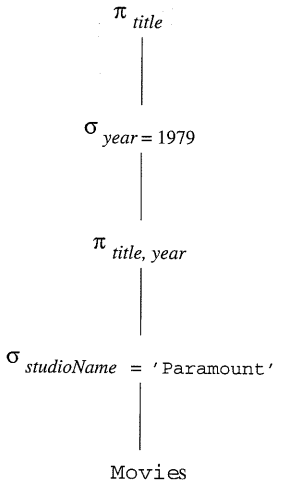

Figure 16.9: Expressing the query in terms of base tables

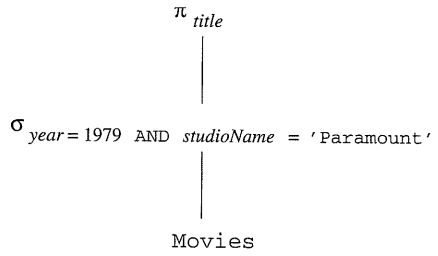

Figure 16.10: Simplifying the query over base tables

16.1.5 Exercises for Section 16.1

Exercise 16.1.1: Add to or modify the rules for <Query> to include simple versions of the following features of SQL select-from-where expressions:

a) A query with no where-clause.

b) The ability to produce a set with the `DISTINCT` keyword.

c) A `GROUP BY` clause and a `HAVING` clause.

d) Sorted output with the `ORDER BY` clause.

Exercise 16.1.2: Add to the rules for <Condition> to allow the following features of SQL conditionals:

a) Comparisons other than =.

b) Parenthesized conditions.

c) EXISTS expressions.

d) Logical operators OR and NOT.

Exercise 16.1.3: Using the simple SQL grammar exhibited in this section, give parse trees for the following queries about relations $R(a, b)$ and $S(b, c)$:

```
a)  SELECT a FROM R WHERE b IN
        (SELECT a FROM R, S WHERE R.b = S.b);

b)  SELECT a, c FROM R, S WHERE R.b = S.b;
```

16.2 Algebraic Laws for Improving Query Plans

We resume our discussion of the query compiler in Section 16.3, where we shall transform the parse tree into an expression of the extended relational algebra. Also in Section 16.3, we shall see how to apply heuristics that we hope will improve the algebraic expression of the query, using some of the many algebraic laws that hold for relational algebra. As a preliminary, this section catalogs algebraic laws that turn one expression tree into an equivalent expression tree that may have a more efficient physical query plan. The result of applying these algebraic transformations is the logical query plan that is the output of the query-rewrite phase.

16.2.1 Commutative and Associative Laws

A *commutative law* about an operator says that it does not matter in which order you present the arguments of the operator; the result will be the same. For instance, $+$ and $\times$ are commutative operators of arithmetic. More precisely, $x + y = y + x$ and $x \times y = y \times x$ for any numbers x and y. On the other hand, $-$ is not a commutative arithmetic operator: $x - y \neq y - x$.

An *associative law* about an operator says that we may group two uses of the operator either from the left or the right. For instance, $+$ and $\times$ are associative arithmetic operators, meaning that $(x + y) + z = x + (y + z)$ and $(x \times y) \times z = x \times (y \times z)$. On the other hand, $-$ is not associative: $(x - y) - z \neq x - (y - z)$. When an operator is both associative and commutative, then any number of operands connected by this operator can be grouped and ordered as we wish without changing the result. For example, $\big((w + x) + y\big) + z = (y + x) + (z + w)$.

Several of the operators of relational algebra are both associative and commutative. Particularly:

- $R \times S = S \times R$; $(R \times S) \times T = R \times (S \times T)$.
- $R \bowtie S = S \bowtie R$; $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$.
- $R \cup S = S \cup R$; $(R \cup S) \cup T = R \cup (S \cup T)$.

16.2.8 Exercises for Section 16.2

Exercise 16.2.1: Give examples to show that:

a) Duplicate elimination (δ) cannot be pushed below projection.

b) Duplicate elimination cannot be pushed below bag union or difference.

c) Projection cannot be pushed below set union.

d) Projection cannot be pushed below set or bag difference.

! **Exercise 16.2.2:** Prove that we can always push a projection below both branches of a bag union.

