` that are specified in a simplified SQL notation. The perfect schema matching correspondences are denoted by arrows.

`PRODUCT`	⟶	`ARTICLE`
`Id`: int PRIMARY KEY	⟶	`Key`: varchar(255) PRIMARY KEY
`Name`: varchar(255)	⟶	`Title`: varchar(255)
`DeliveryPrice`: float	⟶	`Price`: real
`Description`: varchar(8000)	⟶	`Information`: varchar(5000)

(a) For each of the five correspondences, indicate which of the following match approaches will probably identify the correspondence:

1. Syntactic comparison of element names, e.g., using edit distance string similarity
2. Comparison of element names using a synonym lookup table
3. Comparison of data types
4. Analysis of instance data values

(b) Is it possible for the listed matching approaches to determine false correspondences for these match tasks? If so, give an example.

Problem 7.9 Consider two relations `S(a, b, c)` and `T(d, e, f)`. A match approach determines the following similarities between the elements of `S` and `T`:

	`T.d`	`T.e`	`T.f`
`S.a`	0.8	0.3	0.1
`S.b`	0.5	0.2	0.9
`S.c`	0.4	0.7	0.8

Based on the given matcher's result, derive an overall schema match result with the following characteristics:

- Each element participates in exactly one correspondence.
- There is no correspondence where both elements match an element of the opposite schema with a higher similarity than its corresponding counterpart.

Problem 7.10 (*) Figure 7.19 illustrates the schema of three different data sources:

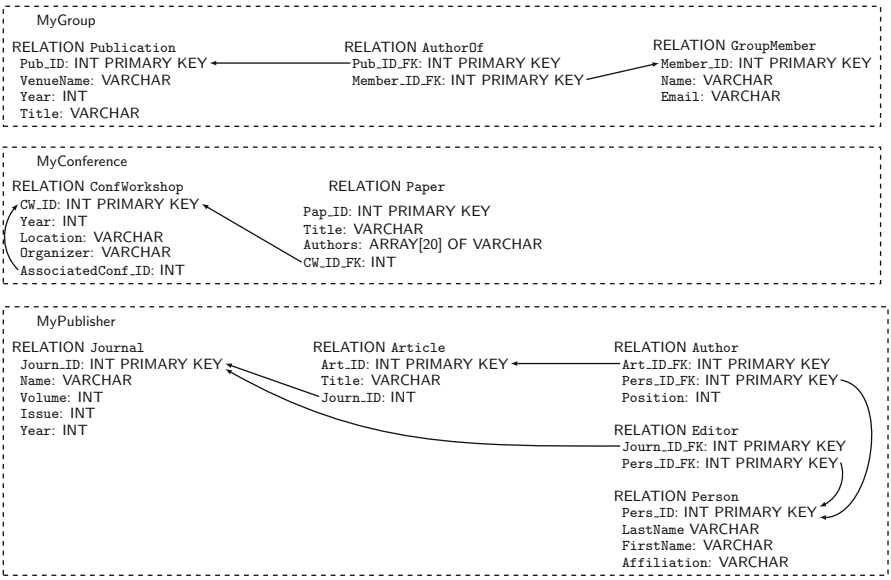

Fig. 7.19 Figure for Exercise 7.10

- MyGroup contains publications authored by members of a working group;
- MyConference contains publications of a conference series and associated workshops;
- MyPublisher contains articles that are published in journals.

The arrows show the foreign key-to-primary key relationships; note that we do not follow the proper SQL syntax of specifying foreign key relationships to save space—we resort to arrows.

The sources are defined as follows:
MyGroup

- `Publication`
  - `Pub_ID`: unique publication ID
  - `VenueName`: name of the journal, conference, or workshop
  - `VenueType`: "journal," "conference," or "workshop"
  - `Year`: year of publication
  - `Title`: publication's title
- `AuthorOf`
  - many-to-many relationship representing "group member is author of publication"
- `GroupMember`
  - `Member_ID`: unique member ID

- Name: name of the group member
- Email: email address of the group member

MyConference

- ConfWorkshop
  - CW_ID: unique ID for the conference/workshop
  - Name: name of the conference or workshop
  - Year: year when the event takes place
  - Location: event's location
  - Organizer: name of the organizing person
  - AssociatedConf_ID_FK: value is NULL if it is a conference, ID of the associated conference if the event is a workshop (this is assuming that workshops are organized in conjunction with a conference)
- Paper
  - Pap_ID: unique paper ID
  - Title: paper's title
  - Author: array of author names
  - CW_ID_FK: conference/workshop where the paper is published

MyPublisher

- Journal
  - Journ_ID: unique journal ID
  - Name: journal's name
  - Year: year when the event takes place
  - Volume: journal volume
  - Issue: journal issue
- Article
  - Art_ID: unique article ID
  - Title: title of the article
  - Journ_ID_FK: journal where the article is published
- Person
  - Pers_ID: unique person ID
  - LastName: last name of the person
  - FirstName: first name of the person
  - Affiliation: person's affiliation (e.g., the name of a university)
- Author
  - represents the many-to-many relationship for "person is author of article"
  - Position: author's position in the author list (e.g., first author has Position 1)
- Editor

