

$$\begin{aligned}
 \text{Total_cost} = & LT(\text{retrieve } \text{card}(S) \text{ tuples from } S) \\
 & + CT(\text{size}(S)) \\
 & + LT(\text{store } \text{card}(S) \text{ tuples in } T) \\
 & + LT(\text{retrieve } \text{card}(R) \text{ tuples from } R) \\
 & + LT(\text{retrieve } s \text{ tuples from } T) * \text{card}(R)
 \end{aligned}$$

Strategy 3.

Fetch tuples of the internal relation as needed for each tuple of the external relation. In this case, for each tuple in R , the join attribute value is sent to the site of S . Then the s tuples of S which match that value are retrieved and sent to the site of R to be joined as they arrive. Thus we have

$$\begin{aligned}
 \text{Total_cost} = & LT(\text{retrieve } \text{card}(R) \text{ tuples from } R) \\
 & + CT(\text{length}(A)) * \text{card}(R) \\
 & + LT(\text{retrieve } s \text{ tuples from } S) * \text{card}(R) \\
 & + CT(s * \text{length}(S)) * \text{card}(R)
 \end{aligned}$$

Strategy 4.

Move both relations to a third site and compute the join there. In this case the internal relation is first moved to a third site and stored in a temporary relation T . Then the external relation is moved to the third site and its tuples are joined with T as they arrive. Thus we have

$$\begin{aligned}
 \text{Total_cost} = & LT(\text{retrieve } \text{card}(S) \text{ tuples from } S) \\
 & + CT(\text{size}(S)) \\
 & + LT(\text{store } \text{card}(S) \text{ tuples in } T) \\
 & + LT(\text{retrieve } \text{card}(R) \text{ tuples from } R) \\
 & + CT(\text{size}(R)) \\
 & + LT(\text{retrieve } s \text{ tuples from } T) * \text{card}(R)
 \end{aligned}$$

Example 8.12. Let us consider a query that consists of the join of relations PROJ, the external relation, and ASG, the internal relation, on attribute PNO. We assume that PROJ and ASG are stored at two different sites and that there is an index on attribute PNO for relation ASG. The possible execution strategies for the query are as follows:

1. Ship whole PROJ to site of ASG.
2. Ship whole ASG to site of PROJ.
3. Fetch ASG tuples as needed for each tuple of PROJ.

4. Move ASG and PROJ to a third site.

The optimization algorithm predicts the total time of each strategy and selects the cheapest. Given that there is no operation following the join $\text{PROJ} \bowtie \text{ASG}$, strategy 4 obviously incurs the highest cost since both relations must be transferred. If $\text{size}(\text{PROJ})$ is much larger than $\text{size}(\text{ASG})$, strategy 2 minimizes the communication time and is likely to be the best if local processing time is not too high compared to strategies 1 and 3. Note that the local processing time of strategies 1 and 3 is probably much better than that of strategy 2 since they exploit the index on the join attribute.

If strategy 2 is not the best, the choice is between strategies 1 and 3. Local processing costs in both of these alternatives are identical. If PROJ is large and only a few tuples of ASG match, strategy 3 probably incurs the least communication time and is the best. Otherwise, that is, if PROJ is small or many tuples of ASG match, strategy 1 should be the best. ♦

Conceptually, the algorithm can be viewed as an exhaustive search among all alternatives that are defined by the permutation of the relation join order, join methods (including the selection of the join algorithm), result site, access path to the internal relation, and intersite transfer mode. Such an algorithm has a combinatorial complexity in the number of relations involved. Actually, the algorithm significantly reduces the number of alternatives by using dynamic programming and the heuristics, as does the System R's optimizer (see Section 8.2.2). With dynamic programming, the tree of alternatives is dynamically constructed and pruned by eliminating the inefficient choices.

Performance evaluation of the algorithm in the context of both high-speed networks (similar to local networks) and medium-speed wide area networks confirm the significant contribution of local processing costs, even for wide area networks [Lohman and Mackert, 1986; Mackert and Lohman, 1986]. It is shown in particular that for the distributed join, transferring the entire internal relation outperforms the fetch-as-needed method.

8.4.3 Semijoin-based Approach

We illustrate the semijoin-based approach with the algorithm of SDD-1 [Bernstein et al., 1981] which takes full advantage of the semijoin to minimize communication cost. The query optimization algorithm is derived from an earlier method called the “hill-climbing” algorithm [Wong, 1977], which has the distinction of being the first distributed query processing algorithm. In the hill-climbing algorithm, refinements of an initial feasible solution are recursively computed until no more cost improvements can be made. The algorithm does not use semijoins, nor does it assume data replication and fragmentation. It is devised for wide area point-to-point networks. The cost of transferring the result to the final site is ignored. This algorithm is quite general in that it can minimize an arbitrary objective function, including the total time and response time.

The hill-climbing algorithm proceeds as follows. The input to the algorithm includes the query graph, location of relations, and relation statistics. Following the completion of initial local processing, an initial feasible solution is selected which is a global execution schedule that includes all intersite communication. It is obtained by computing the cost of all the execution strategies that transfer all the required relations to a single candidate result site, and then choosing the least costly strategy. Let us denote this initial strategy as ES_0 . Then the optimizer splits ES_0 into two strategies, ES_1 followed by ES_2 , where ES_1 consists of sending one of the relations involved in the join to the site of the other relation. The two relations are joined locally and the resulting relation is transmitted to the chosen result site (specified as schedule ES_2). If the cost of executing strategies ES_1 and ES_2 , plus the cost of local join processing, is less than that of ES_0 , then ES_0 is replaced in the schedule by ES_1 and ES_2 . The process is then applied recursively to ES_1 and ES_2 until no more benefit can be gained. Notice that if n -way joins are involved, ES_0 will be divided into n subschedules instead of just two.

The hill-climbing algorithm is in the class of greedy algorithms, which start with an initial feasible solution and iteratively improve it. The main problem is that strategies with higher initial cost, which could nevertheless produce better overall benefits, are ignored. Furthermore, the algorithm may get stuck at a local minimum cost solution and fail to reach the global minimum.

Example 8.13. Let us illustrate the hill-climbing algorithm using the following query involving relations EMP, PAY, PROJ, and ASG of the engineering database:

“Find the salaries of engineers who work on the CAD/CAM project”

The query in relational algebra is

$$\Pi_{\text{SAL}} (\text{PAY} \bowtie_{\text{TITLE}} (\text{EMP} \bowtie_{\text{ENO}} (\text{ASG} \bowtie_{\text{PNO}} (\sigma_{\text{PNAME} = \text{"CAD/CAM"}}(\text{PROJ}))))))$$

We assume that $T_{MSG} = 0$ and $T_{TR} = 1$. Furthermore, we ignore the local processing, following which the database is

Relation	Size	Site
EMP	8	1
PAY	4	2
PROJ	1	3
ASG	10	4

To simplify this example, we assume that the length of a tuple (of every relation) is 1, which means that the size of a relation is equal to its cardinality. Furthermore, the placement of the relation is arbitrary. Based on join selectivities, we know that $\text{size}(\text{EMP} \bowtie \text{PAY}) = \text{size}(\text{EMP})$, $\text{size}(\text{PROJ} \bowtie \text{ASG}) = 2 * \text{size}(\text{PROJ})$, and $\text{size}(\text{ASG} \bowtie \text{EMP}) = \text{size}(\text{ASG})$.

Considering only data transfers, the initial feasible solution is to choose site 4 as the result site, producing the schedule

$$\begin{aligned}
 ES_0 : & \text{EMP} \rightarrow \text{site 4} \\
 & \text{PAY} \rightarrow \text{site 4} \\
 & \text{PROJ} \rightarrow \text{site 4} \\
 Total_cost(ES_0) = & 4 + 8 + 1 = 13
 \end{aligned}$$

This is true because the cost of any other solution is greater than the foregoing alternative. For example, if one chooses site 2 as the result site and transmits all the relations to that site, the total cost will be

$$\begin{aligned}
 Total_cost = & cost(\text{EMP} \rightarrow \text{site 2}) + cost(\text{ASG} \rightarrow \text{site 2}) \\
 & + cost(\text{PROJ} \rightarrow \text{site 2}) \\
 = & 19
 \end{aligned}$$

Similarly, the total cost of choosing either site 1 or site 3 as the result site is 15 and 22, respectively.

One way of splitting this schedule (call it ES') is the following:

$$\begin{aligned}
 ES_1 : & \text{EMP} \rightarrow \text{site 2} \\
 ES_2 : & (\text{EMP} \bowtie \text{PAY}) \rightarrow \text{site 4} \\
 ES_3 : & \text{PROJ} \rightarrow \text{site 4} \\
 Total_cost(ES') = & 8 + 8 + 1 = 17
 \end{aligned}$$

A second splitting alternative (ES'') is as follows:

$$\begin{aligned}
 ES_1 : & \text{PAY} \rightarrow \text{site 1} \\
 ES_2 : & (\text{PAY} \bowtie \text{EMP}) \rightarrow \text{site 4} \\
 ES_3 : & \text{PROJ} \rightarrow \text{site 4} \\
 Total_cost(ES'') = & 4 + 8 + 1 = 13
 \end{aligned}$$

Since the cost of either of the alternatives is greater than or equal to the cost of ES_0 , ES_0 is kept as the final solution. A better solution (ignored by the algorithm) is

$$\begin{aligned}
 B : & \text{PROJ} \rightarrow \text{site 4} \\
 ASG' = & (\text{PROJ} \bowtie \text{ASG}) \rightarrow \text{site 1} \\
 & (\text{ASG}' \bowtie \text{EMP}) \rightarrow \text{site 2} \\
 Total_cost(B) = & 1 + 2 + 2 = 5
 \end{aligned}$$



The semijoin-based algorithm extends the hill-climbing algorithm in a number of ways [Bernstein et al., 1981]. In addition to the extensive use of semijoins, the objective function is expressed in terms of total communication time (local time and response time are not considered). Furthermore, the algorithm uses statistics on the database, called *database profiles*, where a profile is associated with a relation. The algorithm also selects an initial feasible solution that is iteratively refined. Finally, a postoptimization step is added to improve the total time of the solution selected. The main step of the algorithm consists of determining and ordering beneficial semijoins, that is semijoins whose cost is less than their benefit.

The cost of a semijoin is that of transferring the semijoin attributes A ,

$$Cost(R \ltimes_A S) = T_{MSG} + T_{TR} * size(\Pi_A(S))$$

while its benefit is the cost of transferring irrelevant tuples of R (which is avoided by the semijoin):

$$Benefit(R \ltimes_A S) = (1 - SF_{SJ}(S.A)) * size(R) * T_{TR}$$

The semijoin-based algorithm proceeds in four phases: initialization, selection of beneficial semijoins, assembly site selection, and postoptimization. The output of the algorithm is a global strategy for executing the query (Algorithm 8.6).

Algorithm 8.6: Semijoin-based-QOA

Input: QG : query graph with n relations; statistics for each relation
Output: ES : execution strategy

```

begin
     $ES \leftarrow$  local-operations ( $QG$ ) ;
    modify statistics to reflect the effect of local processing ;
     $BS \leftarrow \emptyset$ ;                                {set of beneficial semijoins}
    for each semijoin  $SJ$  in  $QG$  do
        if  $cost(SJ) < benefit(SJ)$  then
             $BS \leftarrow BS \cup SJ$ 
    while  $BS \neq \emptyset$  do
         $SJ \leftarrow most\_beneficial(BS)$ ; { $SJ$ : semijoin with  $\max(benefit - cost)$ }
         $BS \leftarrow BS - SJ$ ;                  {remove  $SJ$  from  $BS$ }
         $ES \leftarrow ES + SJ$ ;                {append  $SJ$  to execution strategy}
        modify statistics to reflect the effect of incorporating  $SJ$  ;
         $BS \leftarrow BS -$  non-beneficial semijoins ;
         $BS \leftarrow BS \cup$  new beneficial semijoins ;
    {assembly site selection}
     $AS(ES) \leftarrow$  select site  $i$  such that  $i$  stores the largest amount of data after all
    local operations ;
     $ES \leftarrow ES \cup$  transfers of intermediate relations to  $AS(ES)$  ;
    {postoptimization}
    for each relation  $R_i$  at  $AS(ES)$  do
        for each semijoin  $SJ$  of  $R_i$  by  $R_j$  do
            if  $cost(ES) > cost(ES - SJ)$  then
                 $ES \leftarrow ES - SJ$ 
end

```

The initialization phase generates a set of beneficial semijoins, $BS = \{SJ_1, SJ_2, \dots, SJ_k\}$, and an execution strategy ES that includes only local processing. The next phase selects the beneficial semijoins from BS by iteratively choosing the most

beneficial semijoin, SJ_i , and modifying the database statistics and BS accordingly. The modification affects the statistics of relation R involved in SJ_i and the remaining semijoins in BS that use relation R . The iterative phase terminates when all semijoins in BS have been appended to the execution strategy. The order in which semijoins are appended to ES will be the execution order of the semijoins.

The next phase selects the assembly site by evaluating, for each candidate site, the cost of transferring to it all the required data and taking the one with the least cost. Finally, a postoptimization phase permits the removal from the execution strategy of those semijoins that affect only relations stored at the assembly site. This phase is necessary because the assembly site is chosen after all the semijoins have been ordered. The SDD-1 optimizer is based on the assumption that relations can be transmitted to another site. This is true for all relations except those stored at the assembly site, which is selected after beneficial semijoins are considered. Therefore, some semijoins may incorrectly be considered beneficial. It is the role of postoptimization to remove them from the execution strategy.

Example 8.14. Let us consider the following query:

```
SELECT R3.C
FROM   R1,R2,R3
WHERE  R1.A = R2.A
AND    R2.B = R3.B
```

Figure 8.15 gives the join graph of the query and of relation statistics. We assume that $T_{MSG} = 0$ and $T_{TR} = 1$. The initial set of beneficial semijoins will contain the following two:

$SJ_1: R_2 \ltimes R_1$, whose benefit is $2100 = (1 - 0.3) * 3000$ and cost is 36
 $SJ_2: R_2 \ltimes R_3$, whose benefit is $1800 = (1 - 0.4) * 3000$ and cost is 80

Furthermore there are two non-beneficial semijoins:

$SJ_3: R_1 \ltimes R_2$, whose benefit is $300 = (1 - 0.8) * 1500$ and cost is 320
 $SJ_4: R_3 \ltimes R_2$, whose benefit is 0 and cost is 400.

At the first iteration of the selection of beneficial semijoins, SJ_1 is appended to the execution strategy ES . One effect on the statistics is to change the size of R_2 to $900 = 3000 * 0.3$. Furthermore, the semijoin selectivity factor of attribute $R_2.A$ is reduced because $card(\Pi_A(R_2))$ is reduced. We approximate $SF_{SJ}(R_2.A)$ by $0.8 * 0.3 = 0.24$. Finally, size of $\Pi_{R_2.A}$ is also reduced to $96 = 320 * 0.3$. Similarly, the semijoin selectivity factor of attribute $R_2.B$ and $\Pi_{R_2.B}$ should also be reduced (but they not needed in the rest of the example).

At the second iteration, there are two beneficial semijoins:

$SJ_2: R'_2 \ltimes R_3$, whose benefit is $540 = 900 * (1 - 0.4)$ and cost is 80
 (here $R'_2 = R_2 \ltimes R_1$, which is obtained by SJ_1)
 $SJ_3: R_1 \ltimes R'_2$, whose benefit is $1140 = (1 - 0.24) * 1500$ and cost is 96

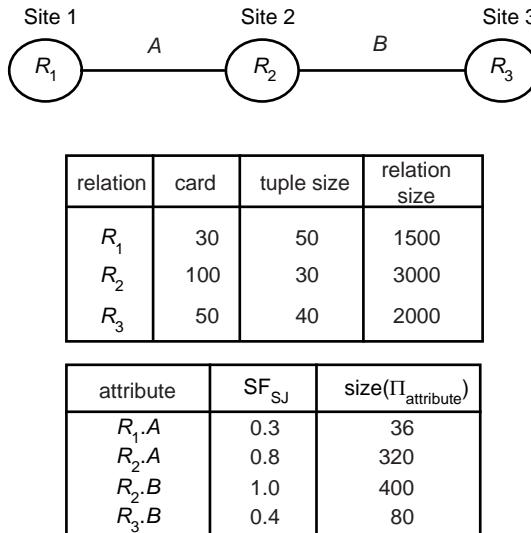


Fig. 8.15 Example Query and Statistics

The most beneficial semijoin is SJ_3 and is appended to ES . One effect on the statistics of relation R_1 is to change the size of R_1 to 360 ($= 1500 * 0.24$). Another effect is to change the selectivity of R_1 and size of $\Pi_{R_1.A}$.

At the third iteration, the only remaining beneficial semijoin, SJ_2 , is appended to ES . Its effect is to reduce the size of relation R_2 to 360 ($= 900 * 0.4$). Again, the statistics of relation R_2 may also change.

After reduction, the amount of data stored is 360 at site 1, 360 at site 2, and 2000 at site 3. Site 3 is therefore chosen as the assembly site. The postoptimization does not remove any semijoin since they all remain beneficial. The strategy selected is to send $(R_2 \ltimes R_1) \ltimes R_3$ and $R_1 \ltimes R_2$ to site 3, where the final result is computed. ♦

Like its predecessor hill-climbing algorithm, the semijoin-based algorithm selects locally optimal strategies. Therefore, it ignores the higher-cost semijoins which would result in increasing the benefits and decreasing the costs of other semijoins. Thus this algorithm may not be able to select the global minimum cost solution.

8.4.4 Hybrid Approach

The static and dynamic distributed optimization approaches have the same advantages and disadvantages as in centralized systems (see Section 8.2.3). However, the problems of accurate cost estimation and comparison of QEPs at compile-time are much more severe in distributed systems. In addition to unknown bindings of parameter values in embedded queries, sites may become unavailable or overloaded at

runtime. In addition, relations (or relation fragments) may be replicated at several sites. Thus, site and copy selection should be done at runtime to increase availability and load balancing of the system.

The hybrid query optimization technique using dynamic QEPs (see Section 8.2.3) is general enough to incorporate site and copy selection decisions. However, the search space of alternative subplans linked by choose-plan operators becomes much larger and may result in heavy static plans and much higher startup time. Therefore, several hybrid techniques have been proposed to optimize queries in distributed systems [Carey and Lu, 1986; Du et al., 1995; Evrendilek et al., 1997]. They essentially rely on the following two-step approach:

1. At compile time, generate a static plan that specifies the ordering of operations and the access methods, without considering where relations are stored.
2. At startup time, generate an execution plan by carrying out site and copy selection and allocating the operations to the sites.

Example 8.15. Consider the following query expressed in relational algebra:

$$\sigma(R_1) \bowtie R_2 \bowtie R_3$$

Figure 8.16 shows a 2-step plan for this query. The static plan shows the relational operation ordering as produced by a centralized query optimizer. The run-time plan extends the static plan with site and copy selection and communication between sites. For instance, the first selection is allocated at site s_1 on copy R_{11} of relation R_1 and sends its result to site s_3 to be joined with R_{23} and so on. ♦

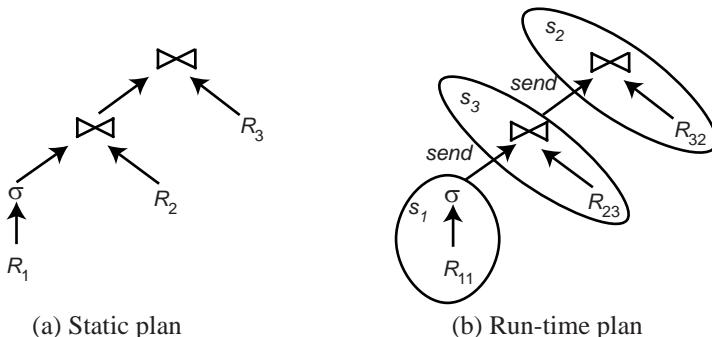


Fig. 8.16 A 2-Step Plan

The first step can be done by a centralized query optimizer. It may also include choose-plan operators so that runtime bindings can be used at startup time to make accurate cost estimations. The second step carries out site and copy selection, possibly in addition to choose-plan operator execution. Furthermore, it can optimize the load

balancing of the system. In the rest of this section, we illustrate this second step based on the seminal paper by [Carey and Lu \[1986\]](#) on two-step query optimization.

We consider a distributed database system with a set of sites $S = \{s_1, \dots, s_n\}$. A query Q is represented as an ordered sequence of subqueries $Q = \{q_1, \dots, q_m\}$. Each subquery q_i is the maximum processing unit that accesses a single base relation and communicates with its neighboring subqueries. For instance, in Figure 8.16, there are three subqueries, one for R_1 , one for R_2 , and one for R_3 . Each site s_i has a load, denoted by $load(s_i)$, which reflects the number of queries currently submitted. The load can be expressed in different ways, e.g. as the number of I/O bound and CPU bound queries at the site [\[Carey and Lu, 1986\]](#). The average load of the system is defined as:

$$\text{Avg_load}(S) = \frac{\sum_{i=1}^n load(s_i)}{n}$$

The balance of the system for a given allocation of subqueries to sites can be measured as the variance of the site loads using the following *unbalance factor* [\[Carey and Lu, 1986\]](#):

$$UF(S) = \frac{1}{n} \sum_{i=1}^n (load(s_i) - \text{Avg_load}(S))^2$$

As the system gets balanced, its unbalance factor approaches 0 (perfect balance). For example, with $load(s_1)=10$ and $load(s_1)=30$, the unbalance factor of s_1, s_2 is 100 while with $load(s_1)=20$ and $load(s_1)=20$, it is 0.