! **Exercise 16.2.3:** Some laws that hold for sets hold for bags; others do not. For each of the laws below that are true for sets, tell whether or not it is true for bags. Either give a proof the law for bags is true, or give a counterexample.

a) $R - R = \emptyset$.

b) $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$ (distribution of union over intersection).

c) $R \cup R = R$ (the idempotent law for union).

d) $R \cap R = R$ (the idempotent law for intersection).

! **Exercise 16.2.4:** We can define $\subseteq$ for bags by: $R \subseteq S$ if and only if for every element x, the number of times x appears in R is less than or equal to the number of times it appears in S. Tell whether the following statements (which are all true for sets) are true for bags; give either a proof or a counterexample:

a) If $R \subseteq S$ and $S \subseteq R$, then $R = S$.

b) If $R \subseteq S$, then $R \cup S = S$.

c) If $R \subseteq S$, then $R \cap S = R$.

Exercise 16.2.5: Starting with an expression $\pi_L\big(R(a,b,c) \bowtie S(b,c,d,e)\big)$, push the projection down as far as it can go if L is:

a) $a,\ b,\ a+d \to z$.

b) $b+c \to x,\ c+d \to y$.

Exercise 16.2.6: When it is possible to push a selection to both arguments of a binary operator, we need to decide whether or not to do so. How would the existence of indexes on one of the arguments affect our choice? Consider, for instance, an expression $\sigma_C(R \cap S)$, where there is an index on S.

! **Exercise 16.2.7:** The following are possible equalities involving operations on a relation $R(a, b)$. Tell whether or not they are true; give either a proof or a counterexample.

a) $\gamma_{MIN(a)\to y,\ x}\big(\gamma_{a,\ SUM(b)\to x}(R)\big) = \gamma_{y,SUM(b)\to x}\big(\gamma_{MIN(a)\to y,\ b}(R)\big)$.

b) $\gamma_{MIN(a)\to y,\ x}\big(\gamma_{a,\ MAX(b)\to x}(R)\big) = \gamma_{y,MAX(b)\to x}\big(\gamma_{MIN(a)\to y,\ b}(R)\big)$.

!! **Exercise 16.2.8:** The join-like operators of Exercise 15.2.3 obey some of the familiar laws, and others do not. Tell whether each of the following is or is not true. Give either a proof that the law holds or a counterexample.

a) $R \ltimes S = S \ltimes R$.

b) $\sigma_C(R \ltimes S) = \sigma_C(R) \ltimes S$.

c) $\sigma_C(R \overset{\circ}{\bowtie} S) = \sigma_C(R) \overset{\circ}{\bowtie} S$.

d) $\sigma_C(R \overset{\circ}{\bowtie}_L S) = \sigma_C(R) \overset{\circ}{\bowtie}_L S$, where C involves only attributes of R.

e) $\sigma_C(R \overset{\circ}{\bowtie}_L S) = R \overset{\circ}{\bowtie}_L \sigma_C(S)$, where C involves only attributes of S.

f) $\pi_L(R \mathbin{\overline{\ltimes}} S) = \pi_L(R) \mathbin{\overline{\ltimes}} S$.

g) $(R \overset{\circ}{\bowtie} S) \overset{\circ}{\bowtie} T = R \overset{\circ}{\bowtie} (S \overset{\circ}{\bowtie} T)$.

h) $R \overset{\circ}{\bowtie} S = S \overset{\circ}{\bowtie} R$.

i) $R \overset{\circ}{\bowtie}_L S = S \overset{\circ}{\bowtie}_L R$.

!! **Exercise 16.2.9:** While it is not precisely an algebraic law, because it involves an indeterminate number of operands, it is generally true that

$$\texttt{SUM}(a_1, a_2, \ldots, a_n) = a_1 + a_2 + \cdots + a_n$$

SQL has both a `SUM` operator and addition for integers and reals. Considering the possibility that one or more of the a_i's could be `NULL`, rather than an integer or real, does this "law" hold in SQL?

! **Exercise 16.2.10:** We mentioned in Example 16.14 that none of the plans we showed is necessarily the best plan. Can you think of a better plan?

16.3 From Parse Trees to Logical Query Plans

We now resume our discussion of the query compiler. Having constructed a parse tree for a query in Section 16.1, we next need to turn the parse tree into the preferred logical query plan. There are two steps, as was suggested in Fig. 16.1.

2. We must add a projection to eliminate duplicate copies of attributes involved in a natural join that has become a theta-join.

