574 Chapter 12 Query Processing

- Sequential I/O
- Random I/O
- File scan
- Linear search
- Selections using indices
- Access paths
- Index scans
- Conjunctive selection
- Disjunctive selection
- Composite index
- Intersection of identifiers
- External sorting
- External sort–merge
- Runs
- N-way merge
- Equi-join
- Nested-loop join
- Block nested-loop join
- Indexed nested-loop join
- Merge join
- Sort-merge join
- Hybrid merge join
- Hash join
- Build
- Probe
- Build input
- Probe input
- Recursive partitioning
- Hash-table overflow
- Skew
- Fudge factor
- Overflow resolution
- Overflow avoidance
- Hybrid hash join
- Operator tree
- Materialized evaluation
- Double buffering
- Pipelined evaluation
  - Demand-driven pipeline (lazy, pulling)
  - Producer-driven pipeline (eager, pushing)
- Iterator
- Double-pipelined join

Practice Exercises

12.1 Assume (for simplicity in this exercise) that only one tuple fits in a block and memory holds at most 3 blocks. Show the runs created on each pass of the sort-merge algorithm, when applied to sort the following tuples on the first attribute: (kangaroo, 17), (wallaby, 21), (emu, 1), (wombat, 13), (platypus, 3), (lion, 8), (warthog, 4), (zebra, 11), (meerkat, 6), (hyena, 9), (hornbill, 2), (baboon, 12).

12.2 Consider the bank database of Figure 12.13, where the primary keys are underlined, and the following SQL query:

```sql
select T.branch_name
from branch T, branch S
where T.assets > S.assets and S.branch_city = "Brooklyn"
```
Write an efficient relational-algebra expression that is equivalent to this query. Justify your choice.

12.3 Let relations $r_1(A, B, C)$ and $r_2(C, D, E)$ have the following properties: $r_1$ has 20,000 tuples, $r_2$ has 45,000 tuples, 25 tuples of $r_1$ fit on one block, and 30 tuples of $r_2$ fit on one block. Estimate the number of block transfers and seeks required, using each of the following join strategies for $r_1 \bowtie r_2$:

a. Nested-loop join.
b. Block nested-loop join.
c. Merge join.
d. Hash join.

12.4 The indexed nested-loop join algorithm described in Section 12.5.3 can be inefficient if the index is a secondary index, and there are multiple tuples with the same value for the join attributes. Why is it inefficient? Describe a way, using sorting, to reduce the cost of retrieving tuples of the inner relation. Under what conditions would this algorithm be more efficient than hybrid merge join?

12.5 Let $r$ and $s$ be relations with no indices, and assume that the relations are not sorted. Assuming infinite memory, what is the lowest-cost way (in terms of I/O operations) to compute $r \bowtie s$? What is the amount of memory required for this algorithm?

12.6 Consider the bank database of Figure 12.13, where the primary keys are underlined. Suppose that a B+ -tree index on branch.city is available on relation branch, and that no other index is available. List different ways to handle the following selections that involve negation:

a. $\sigma_{\text{branch.city} \neq \text{"Brooklyn"}}(\text{branch})$
b. $\sigma_{\text{branch.city} = \text{"Brooklyn"}}(\text{branch})$
c. $\sigma_{\text{branch.city} < \text{"Brooklyn"} \lor \text{assets} < 5000}(\text{branch})$

12.7 Write pseudocode for an iterator that implements indexed nested-loop join, where the outer relation is pipelined. Your pseudocode must define

```plaintext
branch(branch.name, branch.city, assets)
customer(customer.name, customer.street, customer.city)
loan(loan.number, branch.name, amount)
borrower(customer.name, loan.number)
account(account.number, branch.name, balance)
depositor(customer.name, account.number)
```

Figure 12.13 Banking database.
the standard iterator functions `open()`, `next()`, and `close()`. Show what state information the iterator must maintain between calls.

12.8 Design sort-based and hash-based algorithms for computing the relational division operation (see Practise Exercises of Chapter 6 for a definition of the division operation).

12.9 What is the effect on the cost of merging runs if the number of buffer blocks per run is increased, while keeping overall memory available for buffering runs fixed?

**Exercises**

12.10 Suppose you need to sort a relation of 40 gigabytes, with 4 kilobyte blocks, using a memory size of 40 megabytes. Suppose the cost of a seek is 5 milliseconds, while the disk transfer rate is 40 megabytes per second.

   a. Find the cost of sorting the relation, in seconds, with $b_b = 1$ and with $b_b = 100$.

   b. In each case, how many merge passes are required?

   c. Suppose a flash storage device is used instead of a disk, and it has a seek time of 1 microsecond, and a transfer rate of 40 megabytes per second. Recompute the cost of sorting the relation, in seconds, with $b_b = 1$ and with $b_b = 100$, in this setting.