The problem addressed by the second step of two-step query optimization can be formalized as the following subquery allocation problem. Given

1. a set of sites $S = \{s_1, \dots, s_n\}$ with the load of each site;
2. a query $Q = \{q_1, \dots, q_m\}$; and
3. for each subquery q_i in Q , a feasible allocation set of sites $S_q = \{s_1, \dots, s_k\}$ where each site stores a copy of the relation involved in q_i ;

the objective is to find an optimal allocation on Q to S such that

1. $UF(S)$ is minimized, and
2. the total communication cost is minimized.

[Carey and Lu \[1986\]](#) propose an algorithm that finds near-optimal solutions in a reasonable amount of time. The algorithm, which we describe in Algorithm 8.7 for linear join trees, uses several heuristics. The first heuristic (step 1) is to start by allocating subqueries with least allocation flexibility, i.e. with the smaller feasible allocation sets of sites. Thus, subqueries with a few candidate sites are allocated earlier. Another heuristic (step 2) is to consider the sites with least load and best benefit. The benefit of a site is defined as the number of subqueries already allocated to the site and measures the communication cost savings from allocating the subquery

to the site. Finally, in step 3 of the algorithm, the load information of any unallocated subquery that has a selected site in its feasible allocation set is recomputed.

Algorithm 8.7: SQAllocation

Input: $Q: q_1, \dots, q_m$;
 Feasible allocation sets: S_{q_1}, \dots, S_{q_m} ;
 Loads: $load(S_1), \dots, load(S_m)$;
Output: an allocation of Q to S

begin

for each q in Q **do**
 └ compute($load(S_q)$)

while Q not empty **do**
 $a \leftarrow q \in Q$ with least allocation flexibility; {select subquery a for allocation} (1)
 $b \leftarrow s \in S_a$ with least load and best benefit; {select best site b for a } (2)
 $Q \leftarrow Q - a$;
 {recompute loads of remaining feasible allocation sets if necessary} (3)
 for each $q \in Q$ where $b \in S_q$ **do**
 └ compute($load(S_q)$)

end

Example 8.16. Consider the following query Q expressed in relational algebra:

$$\sigma(R_1) \bowtie R_2 \bowtie R_3 \bowtie R_4$$

Figure 8.17 shows the placement of the copies of the 4 relations at the 4 sites, and the site loads. We assume that Q is decomposed as $Q = \{q_1, q_2, q_3, q_4\}$ where q_1 is associated with R_1 , q_2 with R_2 joined with the result of q_1 , q_3 with R_3 joined with the result of q_2 , and q_4 with R_4 joined with the result of q_3 . The SQAllocation algorithm performs 4 iterations. At the first one, it selects q_4 which has the least allocation flexibility, allocates it to s_1 and updates the load of s_1 to 2. At the second iteration, the next set of subqueries to be selected are either q_2 or q_3 since they have the same allocation flexibility. Let us choose q_2 and assume it gets allocated to s_2 (it could be allocated to s_4 which has the same load as s_2). The load of s_2 is increased to 3. At the third iteration, the next subquery selected is q_3 and it is allocated to s_1 which has the same load as s_3 but a benefit of 1 (versus 0 for s_3) as a result of the allocation of q_4 . The load of s_1 is increased to 3. Finally, at the last iteration, q_1 gets allocated to either s_3 or s_4 which have the least loads. If in the second iteration q_2 were allocated to s_4 instead of to s_2 , then the fourth iteration would have allocated q_1 to s_4 because of a benefit of 1. This would have produced a better execution plan with less communication. This illustrates that two-step optimization can still miss optimal plans. ♦

sites	load	R_1	R_2	R_3	R_4
s_1	1	R_{11}		R_{31}	R_{41}
s_2	2		R_{22}		
s_3	2	R_{13}		R_{33}	
s_4	2	R_{14}	R_{24}		

Fig. 8.17 Example Data Placement and Load

This algorithm has reasonable complexity. It considers each subquery in turn, considering each potential site, selects a current one for allocation, and sorts the list of remaining subqueries. Thus, its complexity can be expressed as $O(\max(m*n, m^2 * \log_2 m))$.

Finally, the algorithm includes a refining phase to further optimize join processing and decide whether or not to use semijoins. Although it minimizes communication given a static plan, two-step query optimization may generate runtime plans that have higher communication cost than the optimal plan. This is because the first step is carried out ignoring data location and its impact on communication cost. For instance, consider the runtime plan in 8.16 and assume that the third subquery on R_3 is allocated to site s_1 (instead of site s_2). In this case, the plan that does the join (or Cartesian product) of the result of the selection of R_1 with R_3 first at site s_1 may be better since it minimizes communication. A solution to this problem is to perform plan reorganization using operation tree transformations at startup time [Du et al., 1995].

8.5 Conclusion

In this chapter we have presented the basic concepts and techniques for distributed query optimization. We first introduced the main components of query optimization, including the search space, the cost model and the search strategy. The details of the environment (centralized versus distributed) are captured by the search space and the cost model. The search space describes the equivalent execution plans for the input query. These plans differ on the execution order of operations and their implementation, and therefore on performance. The search space is obtained by applying transformation rules, such as those described in Section 7.1.4.

The cost model is key to estimating the cost of a given execution plan. To be accurate, the cost model must have good knowledge about the distributed execution environment. Important inputs are the database statistics and the formulas used to estimate the size of intermediate results. For simplicity, earlier cost models relied on the strong assumption that the distribution of attribute values in a relation is uniform. However, in case of skewed data distributions, this can result in fairly inaccurate estimations and execution plans which are far from the optimal. An

effective solution to accurately capture data distributions is to use histograms. Today, most commercial DBMS optimizers support histograms as part of their cost model. A difficulty remains to estimate the selectivity of the join operation when it is not on foreign key. In this case, maintaining join selectivity factors is of great benefit [Mackert and Lohman, 1986]. Earlier distributed DBMSs considered transmission costs only. With the availability of faster communication networks, it is important to consider local processing costs as well.

The search strategy explores the search space and selects the best plan, using the cost model. It defines which plans are examined and in which order. The most popular search strategy is dynamic programming which enumerates all equivalent execution plans with some pruning. However, it may incur a high optimization cost for queries involving large number of relations. Thus, it is best suited when optimization is static (done at compile time) and amortized over multiple executions. Randomized strategies, such as Iterative Improvement and Simulated Annealing, have received much attention. They do not guarantee that the best solution is obtained, but avoid the high cost of optimization. Thus, they are appropriate for ad-hoc queries which are not repetitive.

As a prerequisite to understanding distributed query optimization, we have introduced centralized query optimization with the three basic techniques: dynamic, static and hybrid. Dynamic and static query optimization both have advantages and drawbacks. Dynamic query optimization can make accurate optimization choices at run-time, but optimization is repeated for each query execution. Therefore, this approach is best for ad-hoc queries. Static query optimization, done at compilation time, is best for queries embedded in stored procedures, and has been adopted by all commercial DBMSs. However, it can make major estimation errors, in particular, in the case of parameter values not known until runtime, which can lead to the choice of suboptimal execution plans. Hybrid query optimization attempts to provide the advantages of static query optimization while avoiding the issues generated by inaccurate estimates. The approach is basically static, but further optimization decisions may take place at run time.

Next, we have seen two approaches to solve distributed join queries, which are the most important type of queries. The first one considers join ordering. The second one computes joins with semijoins. Semijoins are beneficial only when a join has good selectivity, in which case the semijoins act as powerful size reducers. The first systems that make extensive use of semijoins assumed a slow network and therefore concentrated on minimizing only the communication time at the expense of local processing time. However, with faster networks, the local processing time is as important as the communication time and sometimes even more important. Therefore, semijoins should be employed carefully since they tend to increase the local processing time. Join and semijoin techniques should be considered complementary, not alternative [Valduriez and Gardarin, 1984], because each technique may be better under certain database-dependent parameters. For instance, if a relation has very large tuples, as is the case with multimedia data, semijoin is useful to minimize data transfers. Finally, semijoins implemented by hashed bit arrays [Valduriez, 1982] can be made very efficient [Mackert and Lohman, 1986].

We illustrated the use of the join and semijoin techniques in four basic distributed query optimization algorithms: dynamic, static, semijoin-based and hybrid. The static and dynamic distributed optimization approaches have the same advantages and disadvantages as in centralized systems. The semijoin-based approach is best for slow networks. The hybrid approach is best in today's dynamic environments as it delays important decisions such as copy selection and allocation of subqueries to sites at query startup time. Thus, it can better increase availability and load balancing of the system. We illustrated the hybrid approach with two-step query optimization which first generates a static plan that specifies the operations ordering as in a centralized system and then generates an execution plan at startup time, by carrying out site and copy selection and allocating the operations to the sites.

In this chapter we focused mostly on join queries for two reasons: join queries are the most frequent queries in the relational framework and they have been studied extensively. Furthermore, the number of joins involved in queries expressed in languages of higher expressive power than relational calculus (e.g., Horn clause logic) can be extremely large, making the join ordering more crucial [Krishnamurthy et al., 1986]. However, the optimization of general queries containing joins, unions, and aggregate functions is a harder problem [Selinger and Adiba, 1980]. Distributing unions over joins is a simple and good approach since the query can be reduced as a union of join subqueries, which are optimized individually. Note also that the unions are more frequent in distributed DBMSs because they permit the localization of horizontally fragmented relations.

8.6 Bibliographic Notes

Good surveys of query optimization are provided in [Graefe, 1993], [Ioannidis, 1996] and [Chaudhuri, 1998]. Distributed query optimization is surveyed in [Kossmann, 2000].

The three basic algorithms for query optimization in centralized systems are: the dynamic algorithm of INGRES [Wong and Youssefi, 1976] which performs query reduction, the static algorithm of System R [Selinger et al., 1979] which uses dynamic programming and a cost model and the hybrid algorithm of Volcano [Cole and Graefe, 1994] which uses choose-plan operators.

The theory of semijoins and their value for distributed query processing has been covered in [Bernstein and Chiu, 1981], [Chiu and Ho, 1980], and [Kambayashi et al., 1982]. Algorithms for improving the processing of semijoins in distributed systems are proposed in [Valduriez, 1982]. The value of semijoins for multiprocessor database machines having fast communication networks is also shown in [Valduriez and Gardarin, 1984]. Parallel execution strategies for horizontally fragmented databases is treated in [Ceri and Pelagatti, 1983] and [Khoshafian and Valduriez, 1987]. The solutions in [Shasha and Wang, 1991] are also applicable to parallel systems.

The dynamic approach to distributed query optimization was first proposed for Distributed INGRES in [Epstein et al., 1978]. It extends the dynamic algorithm

of INGRES, with a heuristic approach. The algorithm takes advantage of the network topology (general or broadcast networks). Improvements on this method based on the enumeration of all possible solutions are given and analyzed in [Epstein and Stonebraker, 1980].

The static approach to distributed query optimization was first proposed for R* in [Selinger and Adiba, 1980] as an extension of the static algorithm of System R. It is one of the first papers to recognize the significance of local processing on the performance of distributed queries. Experimental validation in [Lohman and Mackert, 1986] have confirmed this important statement.

The semijoin-based approach to distributed query optimization was proposed in [Bernstein et al., 1981] for SDD-1 [Wong, 1977]. It is one of the most complete algorithms which make full use of semijoins.

Several hybrid approaches based on two-step query optimization have been proposed for distributed systems [Carey and Lu, 1986; Du et al., 1995; Evrendilek et al., 1997]. The content of Section 8.4.4 is based on [Carey and Lu, 1986] which is the first paper on two-step query optimization. In [Du et al., 1995], efficient operations to transform linear join trees (produced by the first step) into bushy trees which exhibit more parallelism are proposed. In [Evrendilek et al., 1997], a solution to maximize intersite join parallelism in the second step is proposed.

Exercises

Problem 8.1 (*). Apply the dynamic query optimization algorithm in Section 8.2.1 to the query of Exercise 7.3, and illustrate the successive detachments and substitutions by giving the monorelation subqueries generated.

Problem 8.2. Consider the join graph of Figure 8.12 and the following information: $\text{size}(\text{EMP}) = 100$, $\text{size}(\text{ASG}) = 200$, $\text{size}(\text{PROJ}) = 300$, $\text{size}(\text{EMP} \bowtie \text{ASG}) = 300$, and $\text{size}(\text{ASG} \bowtie \text{PROJ}) = 200$. Describe an optimal join program based on the objective function of total transmission time.

Problem 8.3. Consider the join graph of Figure 8.12 and make the same assumptions as in Problem 8.2. Describe an optimal join program that minimizes response time (consider only communication).

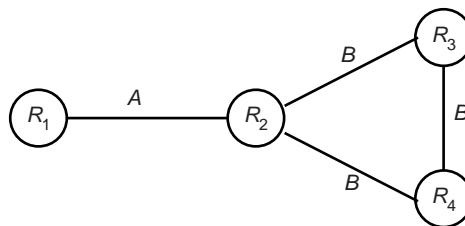
Problem 8.4. Consider the join graph of Figure 8.12, and give a program (possibly not optimal) that reduces each relation fully by semijoins.

Problem 8.5 (*). Consider the join graph of Figure 8.12 and the fragmentation depicted in Figure 8.18. Also assume that $\text{size}(\text{EMP} \bowtie \text{ASG}) = 2000$ and $\text{size}(\text{ASG} \bowtie \text{PROJ}) = 1000$. Apply the dynamic distributed query optimization algorithm in Section 8.4.1 in two cases, general network and broadcast network, so that communication time is minimized.

Rel.	Site 1	Site 2	Site 3
EMP	1000	1000	1000
ASG		2000	
PROJ	1000		

Fig. 8.18 Fragmentation

Problem 8.6. Consider the join graph of Figure 8.19 and the statistics given in Figure 8.20. Apply the semijoin-based distributed query optimization algorithm in Section 8.4.3 with $T_{MSG} = 20$ and $T_{TR} = 1$.

**Fig. 8.19** Join Graph

relation	size	attribute	size	SF_{SJ}
R_1	1000	$R_1.A$	200	0.5
R_2	1000	$R_2.A$	100	0.1
R_3	2000	$R_2.A$	100	0.2
R_3	1000	$R_3.B$	300	0.9
		$R_4.B$	150	0.4

(a)

(b)

Fig. 8.20 Relation Statistics

Problem 8.7 ().** Consider the query in Problem 7.5. Assume that relations EMP, ASG, PROJ and PAY have been stored at sites 1, 2, and 3 according to the table in Figure 8.21. Assume also that the transfer rate between any two sites is equal and that data transfer is 100 times slower than data processing performed by any site. Finally, assume that $\text{size}(R \bowtie S) = \max(\text{size}(R), \text{size}(S))$ for any two relations R and S , and the selectivity factor of the disjunctive selection of the query in Exercise 7.5 is

0.5. Compose a distributed program which computes the answer to the query and minimizes total time.

Rel.	Site 1	Site 2	Site 3
EMP	2000		
ASG		3000	
PROJ			1000
PAY			500

Fig. 8.21 Fragmentation Statistics

Problem 8.8 ().** In Section 8.4.4, we described Algorithm 8.7 for linear join trees. Extend this algorithm to support bushy join trees. Apply it to the bushy join tree in Figure 8.3 using the data placement and site loads shown in Figure 8.17.

Chapter 9

Multidatabase Query Processing

In the previous three chapters, we have considered query processing in tightly-coupled homogeneous distributed database systems. As we discussed in Chapter 1, these systems are logically integrated and provide a single image of the database, even though they are physically distributed. In this chapter, we concentrate on query processing in multidatabase systems that provide interoperability among a set of DBMSs. This is only one part of the more general *interoperability* problem. Distributed applications pose major requirements regarding the databases they access, in particular, the ability to access legacy data as well as newly developed databases. Thus, providing integrated access to multiple, distributed databases and other heterogeneous data sources has become a topic of increasing interest and focus.

Many of the distributed query processing and optimization techniques carry over to multidatabase systems, but there are important differences. Recall from Chapter 6 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, a multi-DBMS layer (see Figure 1.17) is necessary to communicate with component DBMSs in an effective way, and this requires additional query processing steps (Figure 9.1). Furthermore, there may be many component DBMSs, each of which may exhibit different behavior, thereby posing new requirements for more adaptive query processing techniques.

This chapter is organized as follows. In Section 9.1 we introduce in more detail the main issues in multidatabase query processing. Assuming the mediator/wrapper architecture, we describe the multidatabase query processing architecture in Section 9.2. Section 9.3 describes the techniques for rewriting queries using multidatabase views. Section 9.4 describes multidatabase query optimization and execution, in particular, heterogeneous cost modeling, heterogeneous query optimization, and adaptive query processing. Section 9.5 describes query translation and execution at the wrappers, in particular, the techniques for translating queries for execution by the component DBMSs and for generating and managing wrappers.

9.1 Issues in Multidatabase Query Processing

Query processing in a multidatabase system is more complex than in a distributed DBMS for the following reasons [Sheth and Larson, 1990]:

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.
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, XML, 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 [Lu et al., 1993]. 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 Figure 1.17 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 are 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 multi-DBMS: the multi-DBMS layer at the control site (i.e., the mediator) receives the global query, the multi-DBMS layers at the sites (i.e., the wrappers) participate in processing the query, and the component DBMSs ultimately optimize and execute the query.

9.2 Multidatabase Query Processing Architecture

Most of the work on multidatabase query processing has been done in the context of the mediator/wrapper architecture (see Figure 1.18). 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 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 (based on XML). 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 Figure 9.1. 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 Figure 6.3). 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 meta-information stored in the global directory (global schema, allocation and capability schema). Query rewriting transforms the input query into a query on local relations, using the global schema. Recall from Chapter 4 that there are two main

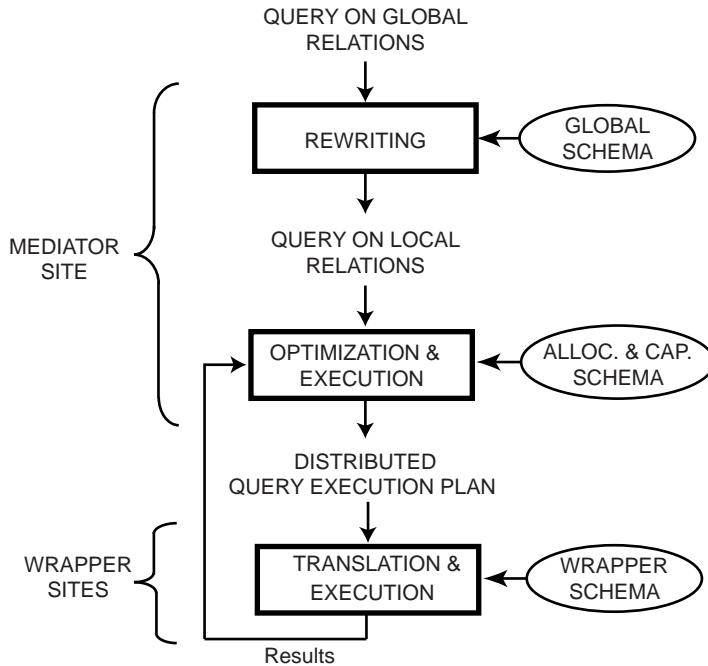


Fig. 9.1 Generic Layering Scheme for Multidatabase Query Processing

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 [Ullman, 1988] which 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 schema 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) make dynamic query optimization more critical. In the case of dynamic optimization, there may be subsequent calls to this layer after execution by the next layer. This is illustrated by the arrow showing results coming from the next 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 (see Chapter 4) 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 schema contains information describing 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 schema 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 schema.

9.3 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., global-as-view (GAV) or local-as-view (LAV). In particular, the techniques for LAV (and its extension GLAV) are much more involved [Halevy, 2001]. Most of the work on query rewriting using views has been done using Datalog [Ullman, 1988], 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.

9.3.1 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:

$$Q(T) : -R_1(T_1), \dots, R_n(T_n)$$

The atom $Q(T)$ is the *head* of the query and denotes the result relation. The atoms $R_1(T_1), \dots, R_n(T_n)$ are the *subgoals* in the body of the query and denote database relations. Q and R_1, \dots, R_n are predicate names and correspond to relation names. T, T_1, \dots, 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 which 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 9.1. Let us consider relations $\text{EMP}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY})$ and $\text{ASG}(\text{ENO}, \text{PNO}, \text{DUR})$ assuming that ENO is the primary key of EMP and (ENO, PNO) is the primary key of ASG . Consider the following SQL query:

```
SELECT ENO, TITLE, PNO
FROM   EMP, ASG
WHERE  EMP.ENO = ASG.ENO
AND    TITLE = "Programmer" OR DUR = 24
```

The corresponding query in Datalog can be expressed as:

```
Q(ENO, TITLE, PNO) : - EMP(ENO, ENAME, "Programmer", CITY),
                      ASG(ENO, PNO, DUR)
Q(ENO, TITLE, PNO) : - EMP(ENO, ENAME, TITLE, CITY),
                      ASG(ENO, PNO, 24)
```



9.3.2 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 pre-exist 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 9.2. Let us consider the local relations $\text{EMP1}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY})$, $\text{EMP2}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY})$ and $\text{ASG1}(\text{ENO}, \text{PNO}, \text{DUR})$. The global relations $\text{EMP}(\text{ENO}, \text{ENAME}, \text{CITY})$ and $\text{ASG}(\text{ENO}, \text{PNO}, \text{TITLE}, \text{DUR})$ can be simply defined with the following Datalog rules:

$$\text{EMP}(\text{ENO}, \text{ENAME}, \text{CITY}) : -\text{EMP1}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY}) \quad (r_1)$$

$$\text{EMP}(\text{ENO}, \text{ENAME}, \text{CITY}) : -\text{EMP2}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY}) \quad (r_2)$$

$$\begin{aligned} \text{ASG}(\text{ENO}, \text{PNO}, \text{TITLE}, \text{DUR}) : & -\text{EMP1}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY}), \\ & \text{ASG1}(\text{ENO}, \text{PNO}, \text{DUR}) \end{aligned} \quad (r_3)$$

$$\begin{aligned} \text{ASG}(\text{ENO}, \text{PNO}, \text{TITLE}, \text{DUR}) : & -\text{EMP2}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY}), \\ & \text{ASG1}(\text{ENO}, \text{PNO}, \text{DUR}) \end{aligned} \quad (r_4)$$

◆

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 Section 7.2). The rewriting technique using views is called *unfolding* [Ullman, 1997], 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 9.2), unfolding can generate redundant queries that need to be eliminated.