3. The theta-join conditions must be associative. Recall there are cases, as discussed in Section 16.2.1, where theta-joins are not associative.

In addition, products can be considered as a special case of natural join and combined with joins if they are adjacent in the tree. Figure 16.25 illustrates this transformation in a situation where the logical query plan has a cluster of two union operators and a cluster of three natural join operators. Note that the letters R through W stand for any expressions, not necessarily for stored relations.

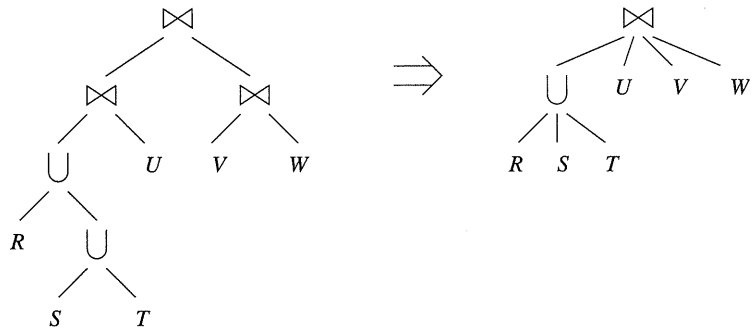

Figure 16.25: Final step in producing the logical query plan: group the associative and commutative operators

16.3.5 Exercises for Section 16.3

Exercise 16.3.1: Convert to relational algebra your parse trees from Exercise 16.1.3(a) and (b). For (a), show both the form with a two-argument selection and its eventual conversion to a one-argument (conventional σ_C) selection.

Exercise 16.3.2: Replace the natural joins in the following expressions by equivalent theta-joins and projections. Tell whether the resulting theta-joins form a commutative and associative group.

a) $\big(R(a,b) \bowtie S(b,c)\big) \bowtie \big(T(c,d) \bowtie U(d,e)\big)$.

b) $\big(R(a,b) \bowtie S(b,c)\big) \bowtie \big(T(c,d) \bowtie U(a,d)\big)$.

c) $\big(R(a,b) \bowtie S(b,c)\big) \bowtie_{R.a<T.c} T(c,d)$.

! **Exercise 16.3.3:** Give a rule for converting each of the following forms of <Condition> to relational algebra. All conditions may be assumed to be applied (by a two-argument selection) to a relation R. You may assume that the

subquery is not correlated with R. Be careful that you do not introduce or eliminate duplicates in opposition to the formal definition of SQL.

a) A condition of the form a = ANY <Query>, where a is an attribute of R.

b) A condition of the form a = ALL <Query>, where a is an attribute of R.

c) A condition of the form EXISTS(<Query>).

!! **Exercise 16.3.4:** Repeat Exercise 16.3.3, but allow the subquery to be correlated with R. For simplicity, you may assume that the subquery has the simple form of select-from-where expression described in this section, with *no* further subqueries.

!! **Exercise 16.3.5:** From how many different expression trees could the grouped tree on the right of Fig. 16.25 have come? Remember that the order of children after grouping is not necessarily reflective of the ordering in the original expression tree.

16.4 Estimating the Cost of Operations

Having parsed a query and transformed it into a logical query plan, we must next turn the logical plan into a physical plan. We normally do so by considering many different physical plans that are derived from the logical plan, and evaluating or estimating the cost of each. After this evaluation, often called *cost-based enumeration*, we pick the physical query plan with the least estimated cost; that plan is the one passed to the query-execution engine. When enumerating possible physical plans derivable from a given logical plan, we select for each physical plan:

1. An order and grouping for associative-and-commutative operations like joins, unions, and intersections.
2. An algorithm for each operator in the logical plan, for instance, deciding whether a nested-loop join or a hash-join should be used.
3. Additional operators — scanning, sorting, and so on — that are needed for the physical plan but that were not present explicitly in the logical plan.
4. The way in which arguments are passed from one operator to the next, for instance, by storing the intermediate result on disk or by using iterators and passing an argument one tuple or one main-memory buffer at a time.