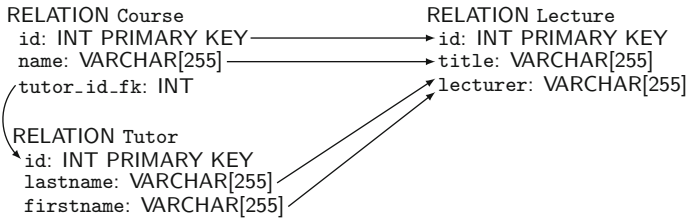

Fig. 7.20 Figure for Exercise 7.11

- represents the many-to-many relationship for "person is editor of journal issue"

(a) Identify all schema matching correspondences between the schema elements of the sources. Use the names and data types of the schema elements as well as the given description.

(b) Classify your correspondences along the following dimensions:

1. Type of schema elements (e.g., attribute–attribute or attribute–relation)
2. Cardinality (e.g., 1:1 or 1:N)

(c) Give a consolidated global schema that covers all information of the source schemas.

Problem 7.11 (*) Figure 7.20 illustrates (using a simplified SQL syntax) two sources $Source_1$ and $Source_2$. $Source_1$ has two relations, `Course` and `Tutor`, and $Source_2$ has only one relation, `Lecture`. The solid arrows denote schema matching correspondences. The dashed arrow represents a foreign key relationship between the two relations in $Source_1$.

The following are four schema mappings (represented as SQL queries) to transform $Source_1$'s data into $Source_2$.

```
1. SELECT C.id, C.name as Title, CONCAT(T.lastname,
          T.firstname) AS Lecturer
   FROM   Course AS C
   JOIN   Tutor AS T ON (C.tutor_id_fk = T.id)
2. SELECT C.id, C.name AS Title, NULL AS Lecturer
   FROM   Course AS C
   UNION
   SELECT T.id AS ID, NULL AS Title, T,
          lastname AS Lecturer
   FROM   Course AS C
   FULL OUTER JOIN Tutor AS T ON(C.tutor_id_fk=T.id)
3. SELECT C.id, C.name as Title, CONCAT(T.lastname,
          T.firstname) AS Lecturer
   FROM   Course AS C
   FULL OUTER JOIN Tutor AS T ON(C.tutor_id_fk=T.id)
```

Discuss each of these schema mappings with respect to the following questions:

(a) Is the mapping meaningful?
(b) Is the mapping complete (i.e., are all data instances of O_1 transformed)?
(c) Does the mapping potentially violate key constraints?

Problem 7.12 (*) Consider three data sources:

- Database 1 has one relation `AREA(ID, FIELD)` providing areas of specialization of employees where ID identifies an employee.
- Database 2 has two relations: `TEACH(PROFESSOR, COURSE)` and `IN(COURSE, FIELD)` specifying possible fields a course can belong to.
- Database 3 has two relations: `GRANT(RESEARCHER, GRANT#)` for grants given to researchers, and `FOR(GRANT#, FIELD)` indicating the fields that the grants are in.

Design a global schema with two relations: `WORKS(ID, PROJECT)` that records which projects employees work in, and `AREA(PROJECT, FIELD)` that associates projects with one or more fields for the following cases:

(a) There should be a LAV mapping between Database 1 and the global schema.
(b) There should be a GLAV mapping between the global schema and the local schemas.
(c) There should be a GAV mapping when one extra relation `FUNDS(GRANT#, PROJECT)` is added to Database 3.

Problem 7.13 (**) Logic (first-order logic, to be precise) has been suggested as a uniform formalism for schema translation and integration. Discuss how logic can be useful for this purpose.

Problem 7.14 (**) Can any type of global optimization be performed on global queries in a multidatabase system? Discuss and formally specify the conditions under which such optimization would be possible.

Problem 7.15 (**) Consider the global relations `EMP(ENAME, TITLE, CITY)` and `ASG(ENAME, PNAME, CITY, DUR)`. `CITY` in `ASG` is the location of the project of name `PNAME` (i.e., `PNAME` functionally determines `CITY`). Consider the local relations `EMP1(ENAME,TITLE, CITY)`, `EMP2(ENAME, TITLE, CITY)`, `PROJ1(PNAME, CITY)`, `PROJ2(PNAME, CITY)`, and `ASG1(ENAME, PNAME, DUR)`. Consider query Q which selects the names of the employees assigned to a project in Rio de Janeiro for more than 6 months and the duration of their assignment.

(a) Assuming the GAV approach, perform query rewriting.
(b) Assuming the LAV approach, perform query rewriting using the bucket algorithm.
(c) Same as (b) using the MinCon algorithm.