12.11 Consider the following extended relational-algebra operators. Describe how to implement each operation using sorting, and using hashing.

   a. **Semijoin** $(\bowtie_{\theta})$: $r \bowtie_{\theta} s$ is defined as $\Pi_R(r \bowtie_{\theta} s)$, where $R$ is the set of attributes in the schema of $r$; that it it selects those tuples $r_i$ in $r$ for which there is a tuple $s_j$ in $s$ such that $r_i$ and $s_j$ satisfy predicate $\theta$.

   b. **Anti-semijoin** $(\bar{\bowtie}_{\theta})$: $r \bar{\bowtie}_{\theta} s$ is defined as $r - \Pi_R(r \bowtie_{\theta} s)$; that it it selects those tuples $r_i$ in $r$ for which there is no tuple $s_j$ in $s$ such that $r_i$ and $s_j$ satisfy predicate $\theta$.

12.12 Why is it not desirable to force users to make an explicit choice of a query-processing strategy? Are there cases in which it is desirable for users to be aware of the costs of competing query-processing strategies? Explain your answer.

12.13 Design a variant of the hybrid merge-join algorithm for the case where both relations are not physically sorted, but both have a sorted secondary index on the join attributes.

12.14 Estimate the number of block transfers and seeks required by your solution to Exercise 12.13 for $r_1 \bowtie r_2$, where $r_1$ and $r_2$ are as defined in Practice Exercise 12.3.
12.15 The hash-join algorithm as described in Section 12.5.5 computes the natural join of two relations. Describe how to extend the hash-join algorithm to compute the natural left outer join, the natural right outer join and the natural full outer join. (Hint: Keep extra information with each tuple in the hash index, to detect whether any tuple in the probe relation matches the tuple in the hash index.) Try out your algorithm on the takes and student relations.

12.16 Pipelining is used to avoid writing intermediate results to disk. Suppose you need to sort relation \( r \) using sort–merge and merge-join the result with an already sorted relation \( s \).

a. Describe how the output of the sort of \( r \) can be pipelined to the merge join without being written back to disk.

b. The same idea is applicable even if both inputs to the merge join are the outputs of sort–merge operations. However, the available memory has to be shared between the two merge operations (the merge-join algorithm itself needs very little memory). What is the effect of having to share memory on the cost of each sort–merge operation?

12.17 Write pseudocode for an iterator that implements a version of the sort–merge algorithm where the result of the final merge is pipelined to its consumers. Your pseudocode must define the standard iterator functions \( \text{open}() \), \( \text{next}() \), and \( \text{close}() \). Show what state information the iterator must maintain between calls.

12.18 Suppose you have to compute \( A \sum_{C} (r) \) as well as \( A, B \sum_{C} (r) \). Describe how to compute these together using a single sorting of \( r \).

Bibliographical Notes

A query processor must parse statements in the query language, and must translate them into an internal form. Parsing of query languages differs little from parsing of traditional programming languages. Most compiler texts cover the main parsing techniques, and present optimization from a programming-language point of view.

Graefe and McKenna [1993b] presents an excellent survey of query-evaluation techniques.

Knuth [1973] presents an excellent description of external sorting algorithms, including an optimization called replacement selection, which can create initial runs that are (on the average) twice the size of memory. Nyberg et al. [1995] shows that due to poor processor-cache behavior, replacement selection performs worse than in-memory quicksort for run generation, negating the benefits of generating longer runs. Nyberg et al. [1995] presents an efficient external sorting algorithm that takes processor cache effects into account. Query evaluation algorithms that
take cache effects into account have been extensively studied; see, for example, Harizopoulos and Ailamaki [2004].

According to performance studies conducted in the mid-1970s, database systems of that period used only nested-loop join and merge join. These studies, including Blasgen and Eswaran [1976], which was related to the development of System R, determined that either the nested-loop join or merge join nearly always provided the optimal join method. Hence, these two were the only join algorithms implemented in System R. However, Blasgen and Eswaran [1976] did not include an analysis of hash-join algorithms. Today, hash joins are considered to be highly efficient and widely used.

Hash-join algorithms were initially developed for parallel database systems. Hybrid hash join is described in Shapiro [1986], Zeller and Gray [1990] and Davison and Graefe [1994] describe hash-join techniques that can adapt to the available memory, which is important in systems where multiple queries may be running at the same time. Graefe et al. [1998] describes the use of hash joins and hash teams, which allow pipelining of hash joins by using the same partitioning for all hash joins in a pipeline sequence, in the Microsoft SQL Server.