Example 9.3. Let us consider the global schema in Example 9.2 and the following query Q that asks for assignment information about the employees living in “Paris”:

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

Unfolding Q produces Q' as follows:

$$Q'(e, p) : -\text{EMP1}(e, \text{ENAME}, \text{TITLE}, \text{“Paris”}), \text{ASG1}(e, p, \text{DUR}). \quad (q_1)$$

$$Q'(e, p) : -\text{EMP2}(e, \text{ENAME}, \text{TITLE}, \text{“Paris”}), \text{ASG1}(e, p, \text{DUR}). \quad (q_2)$$

Q' is the union of two conjunctive queries labeled as q_1 and q_2 . q_1 is obtained by applying rule r_3 or both rules r_1 and r_3 . In the latter case, the query obtained is redundant with respect to that obtained with r_3 only. Similarly, q_2 is obtained by applying rule r_4 or both rules r_2 and r_4 . ◆

Although the basic technique is simple, rewriting in GAV becomes difficult when local databases have limited access patterns [Calì and Calvanese, 2002]. 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.

9.3.3 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 9.4. To facilitate comparison with GAV, we develop an example that is symmetric to Example 9.2 with $\text{EMP}(\text{ENO}, \text{ENAME}, \text{CITY})$ and $\text{ASG}(\text{ENO}, \text{PNO}, \text{TITLE}, \text{DUR})$ as global relations. In the LAV approach, the local relations $\text{EMP1}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY})$, $\text{EMP2}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY})$ and $\text{ASG1}(\text{ENO}, \text{PNO}, \text{DUR})$ can be defined with the following Datalog rules:

```

 $\text{EMP1}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY}) : -\text{EMP}(\text{ENO}, \text{ENAME}, \text{CITY}), \quad (r_1)$ 
 $\text{ASG}(\text{ENO}, \text{PNO}, \text{TITLE}, \text{DUR})$ 
 $\text{EMP2}(\text{ENO}, \text{ENAME}, \text{TITLE}, \text{CITY}) : -\text{EMP}(\text{ENO}, \text{ENAME}, \text{CITY}), \quad (r_2)$ 
 $\text{ASG}(\text{ENO}, \text{PNO}, \text{TITLE}, \text{DUR})$ 
 $\text{ASG1}(\text{ENO}, \text{PNO}, \text{DUR}) : -\text{ASG}(\text{ENO}, \text{PNO}, \text{TITLE}, \text{DUR}) \quad (r_3)$ 

```

◆

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 [Halevy, 2001]. 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 [Levy et al., 1995]. 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 [Levy et al., 1996b], the inverse rule algorithm [Duschka and Genesereth, 1997], and the MinCon algorithm [Pottinger and Levy, 2000]. 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 9.3 and the views in Example 9.4. The following mapping unifies the subgoal $\text{EMP}(e, \text{ENAME}, \text{"Paris"})$ of Q with the subgoal $\text{EMP}(\text{ENO}, \text{ENAME}, \text{CITY})$ in view EMP1 :

$$e \rightarrow \text{ENO, "Paris"} \rightarrow \text{CITY}$$

In the second step, for each view V of the Cartesian product of the non-empty 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 9.5. Let us consider query Q in Example 9.3 and the views in Example 9.4. 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 $\text{EMP}(e, \text{ENAME}, \text{"Paris"})$ and by b_2 the bucket for the subgoal $\text{ASG}(e, p, \text{TITLE}, \text{DUR})$. 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:

$$\begin{aligned} b_1 &= \{\text{EMP1}(\text{ENO}, \text{ENAME}, \text{TITLE}', \text{CITY}), \\ &\quad \text{EMP2}(\text{ENO}, \text{ENAME}, \text{TITLE}', \text{CITY})\} \\ b_2 &= \{\text{ASG1}(\text{ENO}, \text{PNO}, \text{DUR}')\} \end{aligned}$$

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

$$Q'(e, p) : -\text{EMP1}(e, \text{ENAME}, \text{TITLE}, \text{"Paris"}), \text{ASG1}(e, p, \text{DUR}) \quad (q_1)$$

$$Q'(e, p) : -\text{EMP2}(e, \text{ENAME}, \text{TITLE}, \text{"Paris"}), \text{ASG1}(e, p, \text{DUR}) \quad (q_2)$$

◆

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 9.6. Let us consider query Q in Example 9.3 and the views in Example 9.4 with the addition of the following view that gives the projects for which there are employees who live in Paris.

$$\begin{aligned} \text{PROJ1(PNO)} : -\text{EMP1(ENO, ENAME, "Paris")}, \\ \text{ASG(ENO, PNO, TITLE, DUR)} \end{aligned} \quad (r_4)$$

Now, the following mapping unifies the subgoal $\text{ASG}(e, p, \text{TITLE}, \text{DUR})$ of Q with the subgoal $\text{ASG}(\text{ENO}, \text{PNO}, \text{TITLE}, \text{DUR})$ in view PROJ1:

$$p \rightarrow \text{PNAME}$$

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 similar to that of the bucket algorithm, 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 9.6, 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 9.5. 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.

9.4 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 these three problems. We note that the result is a distributed execution plan to be executed by the wrappers and the mediator.

9.4.1 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 tree that correspond to the parts of the query executed at component DBMSs. If we assume that all local processing is “pushed down” in the tree, then we can modify the query plan such that the leaves of the tree 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 tree may be calculated recursively, based on the leaf node costs.

Three alternative approaches exist for determining the cost of executing queries at component DBMSs [Zhu and Larson, 1998]:

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 [Du et al., 1992; Zhu and Larson, 1994].
2. **Customized Approach.** This approach uses previous knowledge about the component DBMSs, as well as their external characteristics, to subjectively determine the cost information [Zhu and Larson, 1996a; Roth et al., 1999; Naacke et al., 1999].
3. **Dynamic Approach.** This approach monitors the run-time behavior of component DBMSs, and dynamically collects the cost information [Lu et al., 1992; Zhu et al., 2000, 2003; Rahal et al., 2004].

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

9.4.1.1 Black box approach

In the black box approach, which is used in the Pegasus project [Du et al., 1992], 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

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

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 formulae, which change with different component DBMSs. The approach taken in the Pegasus project 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 calibration database is synthetic, and the results obtained by using it may not apply well to real DBMSs [Zhu and Larson, 1994]. An alternative is proposed in the CORDS project [Zhu and Larson, 1996b], that 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 [Zhu and Larson, 1998]. 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 tree 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 is similar to the Pegasus cost function in that it 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 [Zhu and Larson, 1996a]. 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. This approach has been validated over various DBMS and has been shown to yield good results [Zhu and Larson, 2000].

The above approaches require a preliminary step to instantiate the cost model (either by calibration or sampling). This may not be appropriate in MDBMSs because it would slow down the system each time a new DBMS component is added. One way to address this problem, as proposed in the Hermes project, is to progressively learn the cost model from queries [Adali et al., 1996b]. The cost model designed in the Hermes mediator assumes that the underlying component DBMSs are invoked 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 components DBMSs. Then it monitors on-going 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 of 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 formulae 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.

9.4.1.2 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:

$$\text{Total_access_cost} = \text{reset_cost} + (\text{cardinality} - 1) * \text{advance_cost}$$

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 [Roth et al., 1999]. 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.

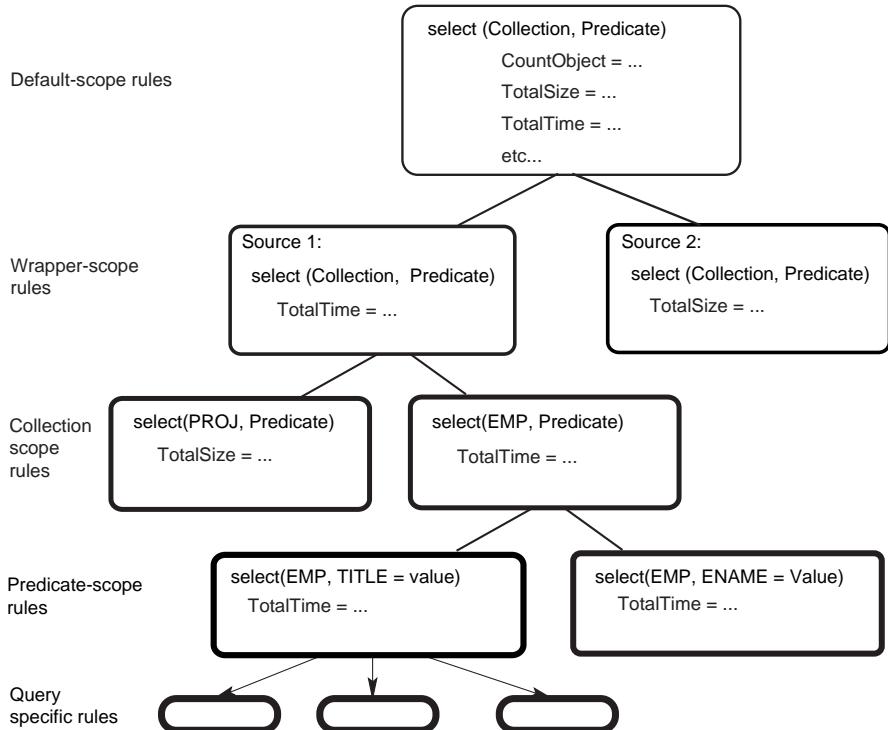
A second solution is to use a hierarchical generic cost model. As shown in Figure 9.2, each node represents a cost rule that associates a query pattern with a cost function for various cost parameters.

The node hierarchy is divided into five levels depending on the genericity of the cost rules (in Figure 9.2, 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 [Naacke et al., 1999].

Example 9.7. Consider the following relations:

EMP(ENO, ENAME, TITLE)
ASG(ENO, PNO, RESP, DUR)

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. 9.2** Hierarchical Cost Formula Tree

of attribute values. Half of the ASG tuples have a duration greater than 6. We detail below some parts of the mediator generic cost model (we use superscripts to indicate the access method):

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

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

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

$$\text{cost}(R \bowtie_A^{\text{nl}} S) = \text{cost}(R) + |R| * \text{cost}(S) \text{ (using a nested-loop (\textit{nl}) join)}$$

Consider the following global query Q :

```

SELECT *
FROM   EMP, ASG
WHERE  EMP.ENO=ASG.ENO
AND    ASG.DUR>6
  
```

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

$$P_1 = \sigma_{DUR>6}(\text{EMP} \bowtie_{\text{ENO}}^{ind} \text{ASG})$$

$$P_2 = \text{EMP} \bowtie_{\text{ENO}}^{nl} \sigma_{DUR>6}(\text{ASG})$$

$$P_3 = \sigma_{DUR>6}(\text{ASG}) \bowtie_{\text{ENO}}^{ind} \text{EMP}$$

$$P_4 = \sigma_{DUR>6}(\text{ASG}) \bowtie_{\text{ENO}}^{nl} \text{EMP}$$

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

$$\begin{aligned} \text{cost}(P_1) &= \text{cost}(\sigma_{DUR>6}(\text{EMP} \bowtie_{\text{ENO}}^{ind} \text{ASG})) \\ &= \text{cost}(\text{EMP} \bowtie_{\text{ENO}}^{ind} \text{ASG}) \\ &= \text{cost}(\text{EMP}) + |\text{EMP}| * \text{cost}(\sigma_{\text{ENO}=v}(\text{ASG})) \\ &= |\text{EMP}| + |\text{EMP}| * |\text{ASG}| = 10,001,000 \\ \text{cost}(P_2) &= \text{cost}(\text{EMP}) + |\text{EMP}| * \text{cost}(\sigma_{DUR>6}(\text{ASG})) \\ &= \text{cost}(\text{EMP}) + |\text{EMP}| * \text{cost}(\text{ASG}) \\ &= |\text{EMP}| + |\text{EMP}| * |\text{ASG}| = 10,001,000 \\ \text{cost}(P_3) &= \text{cost}(P_4) = |\text{ASG}| + \frac{|\text{ASG}|}{2} * |\text{EMP}| \\ &= 5,010,000 \end{aligned}$$

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:

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

db_2 exports the specific cost of selecting ASG tuples that have a given ENO as:

$$\text{cost}(\sigma_{\text{ENO}=v}(\text{ASG})) = |\sigma_{\text{ENO}=v}(\text{ASG})|$$

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

$$\begin{aligned} \text{cost}(P_1) &= |\text{EMP}| + |\text{EMP}| * |\sigma_{\text{ENO}=v}(\text{ASG})| \\ &= 1,000 + 1,000 * 10 \\ &= 11,000 \end{aligned}$$

$$\text{cost}(P_2) = |\text{EMP}| + |\text{EMP}| * |\sigma_{DUR>6}(\text{ASG})|$$

$$\begin{aligned}
 &= |\text{EMP}| + |\text{EMP}| * \frac{|\text{ASG}|}{2} \\
 &= 5,001,000 \\
 \text{cost}(P_3) &= |\text{ASG}| + \frac{|\text{ASG}|}{2} * |\sigma_{\text{ENO}=\nu}(\text{EMP})| \\
 &= 10,000 + 5,000 * 1 \\
 &= 15,000 \\
 \text{cost}(P_4) &= |\text{ASG}| + \frac{|\text{ASG}|}{2} * |\text{EMP}| \\
 &= 10,000 + 5,000 * 1,000 \\
 &= 5,010,000
 \end{aligned}$$

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 tradeoff between the overhead of providing specific cost information for diverse component DBMSs and the benefit of faster heterogeneous query processing.

9.4.1.3 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 [Rahal et al., 2004]. 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 change over time is to extend the sampling method [Zhu, 1995] and consider user queries as new samples. Query response time is measured to adjust the cost model parameters at run time 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 [Zhu

et al., 2000], 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 multi-category 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 [Zhu et al., 2003]. 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 MDBMS query processor and the component DBMS [Vidal et al., 1998]. This allows adjusting the query cost depending on the actual date.

9.4.2 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) [Melton et al., 2001]. 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.

9.4.2.1 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 Chapter 8) with a heterogeneous cost model (see Section 9.4.1). 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 Section 8.4.4): 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 where the right subtree of a join node is always a leaf node corresponding to a base relation (Figure 9.3a). 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 tree, and then convert it to a bushy tree [Du et al., 1995]. 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 tree, transforming it, step-by-step, to a bushy one has been proposed [Du et al., 1995]. The algorithm maintains two pointers, called *upper anchor nodes* (UAN) on the tree. 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 Figure 9.3a), 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 tree balanced, which reduces the response time.

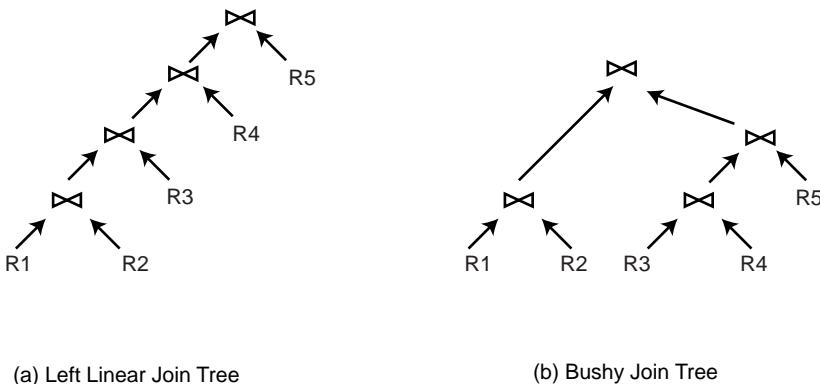


Fig. 9.3 Left Linear versus Bushy Join Tree

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 node 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 tree 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 tree that is generated by a commercial DBMS optimizer. While this is a good starting point, it can be argued that the original 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 [Evrendilek et al., 1997] can take

these into consideration. Starting from an initial join graph, the algorithm checks for different parenthesizations of this linear join execution order and produces a parenthesized order, which is optimal with respect to response time. The result is an (almost) balanced join execution tree. Performance evaluations indicate that this approach produces better quality plans at the expense of longer optimization time.