To make each of these choices, we need to understand what the costs of the various physical plans are. We cannot know these costs exactly without executing the plan. But almost always, the cost of executing a query plan is

Duplicate Elimination

If $R(a_1, a_2, \ldots, a_n)$ is a relation, then $V(R, [a_1, a_2, \ldots, a_n])$ is the size of $\delta(R)$. However, often we shall not have this statistic available, so it must be approximated. In the extremes, the size of $\delta(R)$ could be the same as the size of R (no duplicates) or as small as 1 (all tuples in R are the same).[4] Another upper limit on the number of tuples in $\delta(R)$ is the maximum number of distinct tuples that could exist: the product of $V(R, a_i)$ for $i = 1, 2, \ldots, n$. That number could be smaller than other estimates of $T\big(\delta(R)\big)$. There are several rules that could be used to estimate $T\big(\delta(R)\big)$. One reasonable one is to take the smaller of $T(R)/2$ and the product of all the $V(R, a_i)$'s.

Grouping and Aggregation

Suppose we have an expression $\gamma_L(R)$, the size of whose result we need to estimate. If the statistic $V(R, [g_1, g_2, \ldots, g_k])$, where the g_i's are the grouping attributes in L, is available, then that is our answer. However, that statistic may well not be obtainable, so we need another way to estimate the size of $\gamma_L(R)$. The number of tuples in $\gamma_L(R)$ is the same as the number of groups. There could be as few as one group in the result or as many groups as there are tuples in R. As with δ, we can also upper-bound the number of groups by a product of $V(R, A)$'s, but here attribute A ranges over only the grouping attributes of L. We again suggest an estimate that is the smaller of $T(R)/2$ and this product.

16.4.8 Exercises for Section 16.4

Exercise 16.4.1: Below are the vital statistics for four relations, W, X, Y, and Z:

$W(a,b)$	$X(b,c)$	$Y(c,d)$	$Z(d,e)$
$T(W) = 400$	$T(X) = 300$	$T(Y) = 200$	$T(Z) = 100$
$V(W,a) = 50$	$V(X,b) = 60$	$V(Y,c) = 50$	$V(Z,d) = 10$
$V(W,b) = 40$	$V(X,c) = 100$	$V(Y,d) = 20$	$V(Z,e) = 50$

Estimate the sizes of relations that are the results of the following expressions:

(a)	$W \bowtie X \bowtie Y \bowtie Z$	(b)	$\sigma_{a=10}(W)$	(c)	$\sigma_{c=20}(Y)$
(d)	$\sigma_{c=20}(Y) \bowtie Z$	(e)	$W \times Y$	(f)	$\sigma_{d>10}(Z)$
(g)	$\sigma_{a=1 \text{ AND } b=2}(W)$	(h)	$\sigma_{a=1 \text{ AND } b>2}(W)$	(i)	$X \bowtie_{X.c<Y.c} Y$

Exercise 16.4.2: Here are the statistics for four relations E, F, G, and H:

[4] Strictly speaking, if R is empty there are no tuples in either R or $\delta(R)$, so the lower bound is 0. However, we are rarely interested in this special case.

$E(a,b,c)$	$F(a,b,d)$	$G(a,c,d)$	$H(b,c,d)$
$T(E) = 1000$	$T(F) = 2000$	$T(G) = 3000$	$T(H) = 4000$
$V(E,a) = 500$	$V(F,a) = 50$	$V(G,a) = 500$	$V(H,b) = 400$
$V(E,b) = 100$	$V(F,b) = 200$	$V(G,c) = 300$	$V(H,c) = 200$
$V(E,c) = 20$	$V(F,d) = 100$	$V(G,d) = 100$	$V(H,d) = 800$

How many tuples does the join of these tuples have, using the techniques for estimation from this section?

!! **Exercise 16.4.3:** Suppose we compute $R(a,b) \bowtie S(a,c)$, where R and S each have 1000 tuples. The a attribute of each relation has 100 different values, and they are the *same* 100 values. If the distribution of values was uniform; i.e., each a-value appeared in exactly 10 tuples of each relation, then there would be 10,000 tuples in the join. Suppose instead that the 100 a-values have the same Zipfian distribution in each relation. Precisely, let the values be $a_1, a_2, \ldots, a_{100}$. Then the number of tuples of both R and S that have a-value a_i is proportional to $1/\sqrt{i}$. Under these circumstances, how many tuples does the join have? You should ignore the fact that the number of tuples with a given a-value may not be an integer.

! **Exercise 16.4.4:** How would you estimate the size of a semijoin?