Problem 7.16 (*) Consider relations EMP and ASG of Example 7.18. We denote by |R| the number of pages to store R on disk. Consider the following statistics about the data:

$$|\texttt{EMP}| = 100$$
$$|\texttt{ASG}| = 2\,000$$
$$selectivity(\texttt{ASG.DUR} > 36) = 1\%$$

The mediator generic cost model is

$cost(\sigma_{\texttt{A}=v}(\texttt{R})) = |\texttt{R}|$
$cost(\sigma(X)) = cost(X)$, where X contains at least one operator.
$cost(\texttt{R} \ltimes^{ind}_{\texttt{A}} \texttt{S}) = cost(\texttt{R}) + |\texttt{R}| * cost(\sigma_{\texttt{A}=v}(\texttt{S}))$ using an indexed join algorithm.
$cost(\texttt{R} \ltimes^{nl}_{\texttt{A}} \texttt{S}) = cost(\texttt{R}) + |\texttt{R}| * cost(\texttt{S})$ using a nested loop join algorithm.

Consider the MDBS input query Q:

```
SELECT *
FROM   EMP NATURAL JOIN ASG
WHERE  ASG.DUR>36
```

Consider four plans to process Q:

$$P_1 = \texttt{EMP} \ltimes^{ind}_{\texttt{ENO}} \sigma_{\texttt{DUR}>36}(\texttt{ASG})$$
$$P_2 = \texttt{EMP} \ltimes^{nl}_{\texttt{ENO}} \sigma_{\texttt{DUR}>36}(\texttt{ASG})$$
$$P_3 = \sigma_{\texttt{DUR}>36}(\texttt{ASG}) \ltimes^{ind}_{\texttt{ENO}} \texttt{EMP}$$
$$P_4 = \sigma_{\texttt{DUR}>36}(\texttt{ASG}) \ltimes^{nl}_{\texttt{ENO}} \texttt{EMP}$$

(a) What is the cost of plans P_1 to P_4?
(b) Which plan has the minimal cost?

Problem 7.17 (*) Consider relations EMP and ASG of the previous exercise. Suppose now that the mediator cost model is completed with the following cost information issued from the component DBMSs.

The cost of accessing EMP tuples at db_1 is

$$cost(\sigma_{\texttt{A}=v}(\texttt{R})) = |\sigma_{\texttt{A}=v}(\texttt{R})|$$

The specific cost of selecting ASG tuples that have a given ENO at db_2 is

$$cost(\sigma_{\texttt{ENO}=v}(\texttt{ASG})) = |\sigma_{\texttt{ENO}=v}(\texttt{ASG})|$$

(a) What is the cost of plans P_1 to P_4?
(b) Which plan has the minimal cost?

Problem 7.18 (**) What are the respective advantages and limitations of the query-based and operator-based approaches to heterogeneous query optimization from the points of view of query expressiveness, query performance, development cost of wrappers, system (mediator and wrappers) maintenance, and evolution?

Problem 7.19 (**) Consider Example 7.19 by adding, at a new site, component database db_4 which stores relations `EMP(ENO, ENAME, CITY)` and `ASG(ENO, PNAME, DUR)`. db_4 exports through its wrapper w_3 join and scan capabilities. Let us assume that there can be employees in db_1 with corresponding assignments in db_4 and employees in db_4 with corresponding assignments in db_2.

(a) Define the planning functions of wrapper w_3.
(b) Give the new definition of global view `EMPASG(ENAME, CITY, PNAME, DUR)`.
(c) Give a QEP for the same query as in Example 7.19.

Chapter 8
Parallel Database Systems

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

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

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

Most of the research on parallel database systems has been done in the context of the relational model because it provides a good basis for parallel data processing. In this chapter, we present the parallel database system approach as a solution to high-performance and high-availability data management. We discuss the advantages and

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

disadvantages of the various parallel system architectures and we present the generic implementation techniques.

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

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

8.1 Objectives

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

Parallel database systems combine database management and parallel processing to increase performance and availability. Note that performance was also the objective of *database machines* in the 1980s. The problem faced by conventional database management has long been known as "I/O bottleneck," induced by high disk access time with respect to main memory access time (typically hundreds of thousands times faster). Initially, database machine designers tackled this problem through special-purpose hardware, e.g., by introducing data filtering devices within the disk heads. However, this approach failed because of poor cost/performance compared to the software solution, which can easily benefit from hardware progress in silicon technology. The idea of pushing database functions closer to disk has received renewed interest with the introduction of general-purpose microprocessors in disk controllers, thus leading to intelligent disks. For instance, basic functions that require costly sequential scan, e.g., select operations on tables with fuzzy predicates, can be more efficiently performed at the disk level since they avoid overloading the DBMS memory with irrelevant disk blocks. However, exploiting intelligent disks requires adapting the DBMS, in particular, the query processor to decide whether

to use the disk functions. Since there is no standard intelligent disk technology, adapting to different intelligent disk technologies hurts DBMS portability.