9.4.2.2 Operator-based Approach

Expressing the capabilities of the component DBMSs through relational operators allows tight integration of query processing between mediator and wrappers. In particular, the mediator/wrapper communication can be in terms of subplans. We illustrate the operator-based approach with planning functions proposed in the Garlic project [Haas et al., 1997a]. 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 the rule-based optimizer proposed by Lohman [1988] 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 9.8. We consider three component databases, each at a different site. Component database db_1 stores relation $EMP(ENO, ENAME, CITY)$. Component database db_2 stores relation $ASG(ENO, PNAME, DUR)$. 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( $R, A, P$ )

```

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):

$$EMPASG = (db_1.EMP \bowtie db_2.ASG) \cup db_3.EMPASG$$

After query rewriting in GAV and query optimization, the operator-based approach could produce the QEP shown in Figure 9.4. 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 multi-DBMSs. First, planning functions provide a flexible way to express precisely the capabilities of component data sources. In particular, they can be used to model non-relational data sources such as web sites. Second, since these rules are declarative, 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 9.8. Finally, this approach can be easily incorporated in an existing, centralized query optimizer.

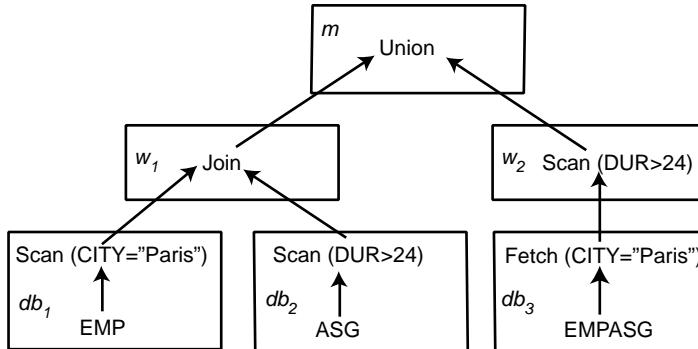


Fig. 9.4 Heterogeneous Query Execution Plan

The operator-based approach has also been successfully used in DISCO, a multi-DBMS designed to access multiple databases over the web [Tomasic et al., 1996, 1997, 1998]. 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 data sets, 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 [Kapitskaia et al., 1997].

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 Section 9.4.1.

9.4.3 Adaptive Query Processing

Multidatabase query processing, as discussed so far, follows essentially the principles of traditional query processing whereby an optimal QEP is produced for a query based on a cost model, which is then executed. The underlying assumption is that the multidatabase query optimizer has sufficient knowledge about query runtime conditions in order to produce an efficient QEP and the runtime conditions remain stable during execution. This is a fair assumption for multidatabase queries with few data sources running in a controlled environment. However, this assumption is inappropriate for changing environments with large numbers of data sources and unpredictable runtime conditions.

Example 9.9. Consider the QEP in Figure 9.5 with relations EMP, ASG, PROJ and PAY at sites s_1, s_2, s_3, s_4 , respectively. The crossed arrow indicates that, for some reason (e.g., failure), site s_2 (where ASG is stored) is not available at the beginning of execution. Let us assume, for simplicity, that the query is to be executed according to the iterator execution model [Graefe and McKenna, 1993], such that tuples flow from the left most relation,

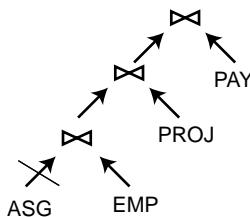


Fig. 9.5 Query Execution Plan with Blocked Data Source

Because of the unavailability of s_2 , the entire pipeline is blocked, waiting for ASG tuples to be produced. However, with some reorganization of the plan, some other operators could be evaluated while waiting for s_2 , for instance, to evaluate the join of EMP and PAY. ♦

This simple example illustrates that a typical static plan cannot cope with unpredictable data source unavailability [Amsaleg et al., 1996a]. More complex examples involve continuous queries [Madden et al., 2002b], expensive predicates [Porto et al., 2003] and data skew [Shah et al., 2003]. The main solution is to have some adaptive behavior during query processing, i.e., *adaptive query processing*. Adaptive query processing is a form of dynamic query processing, with a feedback loop between the execution environment and the query optimizer in order to react to unforeseen variations of runtime conditions. A query processing system is defined as adaptive if it receives information from the execution environment and determines its behavior according to that information in an iterative manner [Hellerstein et al., 2000; Goumaris et al., 2002b]. In the context of multidatabase systems, the execution environment

includes the mediator, wrappers and component DBMSs. In particular, wrappers should be able to collect information regarding execution within the component DBMSs. Obviously, this is harder to do with legacy DBMSs.

In this section, we first provide a general presentation of the adaptive query processing process. Then, we present, in more detail, the Eddy approach [Avnur and Hellerstein, 2000] that provides a powerful framework for adaptive query processing techniques. Finally, we discuss major extensions to Eddy.

9.4.3.1 Adaptive Query Processing Process

Adaptive query processing adds to the traditional query processing process the following activities: monitoring, assessing and reacting. These activities are logically implemented in the query processing system by sensors, assessment components, and reaction components, respectively. These components may be embedded into control operators of the QEP, e.g., the *Exchange* operator [Graefe and McKenna, 1993]. Monitoring involves measuring some environment parameters within a time window, and reporting them to the assessment component. The latter analyzes the reports and considers thresholds to arrive at an adaptive reaction plan. Finally, the reaction plan is communicated to the reaction component that applies the reactions to query execution.

Typically, an adaptive process specifies the frequency with which each component will be executed. There is a tradeoff between reactivity, in which higher values lead to eager reactions, and the overhead caused by the adaptive process. A generic representation of the adaptive process is given by the function $f_{adapt}(E, T) \rightarrow Ad$, where E is a set of monitored environment parameters, T is a set of threshold values and Ad is a possibly empty set of adaptive reactions. The elements of E, T and Ad , called adaptive elements, obviously may vary in a number of ways depending on the application. The most important elements are the monitoring parameters and the adaptive reactions. We now describe them, following the presentation in [Gounaris et al., 2002b].

Monitoring parameters.

Monitoring query runtime parameters involves placing sensors at key places of the QEP and defining observation windows, during which sensors collect information. It also requires the specification of a communication mechanism to pass collected information to the assessment component. Examples of candidates for monitoring are:

- Memory size. Monitoring available memory size allows, for instance, operators to react to memory shortage or memory increase [Shah et al., 2003].
- Data arrival rates. Monitoring the variations in data arrival rates may enable the query processor to do useful work while waiting for a blocked data source.

- Actual statistics. Database statistics in a multidatabase environment tend to be inaccurate, if at all available. Monitoring the actual size of relations and intermediate results may lead to important modifications in the QEP. Furthermore, the usual data assumptions, in which the selectivity of predicates over attributes in a relation are considered to be mutually independent, can be abandoned and real selectivity values can be computed.
- Operator execution cost. Monitoring the actual cost of operator execution, including production rates, is useful for better operator scheduling. Furthermore, monitoring the size of the queues placed before operators may avoid overload situations [Tian and DeWitt, 2003b].
- Network throughput. In multidatabase query evaluation with remote data sources, monitoring network throughput may be helpful to define the data retrieval block size. In a lower throughput network, the system may react with larger block sizes to reduce network penalty.

Adaptive reactions.

Adaptive reactions modify query execution behavior according to the decisions taken by the assessment component. Important adaptive reactions are the following:

- Change schedule: modifies the order in which operators in the QEP get scheduled. *Query Scrambling* [Amsaleg et al., 1996a; Urhan et al., 1998a] reacts by a *change schedule* of the plan, e.g., to reorganize the QEP in Example 9.9, to avoid stalling on a blocked data source during query evaluation. Eddy adopts finer reaction where operator scheduling can be decided on a tuple basis.
- Operator replacement: replaces a physical operator by an equivalent one. For example, depending on the available memory, the system may choose between a nested loop join or a hash join. Operator replacement may also change the plan by introducing a new operator to join the intermediate results produced by a previous adaptive reaction. Query Scrambling, for instance, may introduce new operators to evaluate joins between the results of *change schedule* reactions.
- Operator behavior: modifies the physical behavior of an operator. For example, the symmetric hash join [Wilschut and Apers, 1991] or ripple join algorithms [Haas and Hellerstein, 1999b] constantly alternate the inner/outer relation roles between their input tuples.
- Data repartitioning: considers the dynamic repartitioning of a relation through multiple nodes using intra-operator parallelism [Shah et al., 2003]. Static partitioning of a relation tends to produce load imbalance between nodes. For example, information partitioned according to their associated geographical region (i.e., continent) may exhibit different access rates during the day because of the time differences in users' locations.

- Plan reformulation: computes a new QEP to replace an inefficient one. The optimizer considers actual statistics and state information, collected on the fly, to produce a new plan.

9.4.3.2 Eddy Approach

Eddy is a general framework for adaptive query processing. It was developed in the context of the Telegraph project with the goal of running queries on large volumes of online data with unpredictable input rates and fluctuations in the running environment.

For simplicity, we only consider select-project-join (SPJ) queries. Select operators can include expensive predicates [Hellerstein and Stonebraker, 1993]. The process of generating a QEP from an input SPJ query begins by producing a spanning tree of the query graph G modeling the input query. The choice among join algorithms and relation access methods favors adaptiveness. A QEP can be modeled as a tuple $Q = \langle D, P, C \rangle$, where D is a set of data sources, P is a set of query predicates with associated algorithms, and C is a set of ordering constraints that must be followed during execution. Observe that multiple valid spanning trees can be derived from G that obey the constraints in C , by exploring the search space composed of equivalent plans with different predicate orders. There is no need to find an optimal QEP during query compilation. Instead, operator ordering is done on the fly on a tuple-per-tuple basis (i.e., tuple routing). The process of QEP compilation is completed by adding the *Eddy operator* which is an n -ary physical operator placed between data sources in D and query predicates in P .

Example 9.10. Consider a three-relation query $Q = \sigma_p(R) \bowtie S \bowtie T$, where joins are equi-joins. Assume that the only access method to relation T is through an index on join attribute $T.A$, i.e., the second join can only be an index join over $T.A$. Assume also that σ_p is an expensive predicate (e.g., a predicate over the results of running a program over values of $R.B$). Under these assumptions, the QEP is defined as $D = \{R, S, T\}$, $P = \{\sigma_p(R), R \bowtie_1 S, S \bowtie_2 T\}$ and $C = \{S \prec T\}$. The constraint \prec imposes S tuples to probe T tuples, based on the index on $T.A$.

Figure 9.6 shows a QEP produced by the compilation of query Q with Eddy. An ellipse corresponds to a physical operator (i.e., either the Eddy operator or an algorithm implementing a predicate $p \in P$). As usual, the bottom of the plan presents the data sources. In the absence of a scan access method, relation T access is wrapped by the index join implementing the second join, and, thus, does not appear as a data source. The arrows specify pipeline dataflow following a producer-consumer relationship. Finally, an arrow departing from the Eddy models the production of output tuples. ♦

Eddy provides fine-grain adaptiveness by deciding on the fly how to route tuples through predicates according to a scheduling policy. During query execution, tuples in data sources are retrieved and staged into an input buffer managed by the Eddy operator. Eddy responds to data source unavailability by simply reading from another data source and staging tuples in the buffer pool.

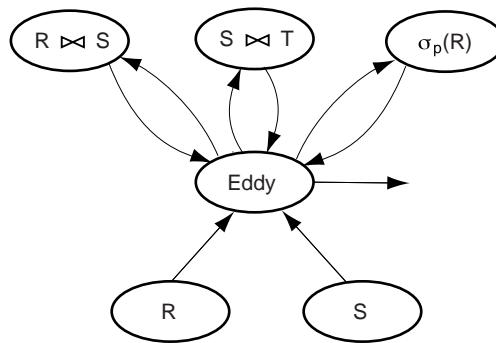


Fig. 9.6 A Query Execution Plan with Eddy.

The flexibility of choosing the currently available data source is obtained by relaxing the fixed order of predicates in a QEP. In Eddy, there is no fixed QEP and each tuple follows its own path through predicates according to the constraints in the plan and its own history of predicate evaluation.

The tuple-based routing strategy produces a new QEP topology. The Eddy operator together with its managed predicates form a circular dataflow in which tuples leave the Eddy operator to be evaluated by the predicates, which in turn bounce back output tuples to the Eddy operator. A tuple leaves the circular dataflow either when it is eliminated by a predicate evaluation or the Eddy operator realizes that the tuple has passed through all the predicates in its list. The lack of a fixed QEP requires each tuple to register the set of predicates it is eligible for. For example, in Figure 9.6, S tuples are eligible for the two join predicates but are not eligible for predicate $\sigma_p(R)$.

Let us now present, in more detail, how Eddy adaptively performs join ordering and scheduling.

Adaptive join ordering.

A fixed QEP (produced at compile time) dictates the join ordering and specifies which relations can be pipelined through the join operators. This makes query execution simple. When, as in Eddy, there is no fixed QEP, the challenge is to dynamically order pipelined join operators at run time, 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 (chosen by the scheduling policy) 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 the basic hash-based join algorithm, for instance: the inner relation is fully read during

the build phase to construct a hash table, whereas tuples in the outer relation are 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 when the entire hash table has been 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 few opportunities 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 where the role played by the relations in a join may change without producing incorrect results.

The earliest example of a symmetric join algorithm is the symmetric hash join [Wilschut and Apers, 1991], 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 [Haas and Hellerstein, 1999b], which can be viewed as 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.

Using symmetric join algorithms, Eddy can achieve flexible join ordering by controlling the history and constraints regarding predicate evaluation on a tuple basis. This control is implemented using two sets of *progress bits* carried by each tuple, which indicate, respectively, the predicates to which the tuple is ready to be evaluated by (i.e., the “ready bits”) and the set of predicates already evaluated (i.e., the “done bits”). When a tuple t is read into an Eddy operator, all done bits are zeroed and the predicates without ordering constraints, and to which t is eligible for, have their corresponding ready bits set. After each predicate evaluation, the corresponding done bit is set and the ready bits are updated, accordingly. When a join concatenates a pair of tuples, their done bits are ORed and a new set of ready bits are turned on. Combining progress bits and symmetric join algorithms allows Eddy to schedule predicates in an adaptive way.

Adaptive scheduling.

Given a set of candidate predicates, Eddy must adaptively select the one to which each tuple will be sent. Two main principles drive the choice of a predicate in Eddy: cost and selectivity. Predicate costs are measured as a function of the consumption

rate of each predicate. Remember that the Eddy operator holds tuples in its internal buffer, which is shared by all predicates. Low cost (i.e., fast) predicates finish their work quicker and request new tuples from the Eddy. As a result, low cost predicates get allocated more tuples than high cost predicates. This strategy, however, is agnostic with respect to predicate selectivity. Eddy's tuple routing strategy is complemented by a simple *lottery scheduling* mechanism that learns about predicate selectivity [Arpacı-Dusseau et al., 1999]. The strategy credits a ticket to a predicate whenever the latter gets scheduled a tuple. Once a tuple has been processed and is bounced back to the Eddy, the corresponding predicate gets its ticket amount decremented. Combining cost and selectivity criteria becomes easy. Eddy continuously runs a lottery among predicates currently requesting tuples. The predicate with higher count of tickets wins the lottery and gets scheduled.

Another interesting issue is the choice of the running tuple from the input buffer. In order to end query processing, all tuples in the input buffer must be evaluated. Thus, a difference in tuple scheduling may reflect user preferences with respect to tuple output. For example, Eddy may favor tuples with higher number of done bits set, so that the user receives first results earlier.

9.4.3.3 Extensions to Eddy

The Eddy approach has been extended in various directions. In the cherry picking approach [Porto et al., 2003], context is used instead of simple ticket-based scheduling. The relationship among expensive predicate input attribute values are discovered at runtime and used as the basis for adaptive tuple scheduling. Given a query Q with $D = \{R[A, B, C]\}$, $P = \{\sigma_p^1(R.A), \sigma_p^2(R.B), \sigma_p^3(R.C)\}$ and $C = \emptyset$, the main idea is to model the input attribute values of the expensive predicates in P as a hypergraph $G = (V, E)$, where V is a set of n node partitions, with n being the number of expensive predicates. Each partition corresponds to a single attribute of the input relation R that are input to a predicate in P and each node corresponds to a distinct value of that attribute. An hyperedge $e = \{a_i, b_j, c_k\}$ corresponds to a tuple of relation R . The degree of a node v_i corresponds to the number of hyperedges in which v_i takes part. With this modeling, efficiently evaluating query Q corresponds to eliminating as quickly as possible the hyperedges in G . An hyperedge is eliminated whenever a value associated with one of its nodes is evaluated by a predicate in P and returns false. Furthermore, node degrees model hidden attribute dependencies, so that when the result of a predicate evaluation over a value v_i returns false, all hyperedges (i.e., tuples) that v_i takes part in are also eliminated. An adaptive content-sensitive strategy to evaluate a query Q is proposed for this model. It schedules values to be evaluated by a predicate according to the *Fanout* of its corresponding node, computed as the product of the node degree in the hypergraph G with the ratio between the corresponding predicate selectivity and predicate unitary evaluation cost.

Another interesting extension is distributed Eddies [Tian and DeWitt, 2003b] to deal with distributed input data streams. Since a centralized Eddy operator may quickly become a bottleneck, a distributed approach is proposed for tuple routing.

Each operator decides on the next operator to route a tuple to based on its history of operator's evaluation (i.e., done bits) and statistics collected from the remaining operators. In a distributed setting, each operator may run at a different node in the network with a queue holding input tuples. The query optimization problem is specified by considering two new metrics for measuring stream query performance: average response time and maximum data rate. The former corresponds to the average time tuples take to traverse the operators in a plan, whereas the latter measures the maximum throughput the system can withstand without overloading. Routing strategies use the following parameters: operator's cost, selectivity, length of operator's input queue and probability of an operator being routed a tuple. The combination of these parameters yields efficient query evaluation. Using operator's cost and selectivity guarantee that low-cost and highly selective operators are given higher routing priority. Queue length provides information on the average time tuples are staged in queues. Managing operator's queue length allows the routing decision to avoid overloaded operators. Thus, by supporting routing policies, each operator is able to individually make routing decisions, thereby avoiding the bottleneck of a centralized router.

9.5 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 [Roth and Schwartz, 1997].

The main function of a wrapper is conversion between the common interface and the DBMS-dependent interface. Figure 9.7 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 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 Chapter 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 Figure 9.4).

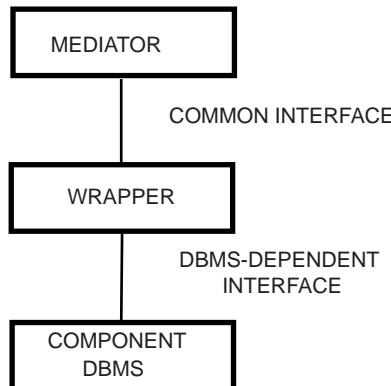


Fig. 9.7 Wrapper interfaces

As discussed in Section 9.4.2, 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, borrowed from [Melton et al., 2001], illustrates how a very simple data source can be wrapped to be accessed through SQL.

Example 9.11. 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 the wrapper supporting access to this relation SQL statements. 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 CASE tools with reusable components to generate most of the wrapper code [Tomasic et al., 1997]. 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). The main problem is due to the heterogeneity of integrity constraints between the common interface and the DBMS-dependent interface. As discussed in Chapter 5, integrity constraints are used to reject updates that violate database consistency. In modern DBMSs, integrity constraints are explicit and specified as rules as 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 9.11, 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 to guarantee 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 CASE tool with reverse engineering techniques to identify within application code the implicit integrity constraints which are then translated into validation code in the wrappers [Thiran et al., 2006].

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 9.11, 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 presented in Chapter 4.

9.6 Conclusion

Query processing in multidatabase systems 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 Chapter 6. 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 run time 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 even semi-structured (based on XML). 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 semi-structured (e.g., XML-based) may be preferred. This requires significant extensions to query processing techniques.

9.7 Bibliographic Notes

Work on multidatabase query processing started in the early 1980's 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 1990's, 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 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 customized approach is developed in [Zhu and Larson, 1996a; Roth et al., 1999; Naacke et al., 1999]. The dynamic approach is used in [Zhu et al., 2000], [Zhu et al., 2003] and [Rahal et al., 2004].

The algorithm we described to illustrate the query-based approach to heterogeneous query optimization has been proposed in [Du et al., 1995]. 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].

Adaptive query processing is surveyed in [Hellerstein et al., 2000; Gounaris et al., 2002b]. The seminal paper on the Eddy approach which we used to illustrate adaptive query processing is [Avnur and Hellerstein, 2000]. Other important techniques for adaptive query processing are query scrambling [Amsaleg et al., 1996a; Urhan et al., 1998a], 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, 2003b].

A software engineering solution to the problem of wrapper creation and maintenance, considering integrity control, is proposed in [Thiran et al., 2006].

Exercises

Problem 9.1 ().** 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 9.2 (*). Consider a marketing application with a ROLAP server at site s_1 which needs to integrate information from two customer databases, each at site s_2 within the corporate network. Assume also that the application needs to combine customer information with information extracted from Web data sources about cities in 10 different countries. For security reasons, a web server at site s_3 is dedicated to Web access outside the corporate network. Propose a multidatabase system architecture with mediator and wrappers to support this application. Discuss and justify design choices.

Problem 9.3 ().** Consider the global relations $\text{EMP}(\text{ENAME}, \text{TITLE}, \text{CITY})$ and $\text{ASG}(\text{ENAME}, \text{PNAME}, \text{CITY}, \text{DUR})$. City in ASG is the location of the project of name PNAME (i.e., PNAME functionnally determines CITY). Consider the local relations $\text{EMP1}(\text{ENAME}, \text{TITLE}, \text{CITY})$, $\text{EMP2}(\text{ENAME}, \text{TITLE}, \text{CITY})$, $\text{PROJ1}(\text{PNAME}, \text{CITY})$, $\text{PROJ2}(\text{PNAME}, \text{CITY})$ and $\text{ASG1}(\text{ENAME}, \text{PNAME}, \text{DUR})$. 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 9.4 (*). Consider relations EMP and ASG of Example 9.7. We denote by $|R|$ the number of pages to store R on disk. Consider the following statistics about the data:

$$\begin{aligned} |\text{EMP}| &= 1\,000 \\ |\text{EMP}| &= 100 \\ |\text{ASG}| &= 10\,000 \\ |\text{ASG}| &= 2\,000 \\ \text{selectivity}(\text{ASG.DUR} > 36) &= 1\% \end{aligned}$$

The mediator generic cost model is:

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

$$\text{cost}(\sigma(X)) = \text{cost}(X) \text{ where } X \text{ contains at least one operator.}$$

$$\text{cost}(R \bowtie_A^{ind} S) = \text{cost}(R) + |R| * \text{cost}(\sigma_{A=v}(S)) \text{ using an indexed join algorithm.}$$

$$\text{cost}(R \bowtie_A^{nl} S) = \text{cost}(R) + |R| * \text{cost}(S) \text{ using a nested loop join algorithm.}$$

Consider the MDBMS input query Q :