16.5 Introduction to Cost-Based Plan Selection

Whether selecting a logical query plan or constructing a physical query plan from a logical plan, the query optimizer needs to estimate the cost of evaluating certain expressions. We study the issues involved in cost-based plan selection here, and in Section 16.6 we consider in detail one of the most important and difficult problems in cost-based plan selection: the selection of a join order for several relations.

As before, we shall assume that the "cost" of evaluating an expression is approximated well by the number of disk I/O's performed. The number of disk I/O's, in turn, is influenced by:

1. The particular logical operators chosen to implement the query, a matter decided when we choose the logical query plan.
2. The sizes of intermediate results, whose estimation we discussed in Section 16.4.
3. The physical operators used to implement logical operators, e.g., the choice of a one-pass or two-pass join, or the choice to sort or not sort a given relation; this matter is discussed in Section 16.7.
4. The ordering of similar operations, especially joins as discussed in Section 16.6.

16.5.5 Exercises for Section 16.5

Exercise 16.5.1: Estimate the size of the join $R(a,b) \bowtie S(b,c)$ using histograms for $R.b$ and $S.b$. Assume $V(R,b) = V(S,b) = 20$, and the histograms for both attributes give the frequency of the four most common values, as tabulated below:

	0	1	2	3	4	others
$R.b$	5	4	10	5		36
$S.b$	10	8	5		7	50

How does this estimate compare with the simpler estimate, assuming that all 20 values are equally likely to occur, with $T(R) = 60$ and $T(S) = 80$?

Exercise 16.5.2: Estimate the size of the join $R(a,b) \bowtie S(b,c)$ if we have the following histogram information:

	$b < 0$	$b = 0$	$b > 0$
R	400	100	200
S	400	300	800

! **Exercise 16.5.3:** In Example 16.29 we suggested that reducing the number of values that either attribute named b had could make plan (a) better than plan (b) of Fig. 16.29. For what values of:

a) $V(S,b)$

b) $V(R,b)$

will plan (a) have a lower estimated cost than plan (b)?

! **Exercise 16.5.4:** Consider four relations R, S, T, and V. Respectively, they have 100, 200, 300, and 400 tuples, chosen randomly and independently from the same pool of 1000 tuples (e.g., the probabilities of a given tuple being in R is 1/10, in S is 1/5, and in both is 1/50).

a) What is the expected size of $R \cap S \cap T \cap V$?

b) What order of intersections gives the least cost (estimated sum of the sizes of the intermediate relations)?

c) What is the expected size of $R \cup S \cup T \cup V$?

d) What order of unions gives the least cost (estimated sum of the sizes of the intermediate relations)?

! **Exercise 16.5.5:** Repeat Exercise 16.5.4 if all four relations have 250 of the 1000 tuples, at random.

!! **Exercise 16.5.6:** Suppose we wish to compute the expression

$$\tau_b\big(R(a,b) \bowtie S(b,c) \bowtie T(d,a)\big)$$

That is, we join the three relations and produce the result sorted on attribute b. Let us make the simplifying assumptions:

i. We shall not "join" R and T first, because that is a product.

ii. Any other join can be performed with a two-pass sort-join or hash-join, but in no other way.

iii. Any relation, or the result of any expression, can be sorted by a two-phase, multiway merge-sort, but in no other way.

iv. The result of the first join will be passed as an argument to the last join one block at a time and not stored temporarily on disk.

v. Each relation occupies 1000 blocks, and the result of either join of two relations occupies 5000 blocks.

Answer the following based on these assumptions:

a) What are all the subexpressions and orders that a Selinger-style optimization would consider?

b) Which query plan uses the fewest disk I/O's?[7]

!! **Exercise 16.5.7:** Give an example of a logical query plan of the form $E \bowtie F$, for some expressions E and F (which you may choose), where using the best plans to evaluate E and F does not allow any choice of algorithm for the final join that minimizes the total cost of evaluating the entire expression. Make whatever assumptions you wish about the number of available main-memory buffers and the sizes of relations mentioned in E and F.

16.6 Choosing an Order for Joins

In this section we focus on a critical problem in cost-based optimization: selecting an order for the (natural) join of three or more relations. Similar ideas can be applied to other binary operations like union or intersection, but these operations are less important in practice, because they typically take less time to execute than joins, and they more rarely appear in clusters of three or more.

[7] Notice that, because we have made some very specific assumptions about the join methods to be used, we can estimate disk I/O's, instead of relying on the simpler, but less accurate, counts of tuples as our cost measure.