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

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

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

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

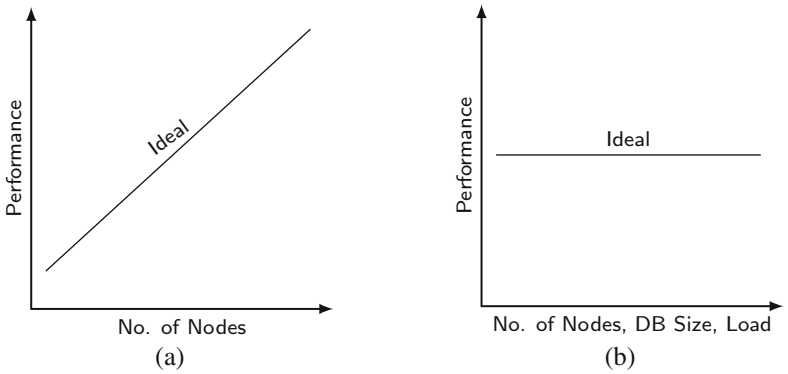

Fig. 8.1 Extensibility metrics. (**a**) Linear speed-up. (**b**) Linear scale-up

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

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

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

8.2 Parallel Architectures

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

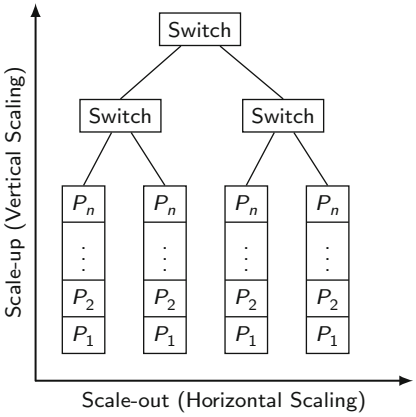

Fig. 8.2 Scale-up versus scale-out

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

8.2.1 General Architecture

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

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

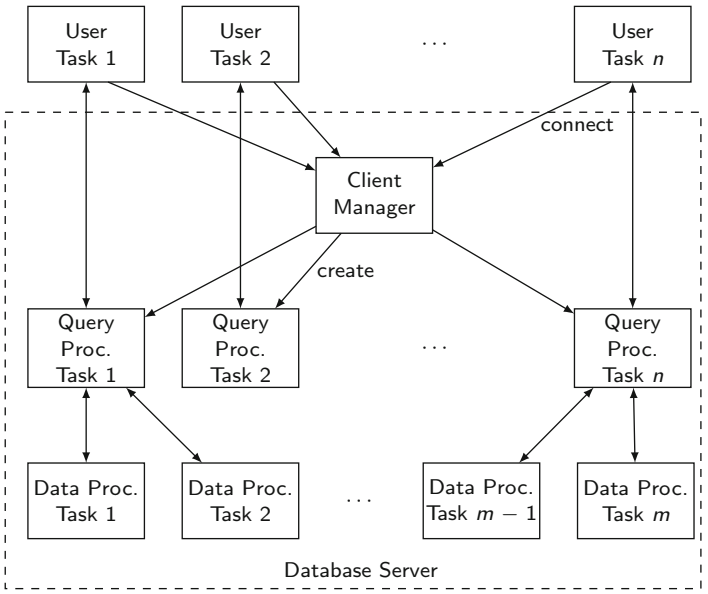

Fig. 8.3 General architecture of a parallel database system

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

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

8.2.2 Shared-Memory

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

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

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

8.2.2.1 Uniform Memory Access (UMA)

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

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

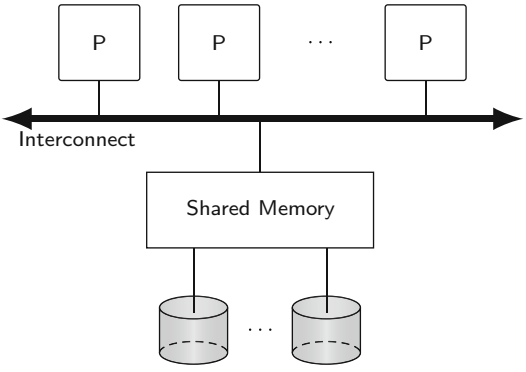

Fig. 8.4 Shared-memory

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

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

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

8.2.2.2 Non-Uniform Memory Access (NUMA)

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

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

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

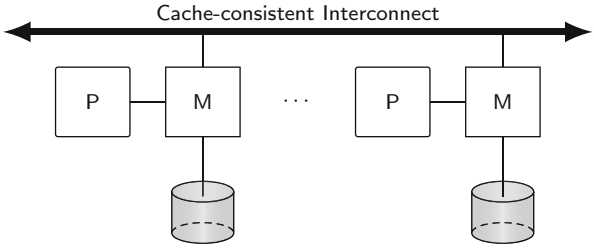

Fig. 8.5 Cache coherent non-uniform memory architecture (CC-NUMA)

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

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

8.2.3 Shared-Disk

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

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

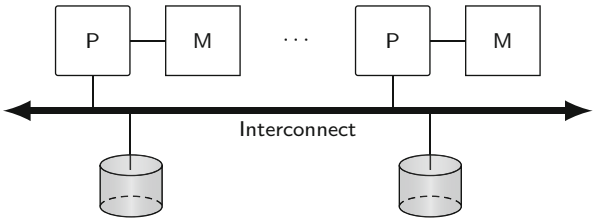

Fig. 8.6 Shared-disk architecture

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

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

8.2.4 Shared-Nothing

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

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

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

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

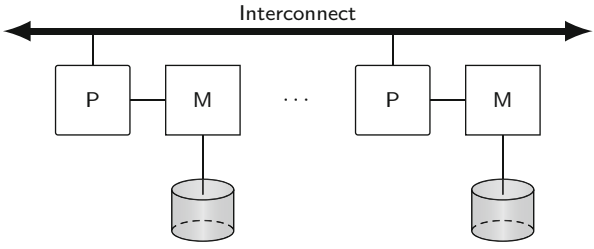

Fig. 8.7 Shared-nothing architecture

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

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

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

8.3 Data Placement

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

The most common data partitioning strategies that are used in parallel DBMSs are the round-robin, hashing, and range-partitioning approaches discussed in Sect. 2.1.1. Data partitioning must scale with the increase in database size and load. Thus, the degree of partitioning, i.e., the number of nodes over which a relation is partitioned, should be a function of the size and access frequency of the relation. Therefore, increasing the degree of partitioning may result in placement