```

SELECT *
FROM   EMP, ASG
WHERE  EMP.ENO=ASG.ENO
AND    ASG.DUR>36

```

Consider four plans to process Q :

$$\begin{aligned}
P_1 &= \text{EMP} \bowtie_{\text{ENO}}^{\text{ind}} \sigma_{\text{DUR}>36}(\text{ASG}) \\
P_2 &= \text{EMP} \bowtie_{\text{ENO}}^{\text{nl}} \sigma_{\text{DUR}>36}(\text{ASG}) \\
P_3 &= \sigma_{\text{DUR}>36}(\text{ASG}) \bowtie_{\text{ENO}}^{\text{ind}} \text{EMP} \\
P_4 &= \sigma_{\text{DUR}>36}(\text{ASG}) \bowtie_{\text{ENO}}^{\text{nl}} \text{EMP}
\end{aligned}$$

- (a) What is the cost of plans P_1 to P_4 ?
- (b) Which plan has the minimal cost?

Problem 9.5 (*). 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:

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

The specific cost of selecting ASG tuples that have a given ENO at D_2 is:

$$\text{cost}(\sigma_{\text{ENO}=v}(\text{ASG})) = |\sigma_{\text{ENO}=v}(\text{ASG})|$$

- (a) What is the cost of plans P_1 to P_4 ?
- (b) Which plan has the minimal cost?

Problem 9.6 ().** 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 9.7 ().** Consider Example 9.8 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 9.8.

Problem 9.8 ().** Consider three relations $R(A, B)$, $S(B, C)$ and $T(C, D)$ and query $Q(\sigma_p^1(R) \bowtie_1 S \bowtie_2 T)$, where \bowtie_1 and \bowtie_2 are natural joins. Assume that S has an index

on attribute B and T has an index on attribute C . Furthermore, σ_p^1 is an expensive predicate (i.e., a predicate over the results of running a program over values of $R.A$). Using the Eddy approach for adaptive query processing, answer the following questions:

- (a) Propose the set C of constraints on Q to produce an Eddy-based QEP.
- (b) Give a query graph G for Q .
- (c) Using C and G , propose an Eddy-based QEP.
- (d) Propose a second QEP that uses State Modules. Discuss the advantages obtained by using state modules in this QEP.

Problem 9.9 ().** Propose a data structure to store tuples in the Eddy buffer pool to help choosing quickly the next tuple to be evaluated according to user specified preference, for instance, produce first results earlier.

Problem 9.10 ().** Propose a predicate scheduling algorithm based on the Cherry picking approach introduced in Section 9.4.3.3.

Chapter 10

Introduction to Transaction Management

Up to this point the basic access primitive that we have considered has been a query. Our focus has been on retrieve-only (or read-only) queries that read data from a distributed database. We have not yet considered what happens if, for example, two queries attempt to update the same data item, or if a system failure occurs during execution of a query. For retrieve-only queries, neither of these conditions is a problem. One can have two queries reading the value of the same data item concurrently. Similarly, a read-only query can simply be restarted after a system failure is handled. On the other hand, it is not difficult to see that for update queries, these conditions can have disastrous effects on the database. We cannot, for example, simply restart the execution of an update query following a system failure since certain data item values may already have been updated prior to the failure and should not be updated again when the query is restarted. Otherwise, the database would contain incorrect data.

The fundamental point here is that there is no notion of “consistent execution” or “reliable computation” associated with the concept of a query. The concept of a *transaction* is used in database systems as a basic unit of consistent and reliable computing. Thus queries are executed as transactions once their execution strategies are determined and they are translated into primitive database operations.

In the discussion above, we used the terms *consistent* and *reliable* quite informally. Due to their importance in our discussion, we need to define them more precisely. We differentiate between *database consistency* and *transaction consistency*.

A database is in a *consistent state* if it obeys all of the consistency (integrity) constraints defined over it (see Chapter 5). State changes occur due to modifications, insertions, and deletions (together called *updates*). Of course, we want to ensure that the database never enters an inconsistent state. Note that the database can be (and usually is) temporarily inconsistent during the execution of a transaction. The important point is that the database should be consistent when the transaction terminates (Figure 10.1).

Transaction consistency, on the other hand, refers to the actions of concurrent transactions. We would like the database to remain in a consistent state even if there are a number of user requests that are concurrently accessing (reading or updating)

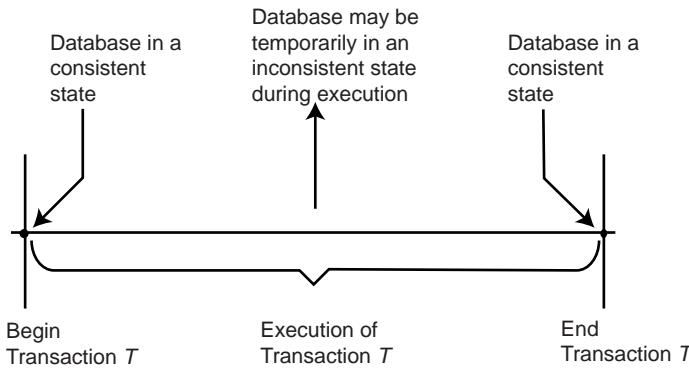


Fig. 10.1 A Transaction Model

the database. A complication arises when replicated databases are considered. A replicated database is in a *mutually consistent state* if all the copies of every data item in it have identical values. This is referred to as *one-copy equivalence* since all replica copies are forced to assume the same state at the end of a transaction's execution. There are more relaxed notions of replica consistency that allow replica values to diverge. These will be discussed later in Chapter 13.

Reliability refers to both the *resiliency* of a system to various types of failures and its capability to *recover* from them. A resilient system is tolerant of system failures and can continue to provide services even when failures occur. A recoverable DBMS is one that can get to a consistent state (by moving back to a previous consistent state or forward to a new consistent state) following various types of failures.

Transaction management deals with the problems of always keeping the database in a consistent state even when concurrent accesses and failures occur. In the upcoming two chapters, we investigate the issues related to managing transactions. A third chapter will address issues related to keeping replicated databases consistent. The purpose of the current chapter is to define the fundamental terms and to provide the framework within which these issues can be discussed. It also serves as a concise introduction to the problem and the related issues. We will therefore discuss the concepts at a high level of abstraction and will not present any management techniques.

The organization of this chapter is as follows. In the next section we formally and intuitively define the concept of a transaction. In Section 10.2 we discuss the properties of transactions and what the implications of each of these properties are in terms of transaction management. In Section 10.3 we present various types of transactions. In Section 10.4 we revisit the architectural model defined in Chapter 1 and indicate the modifications that are necessary to support transaction management.

10.1 Definition of a Transaction

Gray [1981] indicates that the transaction concept has its roots in contract law. He states, “In making a contract, two or more parties negotiate for a while and then make a deal. The deal is made binding by the joint signature of a document or by some other act (as simple as a handshake or a nod). If the parties are rather suspicious of one another or just want to be safe, they appoint an intermediary (usually called an escrow officer) to coordinate the commitment of the transaction.” The nice aspect of this historical perspective is that it does indeed encompass *some* of the fundamental properties of a transaction (atomicity and durability) as the term is used in database systems. It also serves to indicate the differences between a transaction and a query.

As indicated before, a transaction is a unit of consistent and reliable computation. Thus, intuitively, a transaction takes a database, performs an action on it, and generates a new version of the database, causing a state transition. This is similar to what a query does, except that if the database was consistent before the execution of the transaction, we can now guarantee that it will be consistent at the end of its execution regardless of the fact that (1) the transaction may have been executed concurrently with others, and (2) failures may have occurred during its execution.

In general, a transaction is considered to be made up of a sequence of read and write operations on the database, together with computation steps. In that sense, a transaction may be thought of as a program with embedded database access queries [Papadimitriou, 1986]. Another definition of a transaction is that it is a single execution of a program [Ullman, 1988]. A single query can also be thought of as a program that can be posed as a transaction.

Example 10.1. Consider the following SQL query for increasing by 10% the budget of the CAD/CAM project that we discussed (in Example 5.20):

```
UPDATE PROJ
SET      BUDGET = BUDGET*1.1
WHERE    PNAME= "CAD/CAM"
```

This query can be specified, using the embedded SQL notation, as a transaction by giving it a name (e.g., BUDGET_UPDATE) and declaring it as follows:

```
Begin_transaction BUDGET_UPDATE
begin
  EXEC SQL      UPDATE PROJ
  SET          BUDGET = BUDGET*1.1
  WHERE        PNAME= "CAD/CAM"
end.
```



The **Begin.transaction** and **end** statements delimit a transaction. Note that the use of delimiters is not enforced in every DBMS. If delimiters are not specified, a DBMS may simply treat as a transaction the entire program that performs a database access.

Example 10.2. In our discussion of transaction management concepts, we will use an airline reservation system example instead of the one used in the first nine chapters. The real-life implementation of this application almost always makes use of the transaction concept. Let us assume that there is a FLIGHT relation that records the data about each flight, a CUST relation for the customers who book flights, and an FC relation indicating which customers are on what flights. Let us also assume that the relation definitions are as follows (where the underlined attributes constitute the keys):

```
FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)
CUST(CNAME, ADDR, BAL)
FC(FNO, DATE, CNAME, SPECIAL)
```

The definition of the attributes in this database schema are as follows: FNO is the flight number, DATE denotes the flight date, SRC and DEST indicate the source and destination for the flight, STSOLD indicates the number of seats that have been sold on that flight, CAP denotes the passenger capacity on the flight, CNAME indicates the customer name whose address is stored in ADDR and whose account balance is in BAL, and SPECIAL corresponds to any special requests that the customer may have for a booking.

Let us consider a simplified version of a typical reservation application, where a travel agent enters the flight number, the date, and a customer name, and asks for a reservation. The transaction to perform this function can be implemented as follows, where database accesses are specified in embedded SQL notation:

```
Begin_transaction Reservation
begin
  input(flight_no, date, customer_name);                                (1)
  EXEC SQL UPDATE FLIGHT
    SET      STSOLD = STSOLD + 1
    WHERE   FNO = flight_no
    AND     DATE = date;
  EXEC SQL INSERT
    INTO    FC(FNO,DATE,CNAME,SPECIAL)                                     (3)
    VALUES (flight_no,date,customer_name, null);
  output("reservation completed")                                         (4)
end.
```

Let us explain this example. First a point about notation. Even though we use embedded SQL, we do not follow its syntax very strictly. The lowercase terms are the program variables; the uppercase terms denote database relations and attributes as well as the SQL statements. Numeric constants are used as they are, whereas character constants are enclosed in quotes. Keywords of the host language are written in boldface, and *null* is a keyword for the null string.

The first thing that the transaction does [line (1)], is to input the flight number, the date, and the customer name. Line (2) updates the number of sold seats on the requested flight by one. Line (3) inserts a tuple into the FC relation. Here we assume that the customer is an old one, so it is not necessary to have an insertion into the CUST relation, creating a record for the client. The keyword *null* in line (3) indicates that the customer has no special requests on this flight. Finally, line (4) reports the result of the transaction to the agent's terminal. ♦

10.1.1 Termination Conditions of Transactions

The reservation transaction of Example 10.2 has an implicit assumption about its termination. It assumes that there will always be a free seat and does not take into consideration the fact that the transaction may fail due to lack of seats. This is an unrealistic assumption that brings up the issue of termination possibilities of transactions.

A transaction always terminates, even when there are failures as we will see in Chapter 12. If the transaction can complete its task successfully, we say that the transaction *commits*. If, on the other hand, a transaction stops without completing its task, we say that it *aborts*. Transactions may abort for a number of reasons, which are discussed in the upcoming chapters. In our example, a transaction aborts itself because of a condition that would prevent it from completing its task successfully. Additionally, the DBMS may abort a transaction due to, for example, deadlocks or other conditions. When a transaction is aborted, its execution is stopped and all of its already executed actions are *undone* by returning the database to the state before their execution. This is also known as *rollback*.

The importance of commit is twofold. The commit command signals to the DBMS that the effects of that transaction should now be reflected in the database, thereby making it visible to other transactions that may access the same data items. Second, the point at which a transaction is committed is a “point of no return.” The results of the committed transaction are now *permanently* stored in the database and cannot be undone. The implementation of the commit command is discussed in Chapter 12.

Example 10.3. Let us return to our reservation system example. One thing we did not consider is that there may not be any free seats available on the desired flight. To cover this possibility, the reservation transaction needs to be revised as follows:

```
Begin_transaction Reservation
begin
    input(flight_no, date, customer_name);
    EXEC SQL SELECT STSOLD,CAP
        INTO    temp1,temp2
        FROM   FLIGHT
        WHERE  FNO = flight_no
        AND    DATE = date;
```

```

if temp1 = temp2 then
begin
    output("no free seats");
    Abort
end
else begin
    EXEC SQL UPDATE FLIGHT
        SET     STSOLD = STSOLD + 1
        WHERE   FNO = flight_no
        AND    DATE = date;
    EXEC SQL INSERT
        INTO    FC(FNO,DATE,CNAME,SPECIAL)
        VALUES (flight_no, date, customer_name, null);
    Commit;
    output("reservation completed")
end
end-if
end.

```

In this version the first SQL statement gets the STSOLD and CAP into the two variables temp1 and temp2. These two values are then compared to determine if any seats are available. The transaction either aborts if there are no free seats, or updates the STSOLD value and inserts a new tuple into the FC relation to represent the seat that was sold. ♦

Several things are important in this example. One is, obviously, the fact that if no free seats are available, the transaction is aborted¹. The second is the ordering of the output to the user with respect to the abort and commit commands. Transactions can be aborted either due to application logic, as is the case here, or due to deadlocks or system failures. If the transaction is aborted, the user can be notified before the DBMS is instructed to abort it. However, in case of commit, the user notification has to follow the successful servicing (by the DBMS) of the commit command, for reliability reasons. These are discussed further in Section 10.2.4 and in Chapter 12.

10.1.2 Characterization of Transactions

Observe in the preceding examples that transactions read and write some data. This has been used as the basis for characterizing a transaction. The data items that a transaction reads are said to constitute its *read set (RS)*. Similarly, the data items that a transaction writes are said to constitute its *write set (WS)*. The read set and write

¹ We will be kind to the airlines and assume that they never overbook. Thus our reservation transaction does not need to check for that condition.

set of a transaction need not be mutually exclusive. The union of the read set and write set of a transaction constitutes its *base set* ($BS = RS \cup WS$).

Example 10.4. Considering the reservation transaction as specified in Example 10.3 and the insert to be a number of write operations, the above-mentioned sets are defined as follows:

$$\begin{aligned} RS[\text{Reservation}] &= \{\text{FLIGHT.STSOLD}, \text{FLIGHT.CAP}\} \\ WS[\text{Reservation}] &= \{\text{FLIGHT.STSOLD}, \text{FC.FNO}, \text{FC.DATE}, \\ &\quad \text{FC.CNAME}, \text{FC.SPECIAL}\} \\ BS[\text{Reservation}] &= \{\text{FLIGHT.STSOLD}, \text{FLIGHT.CAP}, \\ &\quad \text{FC.FNO}, \text{FC.DATE}, \text{FC.CNAME}, \text{FC.SPECIAL}\} \end{aligned}$$

Note that it may be appropriate to include FLIGHT.FNO and FLIGHT.DATE in the read set of Reservation since they are accessed during execution of the SQL query. We omit them to simplify the example. ♦

We have characterized transactions only on the basis of their read and write operations, without considering the insertion and deletion operations. We therefore base our discussion of transaction management concepts on *static* databases that do not grow or shrink. This simplification is made in the interest of simplicity. Dynamic databases have to deal with the problem of *phantoms*, which can be explained using the following example. Consider that 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. We do not discuss phantoms any further in this book; the topic is discussed at length by Eswaran et al. [1976] and Bernstein et al. [1987].

We should also point out that the read and write operations to which we refer are abstract operations that do not have one-to-one correspondence to physical I/O primitives. One read in our characterization may translate into a number of primitive read operations to access the index structures and the physical data pages. The reader should treat each read and write as a language primitive rather than as an operating system primitive.

10.1.3 Formalization of the Transaction Concept

By now, the meaning of a transaction should be intuitively clear. To reason about transactions and about the correctness of the management algorithms, it is necessary to define the concept formally. We denote by $O_{ij}(x)$ some operation O_j of transaction T_i that operates on a database entity x . Following the conventions adopted in the

preceding section, $O_{ij} \in \{\text{read, write}\}$. Operations are assumed to be *atomic* (i.e., each is executed as an indivisible unit). We let OS_i denote the set of all operations in T_i (i.e., $OS_i = \bigcup_j O_{ij}$). We denote by N_i the termination condition for T_i , where $N_i \in \{\text{abort, commit}\}$ ².

With this terminology we can define a transaction T_i as a partial ordering over its operations and the termination condition. A partial order $P = \{\Sigma, \prec\}$ defines an ordering among the elements of Σ (called the *domain*) according to an irreflexive and transitive binary relation \prec defined over Σ . In our case Σ consists of the operations and termination condition of a transaction, whereas \prec indicates the execution order of these operations (which we will read as “precedes in execution order”). Formally, then, a transaction T_i is a partial order $T_i = \{\Sigma_i, \prec_i\}$, where

1. $\Sigma_i = OS_i \cup \{N_i\}$.
2. For any two operations $O_{ij}, O_{ik} \in OS_i$, if $O_{ij} = \{R(x)\text{ or }W(x)\}$ and $O_{ik} = W(x)$ for any data item x , then either $O_{ij} \prec_i O_{ik}$ or $O_{ik} \prec_i O_{ij}$.
3. $\forall O_{ij} \in OS_i, O_{ij} \prec_i N_i$.

The first condition formally defines the domain as the set of read and write operations that make up the transaction, plus the termination condition, which may be either commit or abort. The second condition specifies the ordering relation between the conflicting read and write operations of the transaction, while the final condition indicates that the termination condition always follows all other operations.

There are two important points about this definition. First, the ordering relation \prec is given and the definition does not attempt to construct it. The ordering relation is actually application dependent. Second, condition two indicates that the ordering between conflicting operations has to exist within \prec . Two operations, $O_i(x)$ and $O_j(x)$, are said to be in *conflict* if $O_i = \text{Write}$ or $O_j = \text{Write}$ (i.e., at least one of them is a Write and they access the same data item).

Example 10.5. Consider a simple transaction T that consists of the following steps:

```

Read(x)
Read(y)
x ← x + y
Write(x)
Commit

```

The specification of this transaction according to the formal notation that we have introduced is as follows:

$$\begin{aligned}\Sigma &= \{R(x), R(y), W(x), C\} \\ \prec &= \{(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)\}\end{aligned}$$

where (O_i, O_j) as an element of the \prec relation indicates that $O_i \prec O_j$. ♦

² From now on, we use the abbreviations *R*, *W*, *A* and *C* for the Read, Write, Abort, and Commit operations, respectively.

Notice that the ordering relation specifies the relative ordering of all operations with respect to the termination condition. This is due to the third condition of transaction definition. Also note that we do not specify the ordering between every pair of operations. That is why it is a *partial* order.

Example 10.6. The reservation transaction developed in Example 10.3 is more complex. Notice that there are two possible termination conditions, depending on the availability of seats. It might first seem that this is a contradiction of the definition of a transaction, which indicates that there can be only one termination condition. However, remember that a transaction is the execution of a program. It is clear that in any execution, only one of the two termination conditions can occur. Therefore, what exists is one transaction that aborts and another one that commits. Using this formal notation, the former can be specified as follows:

$$\begin{aligned}\Sigma &= \{R(\text{STSOLD}), R(\text{CAP}), A\} \\ \prec &= \{(O_1, A), (O_2, A)\}\end{aligned}$$

and the latter can be specified as

$$\begin{aligned}\Sigma &= \{R(\text{STSOLD}), R(\text{CAP}), W(\text{STSOLD}), \\ &\quad W(\text{FNO}), W(\text{DATE}), W(\text{CNAME}), W(\text{SPECIAL}), C\} \\ \prec &= \{(O_1, O_3), (O_2, O_3), (O_1, O_4), (O_1, O_5), (O_1, O_6), (O_1, O_7), (O_2, O_4), \\ &\quad (O_2, O_5), (O_2, O_6), (O_2, O_7), (O_1, C), (O_2, C), (O_3, C), (O_4, C), \\ &\quad (O_5, C), (O_6, C), (O_7, C)\}\end{aligned}$$

where $O_1 = R(\text{STSOLD})$, $O_2 = R(\text{CAP})$, $O_3 = W(\text{STSOLD})$, $O_4 = W(\text{FNO})$, $O_5 = W(\text{DATE})$, $O_6 = W(\text{CNAME})$, and $O_7 = W(\text{SPECIAL})$. \blacklozenge

One advantage of defining a transaction as a partial order is its correspondence to a directed acyclic graph (DAG). Thus a transaction can be specified as a DAG whose vertices are the operations of a transaction and whose arcs indicate the ordering relationship between a given pair of operations. This will be useful in discussing the concurrent execution of a number of transactions (Chapter 11) and in arguing about their correctness by means of graph-theoretic tools.

Example 10.7. The transaction discussed in Example 10.5 can be represented as a DAG as depicted in Figure 10.2. Note that we do not draw the arcs that are implied by transitivity even though we indicate them as elements of \prec . \blacklozenge

In most cases we do not need to refer to the domain of the partial order separately from the ordering relation. Therefore, it is common to drop Σ from the transaction definition and use the name of the partial order to refer to both the domain and the name of the partial order. This is convenient since it allows us to specify the ordering of the operations of a transaction in a more straightforward manner by making use of their relative ordering in the transaction definition. For example, we can define the transaction of Example 10.5 as follows:

$$T = \{R(x), R(y), W(x), C\}$$

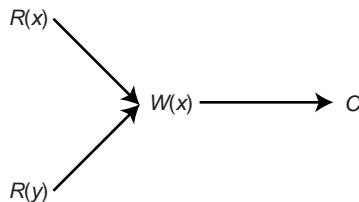


Fig. 10.2 DAG Representation of a Transaction

instead of the longer specification given before. We will therefore use the modified definition in this and subsequent chapters.

10.2 Properties of Transactions

The previous discussion clarifies the concept of a transaction. However, we have not yet provided any justification of our earlier claim that it is a unit of consistent and reliable computation. We do that in this section. The consistency and reliability aspects of transactions are due to four properties: (1) atomicity, (2) consistency, (3) isolation, and (4) durability. Together, these are commonly referred to as the ACID properties of transactions. They are not entirely independent of each other; usually there are dependencies among them as we will indicate below. We discuss each of these properties in the following sections.

10.2.1 Atomicity

Atomicity refers to the fact that a transaction is treated as a unit of operation. Therefore, either all the transaction's actions are completed, or none of them are. This is also known as the “all-or-nothing property.” Notice that we have just extended the concept of atomicity from individual operations to the entire transaction. Atomicity requires that if the execution of a transaction is interrupted by any sort of failure, the DBMS will be responsible for determining what to do with the transaction upon recovery from the failure. There are, of course, two possible courses of action: it can either be terminated by completing the remaining actions, or it can be terminated by undoing all the actions that have already been executed.

One can generally talk about two types of failures. A transaction itself may fail due to input data errors, deadlocks, or other factors. In these cases either the transaction aborts itself, as we have seen in Example 10.2, or the DBMS may abort it while handling deadlocks, for example. Maintaining transaction atomicity in the presence of this type of failure is commonly called the *transaction recovery*. The second type

of failure is caused by system crashes, such as media failures, processor failures, communication link breakages, power outages, and so on. Ensuring transaction atomicity in the presence of system crashes is called *crash recovery*. An important difference between the two types of failures is that during some types of system crashes, the information in volatile storage may be lost or inaccessible. Both types of recovery are parts of the reliability issue, which we discuss in considerable detail in Chapter 12.

10.2.2 Consistency

The *consistency* of a transaction is simply its correctness. In other words, a transaction is a correct program that maps one consistent database state to another. Verifying that transactions are consistent is the concern of integrity enforcement, covered in Chapter 5. Ensuring transaction consistency as defined at the beginning of this chapter, on the other hand, is the objective of concurrency control mechanisms, which we discuss in Chapter 11.

There is an interesting classification of consistency that parallels our discussion above and is equally important. This classification groups databases into four levels of consistency [Gray et al., 1976]. In the following definition (which is taken verbatim from the original paper), *dirty* data refers to data values that have been updated by a transaction prior to its commitment. Then, based on the concept of dirty data, the four levels are defined as follows:

“Degree 3: Transaction T sees *degree 3 consistency* if:

1. T does not overwrite dirty data of other transactions.
2. T does not commit any writes until it completes all its writes [i.e., until the end of transaction (EOT)].
3. T does not read dirty data from other transactions.
4. Other transactions do not dirty any data read by T before T completes.

Degree 2: Transaction T sees *degree 2 consistency* if:

1. T does not overwrite dirty data of other transactions.
2. T does not commit any writes before EOT.
3. T does not read dirty data from other transactions.

Degree 1: Transaction T sees *degree 1 consistency* if:

1. T does not overwrite dirty data of other transactions.
2. T does not commit any writes before EOT.

Degree 0: Transaction T sees *degree 0 consistency* if:

1. T does not overwrite dirty data of other transactions.”

Of course, it is true that a higher degree of consistency encompasses all the lower degrees. The point in defining multiple levels of consistency is to provide application programmers the flexibility to define transactions that operate at different levels. Consequently, while some transactions operate at Degree 3 consistency level, others may operate at lower levels and may see, for example, dirty data.

10.2.3 Isolation

Isolation is the property of transactions that requires each transaction to see a consistent database at all times. In other words, an executing transaction cannot reveal its results to other concurrent transactions before its commitment.

There are a number of reasons for insisting on isolation. One has to do with maintaining the interconsistency of transactions. If two concurrent transactions access a data item that is being updated by one of them, it is not possible to guarantee that the second will read the correct value.

Example 10.8. Consider the following two concurrent transactions (T_1 and T_2), both of which access data item x . Assume that the value of x before they start executing is 50.

T_1 : Read(x)	T_2 : Read(x)
$x \leftarrow x + 1$	$x \leftarrow x + 1$
Write(x)	Write(x)
Commit	Commit

The following is one possible sequence of execution of the actions of these transactions:

T_1 : Read(x)
T_1 : $x \leftarrow x + 1$
T_1 : Write(x)
T_1 : Commit
T_2 : Read(x)
T_2 : $x \leftarrow x + 1$
T_2 : Write(x)
T_2 : Commit

In this case, there are no problems; transactions T_1 and T_2 are executed one after the other and transaction T_2 reads 51 as the value of x . Note that if, instead, T_2 executes before T_1 , T_2 reads 51 as the value of x . So, if T_1 and T_2 are executed one after the other (regardless of the order), the second transaction will read 51 as

the value of x and x will have 52 as its value at the end of execution of these two transactions. However, since transactions are executing concurrently, the following execution sequence is also possible:

```

 $T_1$ : Read( $x$ )
 $T_1$ :  $x \leftarrow x + 1$ 
 $T_2$ : Read( $x$ )
 $T_1$ : Write( $x$ )
 $T_2$ :  $x \leftarrow x + 1$ 
 $T_2$ : Write( $x$ )
 $T_1$ : Commit
 $T_2$ : Commit

```

In this case, transaction T_2 reads 50 as the value of x . This is incorrect since T_2 reads x while its value is being changed from 50 to 51. Furthermore, the value of x is 51 at the end of execution of T_1 and T_2 since T_2 's Write will overwrite T_1 's Write. ♦

Ensuring isolation by not permitting incomplete results to be seen by other transactions, as the previous example shows, solves the *lost updates* problem. This type of isolation has been called *cursor stability*. In the example above, the second execution sequence resulted in the effects of T_1 being lost³. A second reason for isolation is *cascading aborts*. If a transaction permits others to see its incomplete results before committing and then decides to abort, any transaction that has read its incomplete values will have to abort as well. This chain can easily grow and impose considerable overhead on the DBMS.

It is possible to treat consistency levels discussed in the preceding section from the perspective of the isolation property (thus demonstrating the dependence between isolation and consistency). As we move up the hierarchy of consistency levels, there is more isolation among transactions. Degree 0 provides very little isolation other than preventing lost updates. However, since transactions commit write operations before the entire transaction is completed (and committed), if an abort occurs after some writes are committed to disk, the updates to data items that have been committed will need to be undone. Since at this level other transactions are allowed to read the dirty data, it may be necessary to abort them as well. Degree 2 consistency avoids cascading aborts. Degree 3 provides full isolation which forces one of the conflicting transactions to wait until the other one terminates. Such execution sequences are called *strict* and will be discussed further in the next chapter. It is obvious that the issue of isolation is directly related to database consistency and is therefore the topic of concurrency control.

³ A more dramatic example may be to consider x to be your bank account and T_1 a transaction that executes as a result of your *depositing* money into your account. Assume that T_2 is a transaction that is executing as a result of your spouse *withdrawing* money from the account at another branch. If the same problem as described in Example 10.8 occurs and the results of T_1 are lost, you will be terribly unhappy. If, on the other hand, the results of T_2 are lost, the bank will be furious. A similar argument can be made for the reservation transaction example we have been considering.

ANSI, as part of the SQL2 (also known as SQL-92) standard specification, has defined a set of isolation levels [ANSI, 1992]. SQL isolation levels are defined on the basis of what ANSI call *phenomena* which are situations that can occur if proper isolation is not maintained. Three phenomena are specified:

Dirty Read: As defined earlier, dirty data refer to data items whose values have been modified by a transaction that has not yet committed. Consider the case where transaction T_1 modifies a data item value, which is then read by another transaction T_2 before T_1 performs a Commit or Abort. In case T_1 aborts, T_2 has read a value which never exists in the database.

A precise specification⁴ of this phenomenon is as follows (where subscripts indicate the transaction identifiers)

$\dots, W_1(x), \dots, R_2(x), \dots, C_1(\text{or } A_1), \dots, C_2(\text{or } A_2)$

or

$\dots, W_1(x), \dots, R_2(x), \dots, C_2(\text{or } A_2), \dots, C_1(\text{or } A_1)$

Non-repeatable or Fuzzy read: Transaction T_1 reads a data item value. Another transaction T_2 then modifies or deletes that data item and commits. If T_1 then attempts to reread the data item, it either reads a different value or it can't find the data item at all; thus two reads within the same transaction T_1 return different results.

A precise specification of this phenomenon is as follows:

$\dots, R_1(x), \dots, W_2(x), \dots, C_1(\text{or } A_1), \dots, C_2(\text{or } A_2)$

or

$\dots, R_1(x), \dots, W_2(x), \dots, C_2(\text{or } A_2), \dots, C_1(\text{or } A_1)$

Phantom: The phantom condition that was defined earlier occurs when T_1 does a search with a predicate and T_2 inserts new tuples that satisfy the predicate. Again, the precise specification of this phenomenon is (where P is the search predicate)

$\dots, R_1(P), \dots, W_2(y \text{ in } P), \dots, C_1(\text{or } A_1), \dots, C_2(\text{or } A_2)$

or

$\dots, R_1(P), \dots, W_2(y \text{ in } P), \dots, C_2(\text{or } A_2), \dots, C_1(\text{or } A_1)$

⁴ The precise specifications of these phenomena are due to Berenson et al. [1995] and correspond to their *loose interpretations* which they indicate are the more appropriate interpretations.

Based on these phenomena, the isolation levels are defined as follows. The objective of defining multiple isolation levels is the same as defining multiple consistency levels.

Read uncommitted: For transactions operating at this level all three phenomena are possible.

Read committed: Fuzzy reads and phantoms are possible, but dirty reads are not.

Repeatable read: Only phantoms are possible.

Anomaly serializable: None of the phenomena are possible.

ANSI SQL standard uses the term “serializable” rather than “anomaly serializable.” However, a serializable isolation level, as precisely defined in the next chapter, cannot be defined solely in terms of the three phenomena identified above; thus this isolation level is called “anomaly serializable” [Berenson et al., 1995]. The relationship between SQL isolation levels and the four levels of consistency defined in the previous section are also discussed in [Berenson et al., 1995].

One non-serializable isolation level that is commonly implemented in commercial products is *snapshot isolation* [Berenson et al., 1995]. Snapshot isolation provides repeatable reads, but not serializable isolation. Each transaction “sees” a snapshot of the database when it starts and its reads and writes are performed on this snapshot – thus the writes are not visible to other transactions and it does not see the writes of other transactions.

10.2.4 Durability

Durability refers to that property of transactions which ensures that once a transaction commits, its results are permanent and cannot be erased from the database. Therefore, the DBMS ensures that the results of a transaction will survive subsequent system failures. This is exactly why in Example 10.2 we insisted that the transaction commit before it informs the user of its successful completion. The durability property brings forth the issue of *database recovery*, that is, how to recover the database to a consistent state where all the committed actions are reflected. This issue is discussed further in Chapter 12.

10.3 Types of Transactions

A number of transaction models have been proposed in literature, each being appropriate for a class of applications. The fundamental problem of providing “ACID”ity usually remains, but the algorithms and techniques that are used to address them may be considerably different. In some cases, various aspects of ACID requirements are relaxed, removing some problems and adding new ones. In this section we provide

an overview of some of the transaction models that have been proposed and then identify our focus in Chapters 11 and 12.

Transactions have been classified according to a number of criteria. One criterion is the duration of transactions. Accordingly, transactions may be classified as *online* or *batch* [Gray, 1987]. These two classes are also called *short-life* and *long-life* transactions, respectively. Online transactions are characterized by very short execution/response times (typically, on the order of a couple of seconds) and by access to a relatively small portion of the database. This class of transactions probably covers a large majority of current transaction applications. Examples include banking transactions and airline reservation transactions.

Batch transactions, on the other hand, take longer to execute (response time being measured in minutes, hours, or even days) and access a larger portion of the database. Typical applications that might require batch transactions are design databases, statistical applications, report generation, complex queries, and image processing. Along this dimension, one can also define a *conversational* transaction, which is executed by interacting with the user issuing it.

Another classification that has been proposed is with respect to the organization of the read and write actions. The examples that we have considered so far intermix their read and write actions without any specific ordering. We call this type of transactions *general*. If the transactions are restricted so that all the read actions are performed before any write action, the transaction is called a *two-step* transaction [Papadimitriou, 1979]. Similarly, if the transaction is restricted so that a data item has to be read before it can be updated (written), the corresponding class is called *restricted* (or *read-before-write*) [Stearns et al., 1976]. If a transaction is both two-step and restricted, it is called a *restricted two-step* transaction. Finally, there is the *action* model of transactions [Kung and Papadimitriou, 1979], which consists of the restricted class with the further restriction that each \langle read, write \rangle pair be executed atomically. This classification is shown in Figure 10.3, where the generality increases upward.

Example 10.9. The following are some examples of the above-mentioned models. We omit the declaration and commit commands.

General:

$$T_1 : \{R(x), R(y), W(y), R(z), W(x), W(z), W(w), C\}$$

Two-step:

$$T_2 : \{R(x), R(y), R(z), W(x), W(z), W(y), W(w), C\}$$

Restricted:

$$T_3 : \{R(x), R(y), W(y), R(z), W(x), W(z), R(w), W(w), C\}$$

Note that T_3 has to read w before writing.

Two-step restricted:

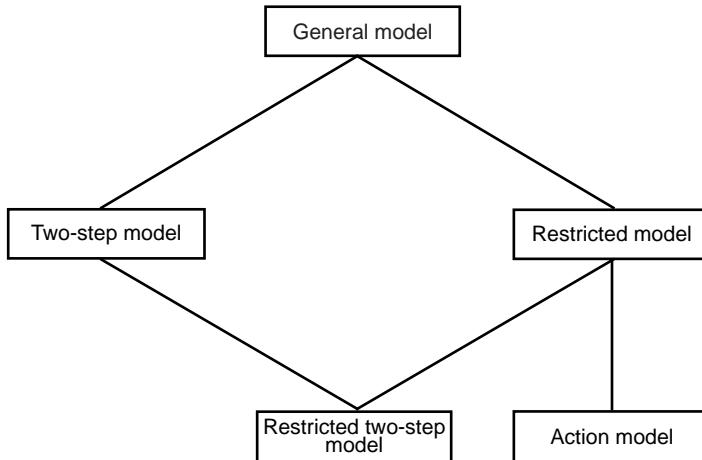


Fig. 10.3 Various Transaction Models (From: C.H. Papadimitriou and P.C. Kanellakis, ON CONCURRENCY CONTROL BY MULTIPLE VERSIONS. ACM Trans. Database Sys.; December 1984; 9(1): 89–99.)

$$T_4 : \{R(x), R(y), R(z), R(w), W(x), W(z), W(y), W(w), C\}$$

Action:

$$T_5 : \{[R(x), W(x)], [R(y), W(y)], [R(z), W(z)], [R(w), W(w)], C\}$$

Note that each pair of actions within square brackets is executed atomically. ♦

Transactions can also be classified according to their structure. We distinguish four broad categories in increasing complexity: *flat transactions*, *closed nested transactions* as in [Moss, 1985], and *open nested transactions* such as sagas [Garcia-Molina and Salem, 1987], and *workflow models* which, in some cases, are combinations of various nested forms. This classification is arguably the most dominant one and we will discuss it at some length.

10.3.1 Flat Transactions

Flat transactions have a single start point (**Begin_transaction**) and a single termination point (**End_transaction**). All our examples in this section are of this type. Most of the transaction management work in databases has concentrated on flat transactions. This model will also be our main focus in this book, even though we discuss management techniques for other transaction types, where appropriate.

10.3.2 Nested Transactions

An alternative transaction model is to permit a transaction to include other transactions with their own begin and commit points. Such transactions are called *nested* transactions. These transactions that are embedded in another one are usually called *subtransactions*.

Example 10.10. Let us extend the reservation transaction of Example 10.2. Most travel agents will make reservations for hotels and car rentals in addition to the flights. If one chooses to specify all of this as one transaction, the reservation transaction would have the following structure:

```

Begin_transaction Reservation
begin
    Begin_transaction Airline
        ...
    end. {Airline}
    Begin_transaction Hotel
        ...
    end. {Hotel}
    Begin_transaction Car
        ...
    end. {Car}
end.

```



Nested transactions have received considerable interest as a more generalized transaction concept. The level of nesting is generally open, allowing subtransactions themselves to have nested transactions. This generality is necessary to support application areas where transactions are more complex than in traditional data processing.

In this taxonomy, we differentiate between *closed* and *open* nesting because of their termination characteristics. Closed nested transactions [Moss, 1985] 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 top-most level. Open nesting 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 [Garcia-Molina and Salem, 1987; Garcia-Molina et al., 1990] and split transactions [Pu, 1988] are examples of open nesting.

A saga is a “sequence of transactions that can be interleaved with other transactions” [Garcia-Molina and Salem, 1987]. The DBMS guarantees that either all the transactions in a saga are successfully completed or *compensating transactions* [Garcia-Molina, 1983; Korth et al., 1990] are run to recover from a partial execution. A compensating transaction effectively does the inverse of the transaction that it is associated with. For example, if the transaction adds \$100 to a bank account,

its compensating transaction deducts \$100 from the same bank account. If a transaction is viewed as a function that maps the old database state to a new database state, its compensating transaction is the inverse of that function.

Two properties of sagas are: (1) only two levels of nesting are allowed, and (2) at the outer level, the system does not support full atomicity. Therefore, a saga differs from a closed nested transaction in that its level structure is more restricted (only 2) and that it is open (the partial results of component transactions or sub-sagas are visible to the outside). Furthermore, the transactions that make up a saga have to be executed sequentially.

The saga concept is extended and placed within a more general model that deals with long-lived transactions and with activities that consist of multiple steps [Garcia-Molina et al., 1990]. The fundamental concept of the model is that of a module that captures code segments each of which accomplishes a given task and access a database in the process. The modules are modeled (at some level) as sub-sagas that communicate with each other via messages over ports. The transactions that make up a saga can be executed in parallel. The model is multi-layer where each subsequent layer adds a level of abstraction.

The advantages of nested transactions are the following. First, they provide a higher-level of concurrency among transactions. Since a transaction consists of a number of other transactions, more concurrency is possible within a single transaction. For example, if the reservation transaction of Example 10.10 is implemented as a flat transaction, it may not be possible to access records about a specific flight concurrently. In other words, if one travel agent issues the reservation transaction for a given flight, any concurrent transaction that wishes to access the same flight data will have to wait until the termination of the first, which includes the hotel and car reservation activities in addition to flight reservation. However, a nested implementation will permit the second transaction to access the flight data as soon as the Airline subtransaction of the first reservation transaction is completed. In other words, it may be possible to perform a finer level of synchronization among concurrent transactions.

A second argument in favor of nested transactions is related to recovery. It is possible to recover independently from failures of each subtransaction. This limits the damage to a smaller part of the transaction, making it less costly to recover. In a flat transaction, if any operation fails, the entire transaction has to be aborted and restarted, whereas in a nested transaction, if an operation fails, only the subtransaction containing that operation needs to be aborted and restarted.

Finally, it is possible to create new transactions from existing ones simply by inserting the old one inside the new one as a subtransaction.

10.3.3 Workflows

Flat transactions model relatively simple and short activities very well. However, they are less appropriate for modeling longer and more elaborate activities. That is

the reason for the development of the various nested transaction models discussed above. It has been argued that these extensions are not sufficiently powerful to model business activities: “after several decades of data processing, we have learned that we have not won the battle of modeling and automating complex enterprises” [Medina-Mora et al., 1993]. To meet these needs, more complex transaction models which are combinations of open and nested transactions have been proposed. There are well-justified arguments for not calling these transactions, since they hardly follow any of the ACID properties; a more appropriate name that has been proposed is a *workflow* [Dogac et al., 1998b; Georgakopoulos et al., 1995].

The term “workflow,” unfortunately, does not have a clear and uniformly accepted meaning. A working definition is that a workflow is “a collection of *tasks* organized to accomplish some business process.” [Georgakopoulos et al., 1995]. This definition, however, leaves a lot undefined. This is perhaps unavoidable given the very different contexts where this term is used. Three types of workflows are identified [Georgakopoulos et al., 1995]:

1. *Human-oriented workflows*, which involve humans in performing the tasks. The system support is provided to facilitate collaboration and coordination among humans, but it is the humans themselves who are ultimately responsible for the consistency of the actions.
 2. *System-oriented workflows* are those that consist of computation-intensive and specialized tasks that can be executed by a computer. The system support in this case is substantial and involves concurrency control and recovery, automatic task execution, notification, etc.
 3. *Transactional workflows* range in between human-oriented and system-oriented workflows and borrow characteristics from both. They involve “coordinated execution of multiple tasks that (a) may involve humans, (b) require access to HAD [heterogeneous, autonomous, and/or distributed] systems, and (c) support selective use of transactional properties [i.e., ACID properties] for individual tasks or entire workflows.” [Georgakopoulos et al., 1995].
- Among the features of transactional workflows, the selective use of transactional properties is particularly important as it characterizes possible relaxations of ACID properties.

In this book, our primary interest is with transactional workflows. There have been many transactional workflow proposals [Elmagarmid et al., 1990; Nodine and Zdonik, 1990; Buchmann et al., 1982; Dayal et al., 1991; Hsu, 1993], and they differ in a number of ways. The common point among them is that a workflow is defined as an *activity* consisting of a set of tasks with well-defined precedence relationship among them.

Example 10.11. Let us further extend the reservation transaction of Example 10.3. The entire reservation activity consists of the following tasks and involves the following data:

- Customer request is obtained (task T_1) and Customer Database is accessed to obtain customer information, preferences, etc.;
- Airline reservation is performed (T_2) by accessing the Flight Database;
- Hotel reservation is performed (T_3), which may involve sending a message to the hotel involved;
- Auto reservation is performed (T_4), which may also involve communication with the car rental company;
- Bill is generated (T_5) and the billing info is recorded in the billing database.

Figure 10.4 depicts this workflow where there is a serial dependency of T_2 on T_1 , and T_3, T_4 on T_2 ; however, T_3 and T_4 (hotel and car reservations) are performed in parallel and T_5 waits until their completion. ♦

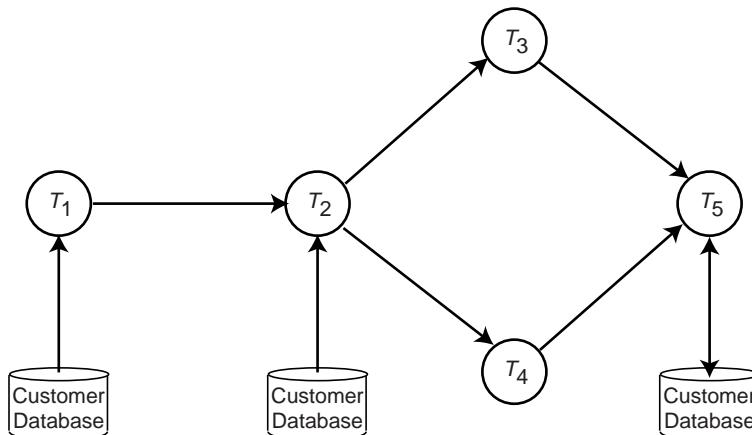


Fig. 10.4 Example Workflow

A number of workflow models go beyond this basic model by both defining more precisely what tasks can be and by allocating different relationships among the tasks. In the following, we define one model that is similar to the models of [Buchmann et al. \[1982\]](#) and [Dayal et al. \[1991\]](#).

A workflow is modeled as an *activity* with open nesting semantics in that it permits partial results to be visible outside the activity boundaries. Thus, tasks that make up the activity are allowed to commit individually. Tasks may be other activities (with the same open transaction semantics) or closed nested transactions that make their results visible to the entire system when they commit. Even though an activity can have both other activities and closed nested transactions as its component, a closed nested transaction task can only be composed of other closed nested transactions (i.e., once closed nesting semantics begins, it is maintained for all components).

An activity commits when its components are ready to commit. However, the components commit individually, without waiting for the root activity to commit.

This raises problems in dealing with aborts since when an activity aborts, all of its components should be aborted. The problem is dealing with the components that have already committed. Therefore, compensating transactions are defined for the components of an activity. Thus, if a component has already committed when an activity aborts, the corresponding compensating transaction is executed to “undo” its effects.

Some components of an activity may be marked as *vital*. When a vital component aborts, its parent must also abort. If a non-vital component of a workflow model aborts, it may continue executing. A workflow, on the other hand, always aborts when one of its components aborts. For example, in the reservation workflow of Example 10.11, T_2 (airline reservation) and T_3 (hotel reservation) may be declared as vital so that if an airline reservation or a hotel reservation cannot be made, the workflow aborts and the entire trip is canceled. However, if a car reservation cannot be committed, the workflow can still successfully terminate.

It is possible to define *contingency tasks* that are invoked if their counterparts fail. For example, in the Reservation example presented earlier, one can specify that the contingency to making a reservation at Hilton is to make a reservation at Sheraton. Thus, if the hotel reservation component for Hilton fails, the Sheraton alternative is tried rather than aborting the task and the entire workflow.

10.4 Architecture Revisited

With the introduction of the transaction concept, we need to revisit the architectural model introduced in Chapter 1. We do not need to revise the model but simply need to expand the role of the distributed execution monitor.

The distributed execution monitor consists of two modules: a *transaction manager* (TM) and a *scheduler* (SC). The transaction manager is responsible for coordinating the execution of the database operations on behalf of an application. The scheduler, on the other hand, is responsible for the implementation of a specific concurrency control algorithm for synchronizing access to the database.

A third component that participates in the management of distributed transactions is the local recovery managers (LRM) that exist at each site. Their function is to implement the local procedures by which the local database can be recovered to a consistent state following a failure.

Each transaction originates at one site, which we will call its *originating site*. The execution of the database operations of a transaction is coordinated by the TM at that transaction’s originating site.

The transaction managers implement an interface for the application programs which consists of five commands: begin_transaction, read, write, commit, and abort. The processing of each of these commands in a non-replicated distributed DBMS is discussed below at an abstract level. For simplicity, we ignore the scheduling of concurrent transactions as well as the details of how data is physically retrieved by the data processor. These assumptions permit us to concentrate on the interface to

the TM. The details are presented in the Chapters 11 and 12, while the execution of these commands in a replicated distributed database is discussed in Chapter 13.

1. *Begin_transaction*. This is an indicator to the TM that a new transaction is starting. The TM does some bookkeeping, such as recording the transaction's name, the originating application, and so on, in coordination with the data processor.
2. *Read*. If the data item to be read is stored locally, its value is read and returned to the transaction. Otherwise, the TM finds where the data item is stored and requests its value to be returned (after appropriate concurrency control measures are taken).
3. *Write*. If the data item is stored locally, its value is updated (in coordination with the data processor). Otherwise, the TM finds where the data item is located and requests the update to be carried out at that site after appropriate concurrency control measures are taken).
4. *Commit*. The TM coordinates the sites involved in updating data items on behalf of this transaction so that the updates are made permanent at every site.
5. *Abort*. The TM makes sure that no effects of the transaction are reflected in any of the databases at the sites where it updated data items.

In providing these services, a TM can communicate with SCs and data processors at the same or at different sites. This arrangement is depicted in Figure 10.5.

As we indicated in Chapter 1, the architectural model that we have described is only an abstraction that serves a pedagogical purpose. It enables the separation of many of the transaction management issues and their independent and isolated discussion. In Chapter 11 we focus on the interface between a TM and an SC and between an SC and a data processor, in addition to the scheduling algorithms. In Chapter 12 we consider the execution strategies for the commit and abort commands in a distributed environment, in addition to the recovery algorithms that need to be implemented for the recovery manager. In Chapter 13, we extend this discussion to the case of replicated databases. We should point out that the computational model that we described here is not unique. Other models have been proposed such as, for example, using a private workspace for each transaction.

10.5 Conclusion

In this chapter we introduced the concept of a transaction as a unit of consistent and reliable access to the database. The properties of transactions indicate that they are larger atomic units of execution which transform one consistent database to another consistent database. The properties of transactions also indicate what the requirements for managing them are, which is the topic of the next two chapters. Consistency requires a definition of integrity enforcement (which we did in Chapter

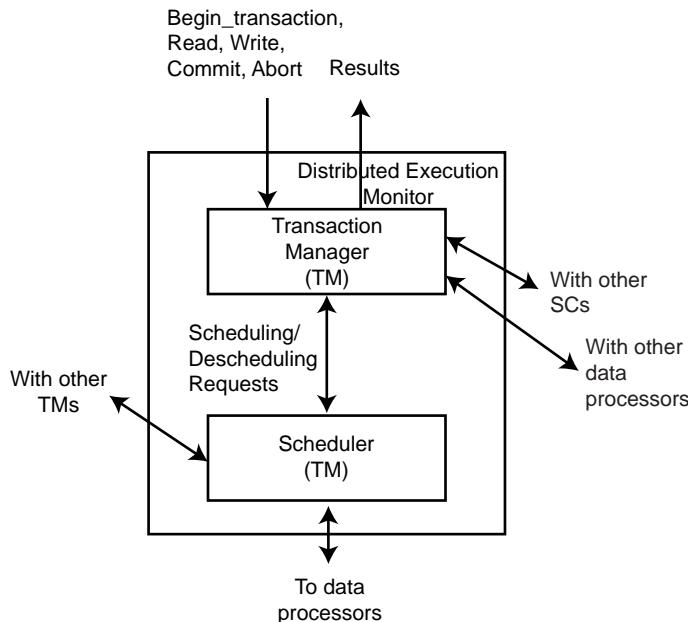


Fig. 10.5 Detailed Model of the Distributed Execution Monitor

5), as well as concurrency control algorithms (which is the topic of Chapter 11). Concurrency control also deals with the issue of isolation. Durability and atomicity properties of transactions require a discussion of reliability, which we cover in Chapter 12. Specifically, durability is supported by various commit protocols and commit management, whereas atomicity requires the development of appropriate recovery protocols.

10.6 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]. 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 good collection of papers that focus on the concurrency control and reliability aspects of distributed systems is [Bhargava, 1987]. Two books focus on the performance of concurrency control mechanisms with a focus on centralized systems [Kumar, 1996; Thomasian, 1996]. Distributed concurrency control is the topic of [Cellary et al., 1988].

Advanced transaction models are discussed and various examples are given in [Elmagarmid, 1992]. Nested transactions are also covered in [Lynch et al., 1993]. A good introduction to workflow systems is [Georgakopoulos et al., 1995]. The same topic is covered in detail in [Dogac et al., 1998b].

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.

The discussion concerning transaction classification in Section 10.3 comes from a number of sources. Part of it is from [Farrag, 1986]. The structure discussion is from [Özsu, 1994] and [Buchmann et al., 1982], where the authors combine transaction structure with the structure of the objects that these transactions operate upon to develop a more complete classification.

There are numerous papers dealing with various transaction management issues. The ones referred to in this chapter are those that deal with the concept of a transaction. More detailed references on their management are left to Chapters 11 and 12.

Chapter 11

Distributed Concurrency Control

As we discussed in Chapter 10, concurrency control deals with the isolation and consistency properties of transactions. The distributed concurrency control mechanism of a distributed DBMS ensures that the consistency of the database, as defined in Section 10.2.2, is maintained in a multiuser distributed environment. If transactions are internally consistent (i.e., do not violate any consistency constraints), the simplest way of achieving this objective is to execute each transaction alone, one after another. It is obvious that such an alternative is only of theoretical interest and would not be implemented in any practical system, since it minimizes the system throughput. The level of concurrency (i.e., the number of concurrent transactions) is probably the most important parameter in distributed systems [Balter et al., 1982]. Therefore, the concurrency control mechanism attempts to find a suitable trade-off between maintaining the consistency of the database and maintaining a high level of concurrency.

In this chapter, we make two major assumptions: the distributed system is fully reliable and does not experience any failures (of hardware or software), and the database is not replicated. Even though these are unrealistic assumptions, they permit us to delineate the issues related to the management of concurrency from those related to the operation of a reliable distributed system and those related to maintaining replicas. In Chapter 12, we discuss how the algorithms that are presented in this chapter need to be enhanced to operate in an unreliable environment. In Chapter 13 we address the issues related to replica management.

We start our discussion of concurrency control with a presentation of serializability theory in Section 11.1. Serializability is the most widely accepted correctness criterion for concurrency control algorithms. In Section 11.2 we present a taxonomy of algorithms that will form the basis for most of the discussion in the remainder of the chapter. Sections 11.3 and 11.4 cover the two major classes of algorithms: locking-based and timestamp ordering-based. Both locking and timestamp ordering classes cover what is called pessimistic algorithms; optimistic concurrency control is discussed in Section 11.5. Any locking-based algorithm may result in deadlocks, requiring special management methods. Various deadlock management techniques are therefore the topic of Section 11.6. In Section 11.7, we discuss “relaxed” con-

currency control approaches. These are mechanisms which use weaker correctness criteria than serializability, or relax the isolation property of transactions.

11.1 Serializability Theory

In Section 10.1.3 we discussed the issue of isolating transactions from one another in terms of their effects on the database. We also pointed out that if the concurrent execution of transactions leaves the database in a state that can be achieved by their serial execution in some order, problems such as lost updates will be resolved. This is exactly the point of the serializability argument. The remainder of this section addresses serializability issues more formally.

A *history* R (also called a *schedule*) is defined over a set of transactions $T = \{T_1, T_2, \dots, T_n\}$ and specifies an interleaved order of execution of these transactions' operations. Based on the definition of a transaction introduced in Section 10.1, the history can be specified as a partial order over T . We need a few preliminaries, though, before we present the formal definition.

Recall the definition of conflicting operations that we gave in Chapter 10. Two operations $O_{ij}(x)$ and $O_{kl}(x)$ (i and k representing transactions and are not necessarily distinct) accessing the same database entity x are said to be in *conflict* if at least one of them is a write operation. Note two things in this definition. First, read operations do not conflict with each other. We can, therefore, talk about two types of conflicts: *read-write* (or *write-read*), and *write-write*. Second, the two operations can belong to the same transaction or to two different transactions. In the latter case, the two transactions are said to be *conflicting*. Intuitively, the existence of a conflict between two operations indicates that their order of execution is important. The ordering of two read operations is insignificant.

We first define a *complete history*, which defines the execution order of all operations in its domain. We will then define a history as a prefix of a complete history. Formally, a complete history H_T^c defined over a set of transactions $T = \{T_1, T_2, \dots, T_n\}$ is a partial order $H_T^c = \{\Sigma_T, \prec_H\}$ where

1. $\Sigma_T = \bigcup_{i=1}^n \Sigma_i$.
2. $\prec_H \supseteq \bigcup_{i=1}^n \prec_{T_i}$.
3. For any two conflicting operations $O_{ij}, O_{kl} \in \Sigma_T$, either $O_{ij} \prec_H O_{kl}$, or $O_{kl} \prec_H O_{ij}$.

The first condition simply states that the domain of the history is the union of the domains of individual transactions. The second condition defines the ordering relation of the history as a superset of the ordering relations of individual transactions. This maintains the ordering of operations within each transaction. The final condition simply defines the execution order among conflicting operations in H .

Example 11.1. Consider the two transactions from Example 10.8, which were as follows:

T_1 : Read(x)	T_2 : Read(x)
$x \leftarrow x + 1$	$x \leftarrow x + 1$
Write(x)	Write(x)
Commit	Commit

A possible complete history H_T^c over $T = \{T_1, T_2\}$ is the partial order $H_T^c = \{\Sigma_T, \prec_T\}$ where

$$\Sigma_1 = \{R_1(x), W_1(x), C_1\}$$

$$\Sigma_2 = \{R_2(x), W_2(x), C_2\}$$

Thus

$$\Sigma_T = \Sigma_1 \cup \Sigma_2 = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\}$$

and

$$\prec_H = \{(R_1, R_2), (R_1, W_1), (R_1, C_1), (R_1, W_2), (R_1, C_2), (R_2, W_1), (R_2, C_1), (R_2, W_2), (R_2, C_2), (W_1, C_1), (W_1, W_2), (W_1, C_2), (C_1, W_2), (C_1, C_2), (W_2, C_2)\}$$

which can be specified as a DAG as depicted in Figure 11.1. Note that consistent with our earlier adopted convention (see Example 10.7), we do not draw the arcs that are implied by transitivity [e.g., (R_1, C_1)].

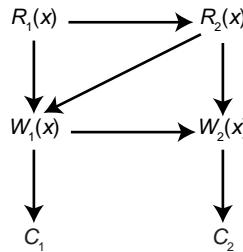


Fig. 11.1 DAG Representation of a Complete History

It is quite common to specify a history as a listing of the operations in Σ_T , where their execution order is relative to their order in this list. Thus H_T^c can be specified as

$$H_T^c = \{R_1(x), R_2(x), W_1(x), C_1, W_2(x), C_2\}$$



A history is defined as a prefix of a complete history. A prefix of a partial order can be defined as follows. Given a partial order $P = \{\Sigma, \prec\}$, $P' = \{\Sigma', \prec'\}$ is a *prefix* of P if

1. $\Sigma' \subseteq \Sigma$;
2. $\forall e_i \in \Sigma', e_1 \prec' e_2$ if and only if $e_1 \prec e_2$; and
3. $\forall e_i \in \Sigma'$, if $\exists e_j \in \Sigma$ and $e_j \prec e_i$, then $e_j \in \Sigma'$.

The first two conditions define P' as a *restriction* of P on domain Σ' , whereby the ordering relations in P are maintained in P' . The last condition indicates that for any element of Σ' , all its predecessors in Σ have to be included in Σ' as well.

What does this definition of a history as a prefix of a partial order provide for us? The answer is simply that we can now deal with incomplete histories. This is useful for a number of reasons. From the perspective of the serializability theory, we deal only with conflicting operations of transactions rather than with all operations. Furthermore, and perhaps more important, when we introduce failures, we need to be able to deal with incomplete histories, which is what a prefix enables us to do.

The history discussed in Example 11.1 is special in that it is complete. It needs to be complete in order to talk about the execution order of these two transactions' operations. The following example demonstrates a history that is not complete.

Example 11.2. Consider the following three transactions:

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

Commit Commit Commit

A complete history H^c for these transactions is given in Figure 11.2, and a history H (as a prefix of H^c) is depicted in Figure 11.3. ♦

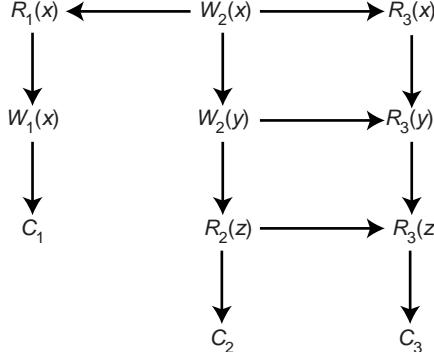


Fig. 11.2 A Complete History

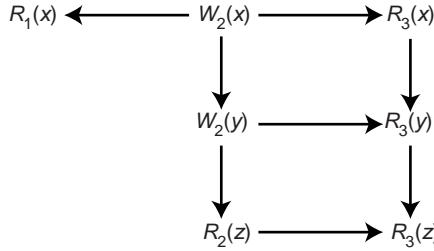


Fig. 11.3 Prefix of Complete History in Figure 11.2

If in a complete history H , the operations of various transactions are not interleaved (i.e., the operations of each transaction occur consecutively), the history is said to be *serial*. As we indicated before, the serial execution of a set of transactions maintains the consistency of the database. This follows naturally from the consistency property of transactions: each transaction, when executed alone on a consistent database, will produce a consistent database.

Example 11.3. Consider the three transactions of Example 11.2. The following history is serial since all the operations of T_2 are executed before all the operations of T_1 and all operations of T_1 are executed before all operations of T_3 ¹.

$$H = \underbrace{\{W_2(x), W_2(y), R_2(z)\}}_{T_2} \underbrace{\{R_1(x), W_1(x)\}}_{T_1} \underbrace{\{R_3(x), R_3(y), R_3(z)\}}_{T_3}$$

One common way to denote this precedence relationship between transaction executions is $T_2 \rightarrow T_1 \rightarrow T_3$ rather than the more formal $T_2 \prec_H T_1 \prec_H T_3$. ♦

Based on the precedence relationship introduced by the partial order, it is possible to discuss the equivalence of histories with respect to their effects on the database. Intuitively, two histories H_1 and H_2 , defined over the same set of transactions T , are *equivalent* if they have the same effect on the database. More formally, two histories, H_1 and H_2 , defined over the same set of transactions T , are said to be *equivalent* if for each pair of conflicting operations O_{ij} and O_{kl} ($i \neq k$), whenever $O_{ij} \prec_{H_1} O_{kl}$, then $O_{ij} \prec_{H_2} O_{kl}$. This is called *conflict equivalence* since it defines equivalence of two histories in terms of the relative order of execution of the conflicting operations in those histories. Here, for the sake of simplicity, we assume that T does not include any aborted transaction. Otherwise, the definition needs to be modified to specify only those conflicting operations that belong to unaborted transactions.

Example 11.4. Again consider the three transactions given in Example 11.2. The following history H' defined over them is conflict equivalent to H given in Example 11.3:

$$H' = \{W_2(x), R_1(x), W_1(x), R_3(x), W_2(y), R_3(y), R_2(z), R_3(z)\}$$

¹ From now on we will generally omit the Commit operation from histories.



We are now ready to define serializability more precisely. A history H is said to be *serializable* if and only if it is conflict equivalent to a serial history. Note that serializability roughly corresponds to degree 3 consistency, which we defined in Section 10.2.2. Serializability so defined is also known as *conflict-based serializability* since it is defined according to conflict equivalence.

Example 11.5. History H' in Example 11.4 is serializable since it is equivalent to the serial history H of Example 11.3. Also note that the problem with the uncontrolled execution of transactions T_1 and T_2 in Example 10.8 was that they could generate an unserializable history. ♦

Now that we have formally defined serializability, we can indicate that the primary function of a concurrency controller is to generate a serializable history for the execution of pending transactions. The issue, then, is to devise algorithms that are guaranteed to generate only serializable histories.

Serializability theory extends in a straightforward manner to the non-replicated (or partitioned) 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 11.6. We will 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 balances 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):

$$\begin{aligned} H_1 &= \{R_1(x), W_1(x), R_2(x)\} \\ H_2 &= \{R_1(y), W_1(y), R_2(y)\} \end{aligned}$$

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, the 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:

$$\begin{aligned} H'_1 &= \{R_1(x), W_1(x), R_2(x)\} \\ H'_2 &= \{R_2(y), R_1(y), W_1(y)\} \end{aligned}$$

Although each local history is still serializable, the serialization orders are different: H'_1 serializes T_1 before T_2 while H'_2 serializes T_2 before T_1 . Therefore, there can be no global history that is serializable. ♦

A weaker version of serializability that has gained importance in recent years is *snapshot isolation* [Berenson et al., 1995] that is now provided as a standard consistency criterion in a number of commercial systems. Snapshot isolation allows read transactions (queries) to read stale data by allowing them to read a snapshot of the database that reflects the committed data at the time the read transaction starts. Consequently, the reads are never blocked by writes, even though they may read old data that may be dirtied by other transactions that were still running when the snapshot was taken. Hence, the resulting histories are not serializable, but this is accepted as a reasonable tradeoff between a lower level of isolation and better performance.

11.2 Taxonomy of Concurrency Control Mechanisms

There are a number of ways that the concurrency control approaches can be classified. One obvious classification criterion is the mode of database distribution. Some algorithms that have been proposed require a fully replicated database, while others can operate on partially replicated or partitioned databases. The concurrency control algorithms may also be classified according to network topology, such as those requiring a communication subnet with broadcasting capability or those working in a star-type network or a circularly connected network.

The most common classification criterion, however, is the synchronization primitive. The corresponding breakdown of the concurrency control algorithms results in two classes [Bernstein and Goodman, 1981]: those algorithms that are based on mutually exclusive access to shared data (locking), and those that attempt to order the execution of the transactions according to a set of rules (protocols). However, these primitives may be used in algorithms with two different viewpoints: the pessimistic view that many transactions will conflict with each other, or the optimistic view that not too many transactions will conflict with one another.

We will thus group the concurrency control mechanisms into two broad classes: pessimistic concurrency control methods and optimistic concurrency control methods. *Pessimistic* algorithms synchronize the concurrent execution of transactions early in their execution life cycle, whereas *optimistic* algorithms delay the synchronization of transactions until their termination. The pessimistic group consists of *locking*-

based algorithms, *ordering* (or *transaction ordering*) *based* algorithms, and *hybrid* algorithms. The optimistic group can, similarly, be classified as locking-based or timestamp ordering-based. This classification is depicted in Figure 11.4.

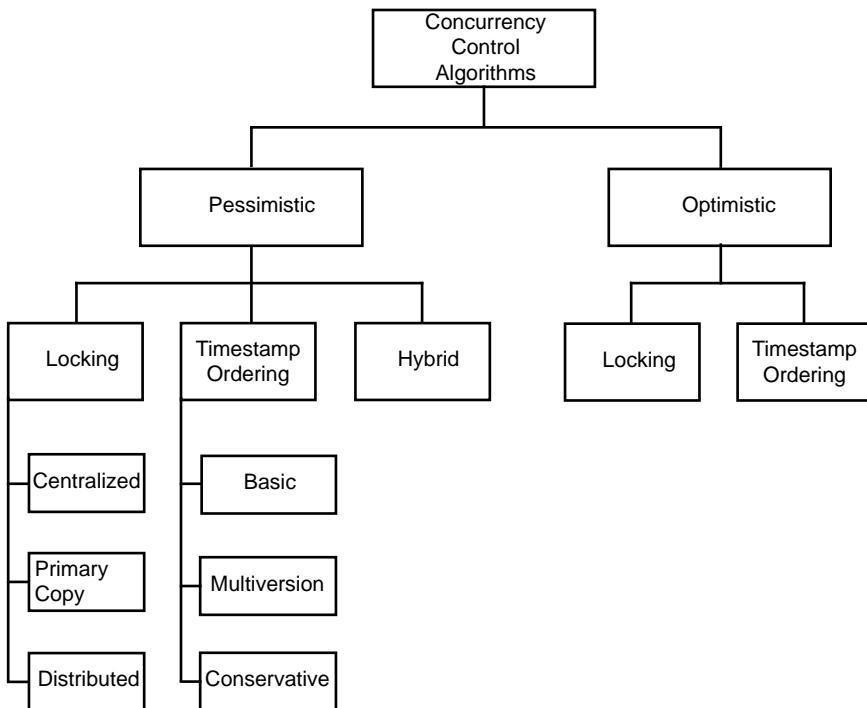


Fig. 11.4 Classification of Concurrency Control Algorithms

In the *locking-based* approach, the synchronization of transactions is achieved by employing physical or logical locks on some portion or granule of the database. The size of these portions (usually called *locking granularity*) is an important issue. However, for the time being, we will ignore it and refer to the chosen granule as a *lock unit*. This class is subdivided further according to where the lock management activities are performed: *centralized* and *decentralized* (or *distributed*) *locking*.

The *timestamp ordering* (TO) class involves organizing the execution order of transactions so that they maintain transaction consistency. This ordering is maintained by assigning timestamps to both the transactions and the data items that are stored in the database. These algorithms can be *basic TO*, *multiversion TO*, or *conservative TO*.

We should indicate that in some locking-based algorithms, timestamps are also used. This is done primarily to improve efficiency and the level of concurrency. We call these *hybrid* algorithms. We will not discuss these algorithms in this chapter since they have not been implemented in any commercial or research prototype distributed

DBMS. The rules for integrating locking and timestamp ordering protocols are discussed by [Bernstein and Goodman \[1981\]](#).

11.3 Locking-Based Concurrency Control Algorithms

The main idea of locking-based concurrency control is to ensure that a data item that is shared by conflicting operations is accessed by one operation at a time. This is accomplished by associating a “lock” with each lock unit. This lock is set by a transaction before it is accessed and is reset at the end of its use. Obviously a lock unit cannot be accessed by an operation if it is already locked by another. Thus a lock request by a transaction is granted only if the associated lock is not being held by any other transaction.

Since we are concerned with synchronizing the conflicting operations of conflicting transactions, there are two types of locks (commonly called *lock modes*) associated with each lock unit: *read lock* (rl) and *write lock* (wl). A transaction T_i that wants to read a data item contained in lock unit x obtains a read lock on x [denoted $rl_i(x)$]. The same happens for write operations. Two lock modes are *compatible* if two transactions that access the same data item can obtain these locks on that data item at the same time. As Figure 11.5 shows, read locks are compatible, whereas read-write or write-write locks are not. Therefore, it is possible, for example, for two transactions to read the same data item concurrently.

	$rl(x)$	$wl(x)$
$rl(x)$	compatible	not compatible
$wl(x)$	not compatible	not compatible

Fig. 11.5 Compatibility Matrix of Lock Modes

The distributed DBMS not only manages locks but also handles the lock management responsibilities on behalf of the transactions. In other words, users do not need to specify when a data item needs to be locked; the distributed DBMS takes care of that every time the transaction issues a read or write operation.

In locking-based systems, the scheduler (see Figure 10.5) is a *lock manager* (LM). The transaction manager passes to the lock manager the database operation (read or write) and associated information (such as the item that is accessed and the identifier of the transaction that issues the database operation). The lock manager then checks if the lock unit that contains the data item is already locked. If so, and if the existing lock mode is incompatible with that of the current transaction, the current operation is delayed. Otherwise, the lock is set in the desired mode and the database operation is passed on to the data processor for actual database access. The transaction manager is then informed of the results of the operation. The termination of a transaction

results in the release of its locks and the initiation of another transaction that might be waiting for access to the same data item.

The locking algorithm as described above will not, unfortunately, properly synchronize transaction executions. This is because to generate serializable histories, the locking and releasing operations of transactions also need to be coordinated. We demonstrate this by an example.

Example 11.7. Consider the following two transactions:

T_1 : Read(x)	T_2 : Read(x)
$x \leftarrow x + 1$	$x \leftarrow x * 2$
Write(x)	Write(x)
Read(y)	Read(y)
$y \leftarrow y - 1$	$y \leftarrow y * 2$
Write(y)	Write(y)
Commit	Commit

The following is a valid history that a lock manager employing the locking algorithm may generate:

$$H = \{wl_1(x), R_1(x), W_1(x), lr_1(x), wl_2(x), R_2(x), w_2(x), lr_2(x), wl_2(y), \\ R_2(y), W_2(y), lr_2(y), wl_1(y), R_1(y), W_1(y), lr_1(y)\}$$

where $lr_i(z)$ indicates the release of the lock on z that transaction T_i holds.

Note that H is not a serializable history. For example, if prior to the execution of these transactions, the values of x and y are 50 and 20, respectively, one would expect their values following execution to be, respectively, either 102 and 38 if T_1 executes before T_2 , or 101 and 39 if T_2 executes before T_1 . However, the result of executing H would give x and y the values 102 and 39. Obviously, H is not serializable. ♦

The problem with history H in Example 11.7 is the following. The locking algorithm releases the locks that are held by a transaction (say, T_i) as soon as the associated database command (read or write) is executed, and that lock unit (say x) no longer needs to be accessed. However, the transaction itself is locking other items (say, y), after it releases its lock on x . Even though this may seem to be advantageous from the viewpoint of increased concurrency, it permits transactions to interfere with one another, resulting in the loss of isolation and atomicity. Hence the argument for *two-phase locking* (2PL).

The two-phase locking rule simply states that no transaction should request a lock after it releases one of its locks. Alternatively, a transaction should not release a lock until it is certain that it will not request another lock. 2PL algorithms execute transactions in two phases. Each transaction has a *growing phase*, where it obtains locks and accesses data items, and a *shrinking phase*, during which it releases locks (Figure 11.6). The *lock point* is the moment when the transaction has achieved all its locks but has not yet started to release any of them. Thus the lock point determines the end of the growing phase and the beginning of the shrinking phase of a transaction. It has been proven that any history generated by a concurrency control algorithm that obeys the 2PL rule is serializable [Eswaran et al., 1976].

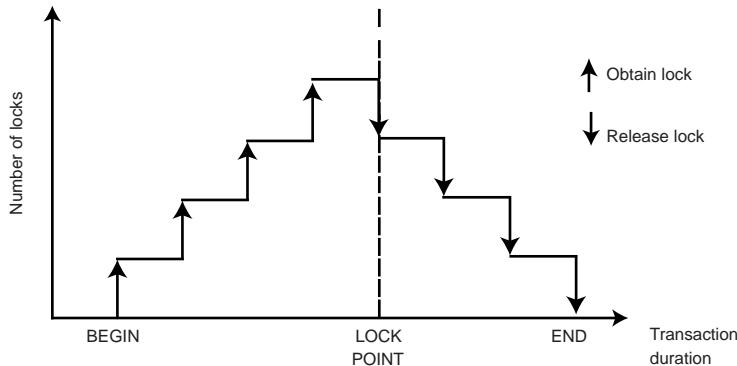
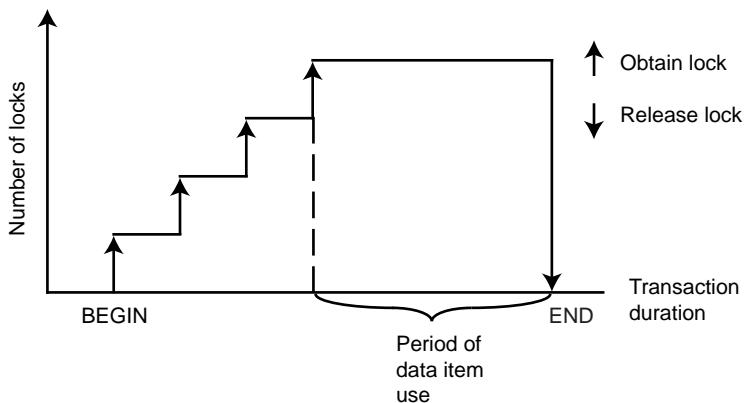
**Fig. 11.6** 2PL Lock Graph**Fig. 11.7** Strict 2PL Lock Graph

Figure 11.6 indicates that the lock manager releases locks as soon as access to that data item has been completed. This permits other transactions awaiting access to go ahead and lock it, thereby increasing the degree of concurrency. However, this is difficult to implement since the lock manager has to know that the transaction has obtained all its locks and will not need to lock another data item. The lock manager also needs to know that the transaction no longer needs to access the data item in question, so that the lock can be released. Finally, if the transaction aborts after it releases a lock, it may cause other transactions that may have accessed the unlocked data item to abort as well. This is known as *cascading aborts*. These problems may be overcome by *strict two-phase locking*, which releases all the locks together when the transaction terminates (commits or aborts). Thus the lock graph is as shown in Figure 11.7.

We should note that even though a 2PL algorithm enforces conflict serializability, it does not allow all histories that are conflict serializable. Consider the following history discussed by [Agrawal and El-Abbadi \[1990\]](#):

$$H = \{W_1(x), R_2(x), W_3(y), W_1(y)\}$$

H is not allowed by 2PL algorithm since T_1 would need to obtain a write lock on y after it releases its write lock on x . However, this history is serializable in the order $T_3 \rightarrow T_1 \rightarrow T_2$. The order of locking can be exploited to design locking algorithms that allow histories such as these [Agrawal and El-Abbadi, 1990].

The main idea is to observe that in serializability theory, the order of serialization of conflicting operations is as important as detecting the conflict in the first place and this can be exploited in defining locking modes. Consequently, in addition to read (shared) and write (exclusive) locks, a third lock mode is defined: *ordered shared*. Ordered shared locking of an object x by transactions T_i and T_j has the following meaning: Given a history H that allows ordered shared locks between operations $o \in T_i$ and $p \in T_j$, if T_i acquires o -lock before T_j acquires p -lock, then o is executed before p . Consider the compatibility table between read and write locks given in Figure 11.5. If the ordered shared mode is added, there are eight variants of this table. Figure 11.5 depicts one of them and two more are shown in Figure 11.8. In Figure 11.8(b), for example, there is an ordered shared relationship between $rl_j(x)$ and $wl_i(x)$ indicating that T_i can acquire a write lock on x while T_j holds a read lock on x as long as the ordered shared relationship from $rl_j(x)$ to $wl_i(x)$ is observed. The eight compatibility tables can be compared with respect to their permissiveness (i.e., with respect to the histories that can be produced using them) to generate a lattice of tables such that the one in Figure 11.5 is the most restrictive and the one in Figure 11.8(b) is the most liberal.

	$rl(x)$	$wl(x)$		$rl(x)$	$wl(x)$
$rl(x)$	compatible	not compatible	$rl(x)$	compatible	ordered shared
$wl(x)$	ordered shared	not compatible	$wl(x)$	ordered shared	ordered shared

Fig. 11.8 Commutativity Table with Ordered Shared Lock Modes

The locking protocol that enforces a compatibility matrix involving ordered shared lock modes is identical to 2PL, except that a transaction may not release any locks as long as any of its locks are on hold. Otherwise circular serialization orders can exist.

Locking-based algorithms may cause deadlocks since they allow exclusive access to resources. It is possible that two transactions that access the same data items may lock them in reverse order, causing each to wait for the other to release its locks causing a deadlock. We discuss deadlock management in Section 11.6.

11.3.1 Centralized 2PL

The 2PL algorithm discussed in the preceding section can easily be extended to the distributed DBMS environment. One way of doing this is to delegate lock management responsibility to a single site only. This means that only one of the sites has a lock manager; the transaction managers at the other sites communicate with it rather than with their own lock managers. This approach is also known as the *primary site* 2PL algorithm [Alberg and Day, 1976].

The communication between the cooperating sites in executing a transaction according to a centralized 2PL (C2PL) algorithm is depicted in Figure 11.9. This communication is between the transaction manager at the site where the transaction is initiated (called 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 item and at which the operation is to be carried out. The order of messages is denoted in the figure.

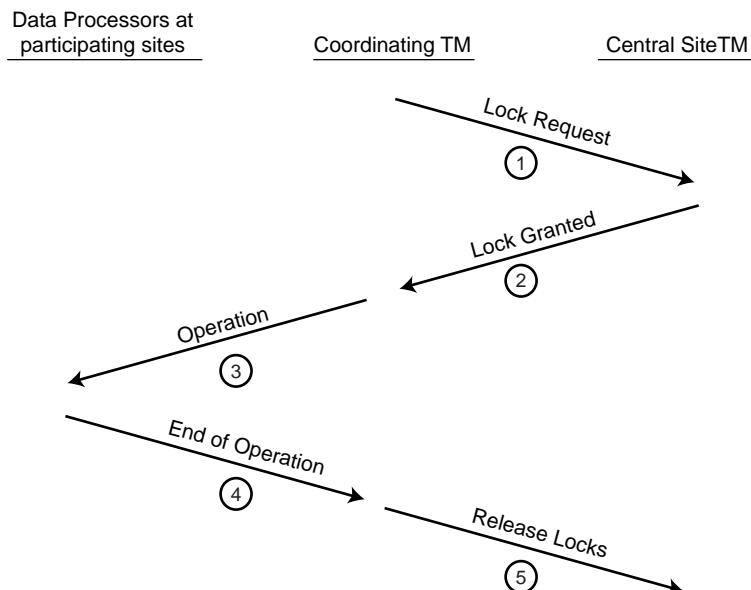


Fig. 11.9 Communication Structure of Centralized 2PL

The centralized 2PL transaction management algorithm (C2PL-TM) that incorporates these changes is given at a very high level in Algorithm 11.1, while the centralized 2PL lock management algorithm (C2PL-LM) is shown in Algorithm 11.2. A highly simplified data processor algorithm (DP) is given in Algorithm 11.3; this is the algorithm that will see major changes when we discuss reliability issues in Chapter 12. For the time being, this is sufficient for our purposes.

There is one important data structure that is used in these algorithms and that is the operation that is defined as a 5-tuple: $Op : \langle Type = \{BT, R, W, A, C\}, arg : Data item, val : Value, tid : Transaction identifier, res : Result \rangle$. The meaning of the components is as follows: 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 also used in case of Read and Write operations to specify the value that has been read or the value to be written for data item arg (otherwise 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. In the high level descriptions of the algorithms in this chapter, res may seem unnecessary, but we will see in Chapter 12 that these return codes will be important.

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. Furthermore, the system may be less reliable since the failure or inaccessibility of the central site would cause major system failures. There are studies that indicate that the bottleneck will indeed form as the transaction rate increases.

11.3.2 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 Figure 11.10.

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 Figure 11.10 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 Chapter 12). Owing to these similarities, we do not

Algorithm 11.1: Centralized 2PL Transaction Manager (C2PL-TM) Algorithm

Input: msg : a message

begin

repeat

wait for a msg ;

switch msg **do**

- case** *transaction operation*
 - let op be the operation ;
 - if** $op.Type = BT$ **then** $DP(op)$ {call DP with operation}
 - else** $C2PL-LM(op)$ {call LM with operation}
- case** *Lock Manager response* {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
- case** *Data Processor response* {operation completed message}
 - switch** *transaction operation* **do**
 - let op be the operation ;
 - case** *R*
 - return $op.val$ (data item value) to the application
 - case** *W*
 - inform application of completion of the write
 - case** *C*
 - 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
 - case** *A*
 - inform application of completion of the abort ;
 - $C2PL-LM(op)$ {need to release locks}

until *forever* ;

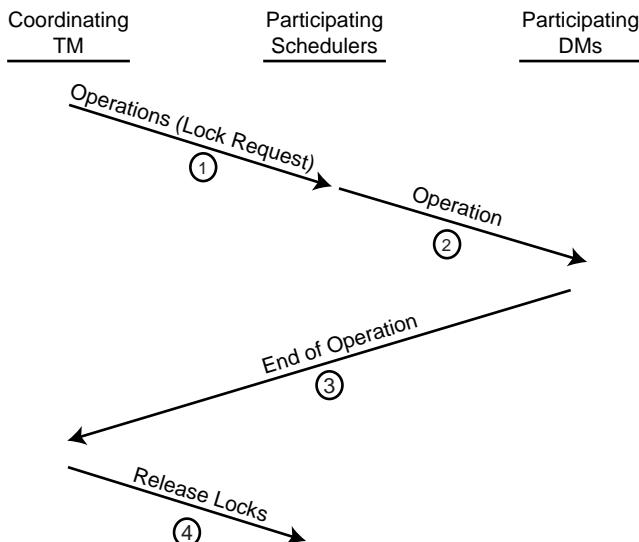
end

Algorithm 11.2: Centralized 2PL Lock Manager (C2PL-LM) Algorithm