Join Selectivity

A useful way to view heuristics such as the greedy algorithm for selecting a left-deep join tree is that each relation R, when joined with the current tree, has a *selectivity*, which is the ratio of the size of the join result to size of the current tree's result. Since we usually do not have the exact sizes of either relation, we estimate these sizes as we have done previously. A greedy approach to join ordering is to pick that relation with the smallest selectivity.

For example, if a join attribute is a key for R, then the selectivity is at most 1, which is usually a favorable situation. Notice that, judging from the statistics of Fig. 16.31, attribute d is a key for U, and there are no keys for other relations, which suggests why joining T with U is the best way to start the join.

latter, with a size of 2000 is better than the former, with a size of 10,000. Thus, we pick as the new current tree $(T \bowtie U) \bowtie S$.

Now there is no choice; we must join R at the last step, leaving us with a total cost of 3000, the sum of the sizes of the two intermediate relations. Note that the tree resulting from the greedy algorithm is the same as that selected by the dynamic-programming algorithm in Example 16.33. However, there are examples where the greedy algorithm fails to find the best solution, while the dynamic-programming algorithm guarantees to find the best; see Exercise 16.6.4. □

16.6.7 Exercises for Section 16.6

Exercise 16.6.1: For the relations of Exercise 16.4.1, give the dynamic-programming table entries that evaluates all possible join orders allowing: a) Left-deep trees only. b) All trees What is the best choice in each case?

Exercise 16.6.2: Repeat Exercise 16.6.1 with the following modifications:

i. The schema for Z is changed to $Z(d, a)$.

ii. $V(Z, a) = 50$.

Exercise 16.6.3: Repeat Exercise 16.6.1 with the relations of Exercise 16.4.2.

Exercise 16.6.4: Consider the join of relations $R(a, b)$, $S(b, c)$, $T(c, d)$, and $U(a, d)$, where R and U each have 1000 tuples, while S and T each have 200 tuples. Further, there are 200 values of all attributes of all relations, except for attribute c, where $V(S, c) = V(T, c) = 20$.

a) What is the order selected by the greedy algorithm? What is its cost?

b) What is the optimum join ordering and its cost?

! **Exercise 16.6.5:** Suppose we wish to join the relations R, S, T, and U in one of the tree structures of Fig. 16.30, and we want to keep all intermediate relations in memory until they are no longer needed. Following our usual assumption, the result of the join of all four will be consumed by some other process as it is generated, so no memory is needed for that relation. In terms of the number of blocks required for the stored relations and the intermediate relations [e.g., $B(R)$ or $B(R \bowtie S)$], give a lower bound on M, the number of blocks of memory needed, for each of the trees in Fig. 16.30? What assumptions let us conclude that one tree is certain to use less memory than another?

! **Exercise 16.6.6:** If we use dynamic programming to select an order for the join of k relations, how many entries of the table do we have to fill?

Exercise 16.6.7: How many trees are there for the join of (a) eight (b) nine relations? How many of these are neither left-deep nor right-deep?

16.7 Completing the Physical-Query-Plan

We have parsed the query, converted it to an initial logical query plan, and improved that logical query plan with transformations described in Section 16.3. Part of the process of selecting the physical query plan is enumeration and cost-estimation for all of our options, which we discussed in Section 16.5. Section 16.6 focused on the question of enumeration, cost estimation, and ordering for joins of several relations. By extension, we can use similar techniques to order groups of unions, intersections, or any associative/commutative operation.

There are still several steps needed to turn the logical plan into a complete physical query plan. The principal issues that we must yet cover are:

1. Selection of algorithms to implement the operations of the query plan, when algorithm-selection was not done as part of some earlier step such as selection of a join order by dynamic programming.

2. Decisions regarding when intermediate results will be *materialized* (created whole and stored on disk), and when they will be *pipelined* (created only in main memory, and not necessarily kept in their entirety at any one time).

3. Notation for physical-query-plan operators, which must include details regarding access methods for stored relations and algorithms for implementation of relational-algebra operators.

We shall not discuss the subject of selection of algorithms for operators in its entirety. Rather, we sample the issues by discussing two of the most important operators: selection in Section 16.7.1 and joins in Section 16.7.2.

3. Execute all nodes of each subtree using a network of iterators. Thus, all the nodes in one subtree are executed simultaneously, with `GetNext` calls among their operators determining the exact order of events.