reorganization. For example, a relation initially placed across eight nodes may have its cardinality doubled by subsequent insertions, in which case it should be placed across 16 nodes.

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

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

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

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

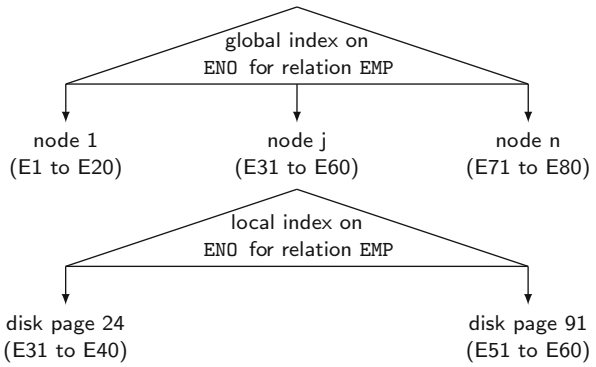

Fig. 8.8 Example of global and local indexes

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

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

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

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

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

Fig. 8.9 Example of interleaved partitioning

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

Fig. 8.10 Example of chained partitioning

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

8.4 Parallel Query Processing

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

8.4.1 Parallel Algorithms for Data Processing

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

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

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

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

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

8.4.1.1 Parallel Sort Algorithms

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

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

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

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

8.4.1.2 Parallel Join Algorithms

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

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

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

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

Parallel Nested Loop Join Algorithm

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

Algorithm 8.1: Parallel Nested Loop (PNL)

Input: $\mathtt{R}_1, \mathtt{R}_2, \ldots, \mathtt{R}_m$: fragments of relation R
$\mathtt{S}_1, \mathtt{S}_2, \ldots, \mathtt{S}_n$: fragments of relation S ;
JP: join predicate
Output: $\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_n$: result fragments
begin
  for *i from* 1 *to m in parallel* **do** {send R entirely to each S-node}
    send $\mathtt{R}_i$ to each node containing a fragment of S
  end for
  for *j from* 1 *to n in parallel* **do** {perform the join at each S-node}
    $\mathtt{R} \leftarrow \bigcup_{i=1}^{m} \mathtt{R}_i$; {$\mathtt{R}_i$ from R-nodes; R is fully replicated at S-nodes}
    $\mathtt{T}_j \leftarrow \mathtt{R} \bowtie_{JP} \mathtt{S}_j$
  end for
end

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

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

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

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

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

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

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

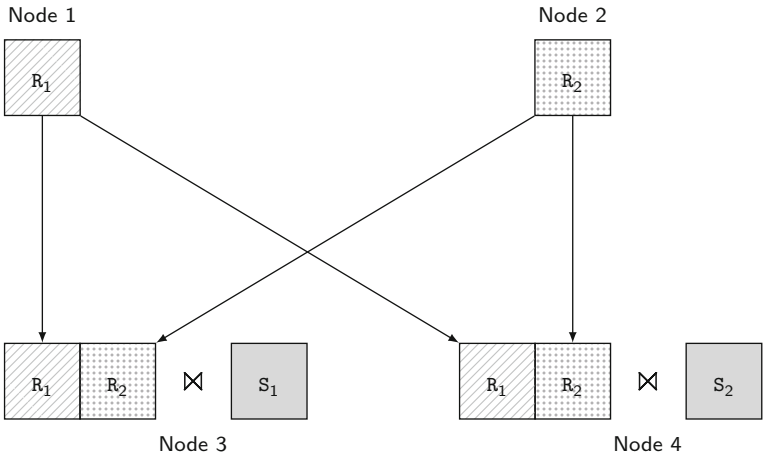

Fig. 8.11 Example of parallel nested loop

Parallel Hash Join Algorithm

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

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

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

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

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

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

Algorithm 8.2: Parallel Hash Join (PHJ)