```

Input:  $op : Op$ 
begin
  switch  $op.Type$  do
    case  $R$  or  $W$  {lock request; see if it can be granted}
      find the lock unit  $lu$  such that  $op.arg \subseteq lu$  ;
      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  $lu$ 
    case  $C$  or  $A$  {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
           $O.tid$ 
      send “Locks released” to coordinating TM of transaction
  end

```

**Fig. 11.10** Communication Structure of Distributed 2PL

Algorithm 11.3: Data Processor (DP) Algorithm

Input: $op : Op$

begin

```

switch  $op.Type$  do           {check the type of operation}
    case  $BT$                  {details to be discussed in Chapter 12}
         $\downarrow$  do some bookkeeping
    case  $R$                   {database READ operation}
         $\downarrow$   $op.res \leftarrow \text{READ}(op.arg);$ 
         $\downarrow$   $op.res \leftarrow \text{"Read done"}$ 
    case  $W$                   {database WRITE of  $val$  into data item  $arg$ }
         $\downarrow$   $\text{WRITE}(op.arg, op.val);$ 
         $\downarrow$   $op.res \leftarrow \text{"Write done"}$ 
    case  $C$                   {execute COMMIT }
         $\downarrow$   $\text{COMMIT};$ 
         $\downarrow$   $op.res \leftarrow \text{"Commit done"}$ 
    case  $A$                   {execute ABORT }
         $\downarrow$   $\text{ABORT};$ 
         $\downarrow$   $op.res \leftarrow \text{"Abort done"}$ 
return  $op$ 

```

end

give the distributed TM and LM algorithms here. Distributed 2PL algorithms have been used in System R* [Mohan et al., 1986] and in NonStop SQL ([Tandem, 1987, 1988] and [Borr, 1988]).

11.4 Timestamp-Based Concurrency Control Algorithms

Unlike locking-based algorithms, timestamp-based concurrency control algorithms do not attempt to maintain serializability by mutual exclusion. Instead, they 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.

A timestamp is a simple identifier that serves to identify each transaction uniquely and is used for ordering. *Uniqueness* is only one of the properties of timestamp generation. The second property is *monotonicity*. Two timestamps generated by the same transaction manager should be monotonically increasing. Thus timestamps are values derived from a totally ordered domain. It is this second property that differentiates a timestamp from a transaction identifier.

There are a number of ways that timestamps can be assigned. 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 $\langle \text{local counter value}, \text{site identifier} \rangle$. 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.

With this information, it is simple to order the execution of the transactions' operations according to their timestamps. Formally, the timestamp ordering (TO) rule can be specified as follows:

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.

A timestamp ordering scheduler that operates in this fashion is guaranteed to generate serializable histories. However, this comparison between the transaction timestamps can be performed only if the scheduler has received all the operations to be scheduled. If operations come to the scheduler one at a time (which is the realistic case), it is necessary to be able to detect, in an efficient manner, if an operation has arrived out of sequence. To facilitate this check, 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.

Architecturally (see Figure 10.5), 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.

11.4.1 Basic TO Algorithm

The basic TO algorithm is a straightforward implementation of the TO rule. The coordinating transaction manager assigns the timestamp to each transaction, deter-

mines the sites where each data item is stored, and sends the relevant operations to these sites. The basic TO transaction manager algorithm (BTO-TM) is depicted in Algorithm 11.4. The histories at each site simply enforce the TO rule. The scheduler algorithm is given in Algorithm 11.5. The data manager is still the one given in Algorithm 11.3. The same data structures and assumptions we used for centralized 2PL algorithms apply to these algorithms as well.

As indicated before, a transaction one of whose operations is rejected by a scheduler 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.

Example 11.8. 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 11.5.

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).