Following this strategy, the query optimizer can now generate executable code, perhaps a sequence of function calls, for the query.

16.7.8 Exercises for Section 16.7

Exercise 16.7.1: Consider a relation $R(a, b, c, d)$ that has a clustering index on a and nonclustering indexes on each of the other attributes. The relevant parameters are: $B(R) = 500$, $T(R) = 5000$, $V(R, a) = 50$, $V(R, b) = 1000$, $V(R, c) = 5000$, and $V(R, d) = 500$. Give the best query plan (index-scan or table-scan followed by a filter step) and the disk-I/O cost for each of the following selections:

a) $\sigma_{a=1 \text{ AND } b=2 \text{ AND } c\geq 3}(R)$.

b) $\sigma_{a=1 \text{ AND } b\leq 2 \text{ AND } c\geq 3}(R)$.

c) $\sigma_{a=1 \text{ AND } b=2 \text{ AND } d=3}(R)$.

Exercise 16.7.2: How would the conclusions about when to pipeline in Example 16.36 change if the size of relation R were not 5000 blocks, but: (a) 1000 blocks ! (b) 100 blocks ! (c) 10,000 blocks?

! **Exercise 16.7.3:** In terms of $B(R)$, $T(R)$, $V(R, x)$, and $V(R, y)$, express the following conditions about the cost of implementing a selection on R:

a) It is better to use index-scan with a nonclustering index on x and a term that equates x to a constant than a clustering index on y and a term that equates y to a constant.

b) It is better to use index-scan with a nonclustering index on x and a term that equates x to a constant than a clustering index on y and a term of the form $y > C$ for some constant C.

c) It is better to use index-scan with a nonclustering index on x and a term that equates x to a constant than a nonclustering index on y and a term that equates y to a constant.

! **Exercise 16.7.4:** Suppose we want to compute $\big(R(a, b) \bowtie S(a, c)\big) \bowtie T(a, d)$ in the order indicated. We have $M = 101$ main-memory buffers, and $B(R) = B(S) = 2000$. Because the join attribute a is the same for both joins, we decide to implement the first join $R \bowtie S$ by a two-pass sort-join, and we shall use the appropriate number of passes for the second join, first dividing T into some number of sublists sorted on a, and merging them with the sorted and pipelined stream of tuples from the join $R \bowtie S$. For what values of $B(T)$ should we choose for the join of T with $R \bowtie S$:

a) A one-pass join; i.e., we read T into memory, and compare its tuples with the tuples of $R \bowtie S$ as they are generated.

b) A two-pass join; i.e., we create sorted sublists for T and keep one buffer in memory for each sorted sublist, while we generate tuples of $R \bowtie S$.

16.8 Summary of Chapter 16

- *Compilation of Queries*: Compilation turns a query into a physical query plan, which is a sequence of operations that can be implemented by the query-execution engine. The principal steps of query compilation are parsing, semantic checking, selection of the preferred logical query plan (algebraic expression), and generation from that of the best physical plan.
- *The Parser*: The first step in processing a SQL query is to parse it, as one would for code in any programming language. The result of parsing is a parse tree with nodes corresponding to SQL constructs.
- *View Expansion*: Queries that refer to virtual views must have these references in the parse tree replaced by the tree for the expression that defines the view. This expansion often introduces several opportunities to optimize the complete query.
- *Semantic Checking*: A preprocessor examines the parse tree, checks that the attributes, relation names, and types make sense, and resolves attribute references.
- *Conversion to a Logical Query Plan*: The query processor must convert the semantically checked parse tree to an algebraic expression. Much of the conversion to relational algebra is straightforward, but subqueries present a problem. One approach is to introduce a two-argument selection that puts the subquery in the condition of the selection, and then apply appropriate transformations for the common special cases.
- *Algebraic Transformations*: There are many ways that a logical query plan can be transformed to a better plan by using algebraic transformations. Section 16.2 enumerates the principal ones.
- *Choosing a Logical Query Plan*: The query processor must select that query plan that is most likely to lead to an efficient physical plan. In addition to applying algebraic transformations, it is useful to group associative and commutative operators, especially joins, so the physical query plan can choose the best order and grouping for these operations.
- *Estimating Sizes of Relations*: When selecting the best logical plan, or when ordering joins or other associative-commutative operations, we use the estimated size of intermediate relations as a surrogate for the true