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

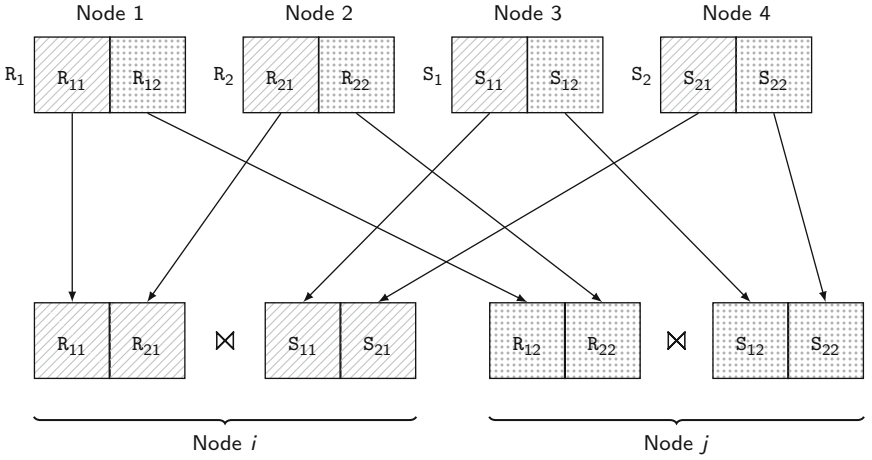

Fig. 8.12 Example of parallel hash join

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

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

Variants

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

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

The earlier example of symmetric join algorithm is the symmetric hash join, which uses two hash tables, one for each input relation. The traditional build and probe phases of the basic hash join algorithm are simply interleaved. When a tuple arrives, it is used to probe the hash table corresponding to the other relation and find matching tuples. Then, it is inserted in its corresponding hash table so that tuples of the other relation arriving later can be joined. Thus, each arriving tuple can be processed on the fly. Another popular symmetric join algorithm is the ripple join, which is a generalization of the nested loop join algorithm where the roles of inner and outer relation continually alternate during query execution. The main idea is to keep the probing state of each input relation, with a pointer that indicates the last tuple used to probe the other relation. At each toggling point, a change of roles

between inner and outer relations occurs. At this point, the new outer relation starts to probe the inner input from its pointer position onwards, to a specified number of tuples. The inner relation, in turn, is scanned from its first tuple to its pointer position minus 1. The number of tuples processed at each stage in the outer relation gives the toggling rate and can be adaptively monitored.

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

8.4.2 Parallel Query Optimization

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

8.4.2.1 Search Space

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

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

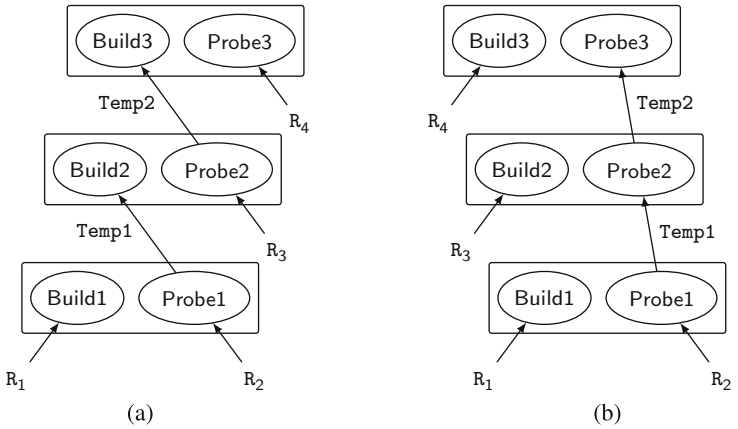

Fig. 8.13 Two hash join trees with a different scheduling. (**a**) No pipeline. (**b**) Pipeline of R2, Temp1, and Temp2

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

Example 8.4 Figure 8.13 shows two execution trees, one with no pipeline (Fig. 8.13a) and one with pipeline (Fig. 8.13b). In Fig. 8.13a, the temporary relation Temp1 must be completely produced and the hash table in Build2 must be built before Probe2 can start consuming R3. The same is true for Temp2, Build3, and Probe3. Thus, the trie is executed in four consecutive phases: (1) build R1's hash table, (2) probe it with R2 and build Temp1's hash table, (3) probe it with R3 and build Temp2's hash table, (4) probe it with R3 and produce the result. Figure 8.13b shows a pipeline execution. The trie can be executed in two phases if enough memory is available to build the hash tables: (1) build the tables for R1 R3 and R4, (2) execute Probe1, Probe2, and Probe3 in pipeline. ♦

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

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

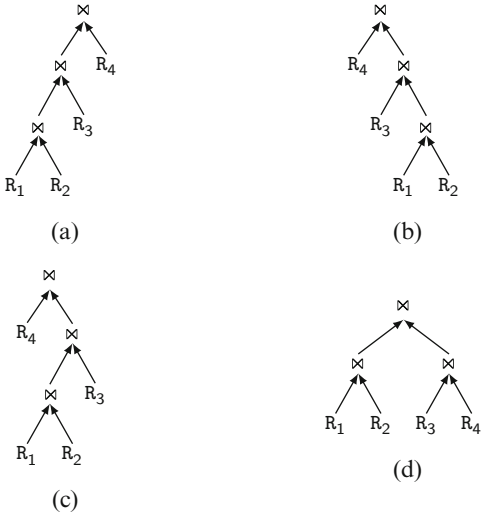

Fig. 8.14 Execution plans as operator trees. (**a**) Left deep. (**b**) Right deep. (**c**) Zigzag. (**d**) Bushy

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

Parallel trie formats other than left or right deep are also interesting. For example, bushy trees (Fig. 8.14d) are the only ones to allow independent parallelism and some pipeline parallelism. Independent parallelism is useful when the relations are partitioned on disjoint homes. Suppose that the relations in Fig. 8.14d are partitioned such that $\mathtt{R}_1$ and $\mathtt{R}_2$ have the same home h_1 and $\mathtt{R}_3$ and $\mathtt{R}_4$ have the same home h_2 that is different than h_1. Then, the two joins of the base relations could be independently executed in parallel by the set of nodes that constitutes h_1 and h_2.

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

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

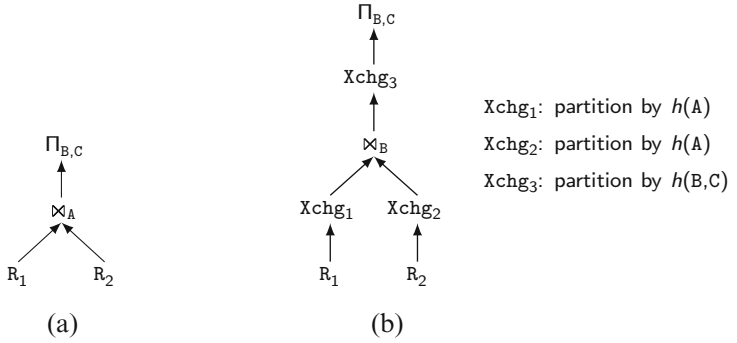

Fig. 8.15 Operator trie with exchange operators. (**a**) Sequential operator trie. (**b**) Parallel operator trie

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

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

Figure 8.15 shows an example of operator trie with exchange operators. The join operation is done by hashed partitioning of the input relations on A (operators $\texttt{Xchg}_1$ and $\texttt{Xchg}_2$) followed by local join. The project operations are done by duplicate elimination by hashing (operator $\texttt{Xchg}_3$), followed by local project.

8.4.2.2 Cost Model

Recall that the optimizer cost model is responsible for estimating the cost of a given execution plan. It consists of two parts: architecture-dependent and architecture-independent. The architecture-independent part is constituted by the cost functions for operator algorithms, e.g., nested loop for join and sequential access for select. If we ignore concurrency issues, only the cost functions for data repartitioning and memory consumption differ and constitute the architecture-dependent part. Indeed, repartitioning a relation's tuples in a shared-nothing system implies transfers of data across the interconnect, whereas it reduces to hashing in shared-memory systems. Memory consumption in the shared-nothing case is complicated by interoperator parallelism. In shared-memory systems, all operators read and write data through a global memory, and it is easy to test whether there is enough space to execute

them in parallel, i.e., the sum of the memory consumption of individual operators is less than the available memory. In shared-nothing, each processor has its own memory, and it becomes important to know which operators are executed in parallel on the same processor. Thus, for simplicity, we can assume that the set of processors (home) assigned to operators do not overlap, i.e., either the intersection of the set of processors is empty or the sets are identical.

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

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

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

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

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

8.4.2.3 Search Strategy

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

8.5 Load Balancing

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

8.5.1 Parallel Execution Problems

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

Initialization

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

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

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

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

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

The optimal number of processors (n_{opt}) is independent of n and only depends on the total processing time and initialization time. Thus, maximizing the degree of

parallelism for an operator, e.g., using all available processors, can hurt speed-up because of the overhead of initialization.

Interference

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

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

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

Skew

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

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

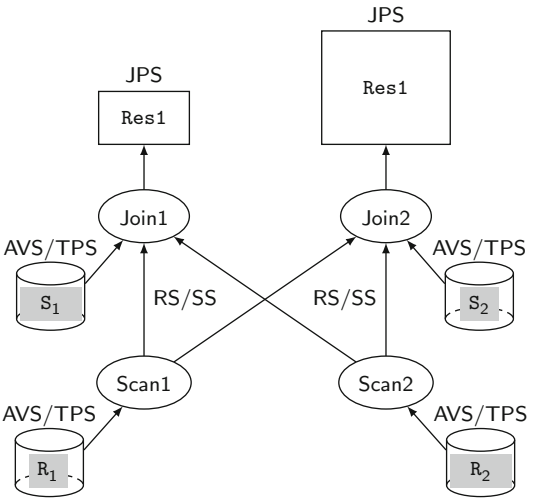

Fig. 8.16 Data skew example

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

8.5.2 Intraoperator Load Balancing

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

Adaptive Techniques

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

Specialized Techniques

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

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

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

8.5.3 Interoperator Load Balancing

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

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

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

8.5.4 Intraquery Load Balancing

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

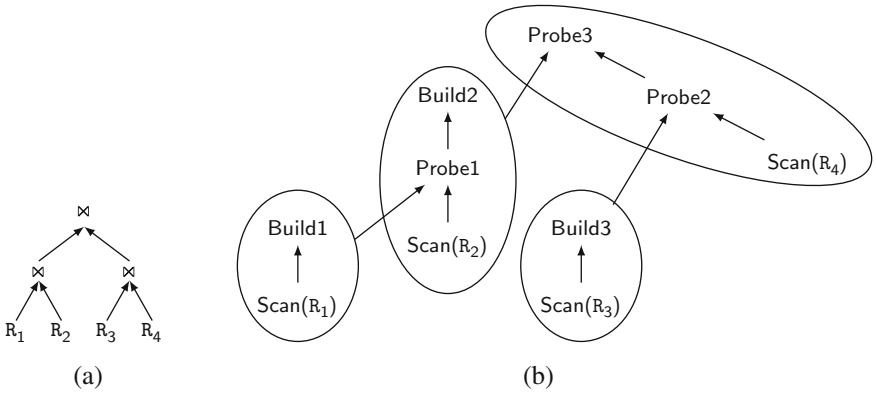

Fig. 8.17 A join trie and associated operator trie. (**a**) Join trie. (**b**) Operator trie (ellipses are pipeline chains)

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

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

Example 8.5 Figure 8.17 shows a join trie with four relations $\mathtt{R}_1$, $\mathtt{R}_2$, $\mathtt{R}_3$, and $\mathtt{R}_4$, and the corresponding operator trie 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:

$$\mathsf{Build1} \prec \mathsf{Probe1}$$
$$\mathsf{Build2} \prec \mathsf{Probe3}$$
$$\mathsf{Build3} \prec \mathsf{Probe2}$$

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

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

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

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

♦

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

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

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

Activations

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

the partition's hash table. Although activations are self-contained, they can only be executed on the SM-node where the associated data (hash tables or base relations) are.

Activation Queues

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

Threads

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

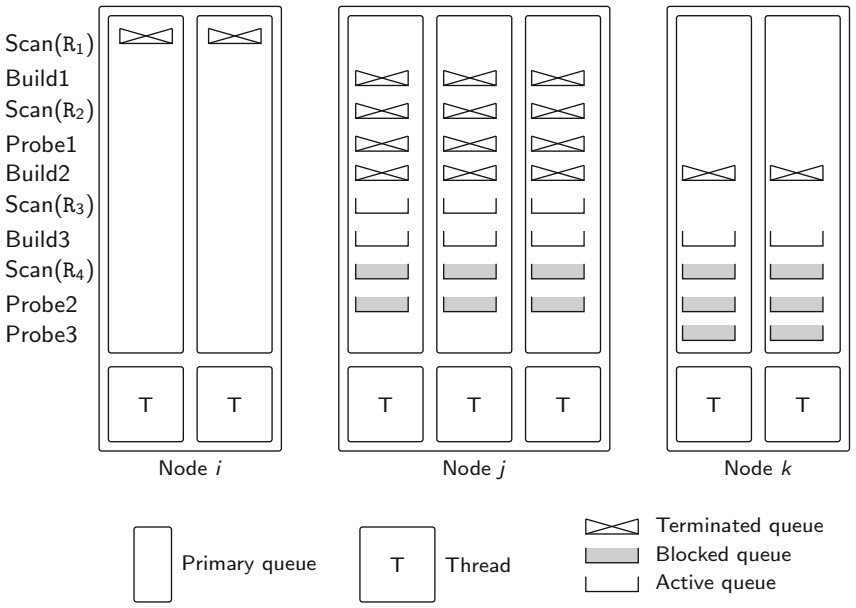

Fig. 8.18 Snapshot of an execution

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

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

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

8.6 Fault-Tolerance

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

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

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

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

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

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

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

8.7 Database Clusters

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

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

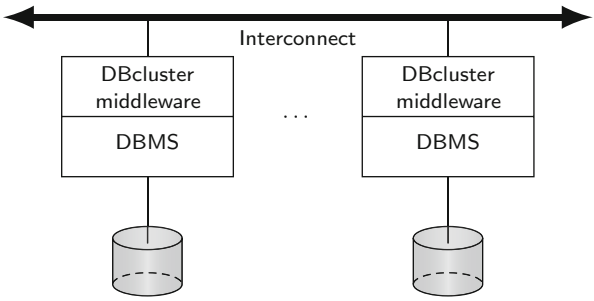

Fig. 8.19 A shared-nothing database cluster

8.7.1 Database Cluster Architecture

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

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

load balancing. Finally, the fault-tolerance manager provides online recovery and failover.

8.7.2 Replication

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

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

8.7.3 Load Balancing

In a database cluster, replication offers good load balancing opportunities. With eager or preventive replication (see Sect. 8.7.2), query load balancing is easy to achieve. Since all copies are mutually consistent, any node that stores a copy of the transaction data, e.g., the least loaded node, can be chosen at runtime by a

conventional load balancing strategy. Transaction load balancing is also easy in the case of lazy distributed replication since all master nodes need to eventually perform the transaction. However, the total cost of transaction execution at all nodes may be high. By relaxing consistency, lazy replication can better reduce transaction execution cost and thus increase performance of both queries and transactions. Thus, depending on the consistency/performance requirements, eager and lazy replication are both useful in database clusters.

8.7.4 Query Processing

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

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

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

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

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

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

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

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

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

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

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

Fine-grained virtual partitioning addresses these limitations by using a large number of subqueries instead of one per DBMS. Working with smaller subqueries avoids full table scans and makes query processing less vulnerable to DBMS idiosyncrasies. However, this approach must estimate the partition sizes, using