16 PHYSICAL DATABASE DESIGN AND TUNING

> Advice to a client who complained about rain leaking through the roof onto the dining table: "Move the table."
>
> —Architect Frank Lloyd Wright

The performance of a DBMS on commonly asked queries and typical update operations is the ultimate measure of a database design. A DBA can improve performance by adjusting some DBMS parameters (e.g., the size of the buffer pool or the frequency of checkpointing) and by identifying performance bottlenecks and adding hardware to eliminate such bottlenecks. The first step in achieving good performance, however, is to make good database design choices, which is the focus of this chapter.

After we have designed the *conceptual* and *external* schemas, that is, created a collection of relations and views along with a set of integrity constraints, we must address performance goals through **physical database design**, in which we design the *physical* schema. As user requirements evolve, it is usually necessary to **tune**, or adjust, all aspects of a database design for good performance.

This chapter is organized as follows. We give an overview of physical database design and tuning in Section 16.1. The most important physical design decisions concern the choice of indexes. We present guidelines for deciding which indexes to create in Section 16.2. These guidelines are illustrated through several examples and developed further in Sections 16.3 through 16.6. In Section 16.3 we present examples that highlight basic alternatives in index selection. In Section 16.4 we look closely at the important issue of clustering; we discuss how to choose clustered indexes and whether to store tuples from different relations near each other (an option supported by some DBMSs). In Section 16.5 we consider the use of indexes with composite or multiple-attribute search keys. In Section 16.6 we emphasize how well-chosen indexes can enable some queries to be answered without ever looking at the actual data records.

In Section 16.7 we survey the main issues of database tuning. In addition to tuning indexes, we may have to tune the conceptual schema, as well as frequently used query and view definitions. We discuss how to refine the conceptual schema in Section 16.8 and how to refine queries and view definitions in Section 16.9. We briefly discuss the performance impact of concurrent access in Section 16.10. We conclude the chap-

Physical design tools: RDBMSs have hitherto provided few tools to assist with physical database design and tuning, but vendors have started to address this issue. Microsoft SQL Server has a tuning wizard that makes suggestions on indexes to create; it also suggests dropping an index when the addition of other indexes makes the maintenance cost of the index outweigh its benefits on queries. IBM DB2 V6 also has a tuning wizard and Oracle Expert makes recommendations on global parameters, suggests adding/deleting indexes etc.

ter with a short discussion of DBMS benchmarks in Section 16.11; benchmarks help evaluate the performance of alternative DBMS products.

16.1 INTRODUCTION TO PHYSICAL DATABASE DESIGN

Like all other aspects of database design, physical design must be guided by the nature of the data and its intended use. In particular, it is important to understand the typical **workload** that the database must support; the workload consists of a mix of queries and updates. Users also have certain requirements about how fast certain queries or updates must run or how many transactions must be processed per second. The workload description and users' performance requirements are the basis on which a number of decisions have to be made during physical database design.

To create a good physical database design and to tune the system for performance in response to evolving user requirements, the designer needs to understand the workings of a DBMS, especially the indexing and query processing techniques supported by the DBMS. If the database is expected to be accessed concurrently by many users, or is a *distributed database*, the task becomes more complicated, and other features of a DBMS come into play. We discuss the impact of concurrency on database design in Section 16.10. We discuss distributed databases in Chapter 21.

16.1.1 Database Workloads

The key to good physical design is arriving at an accurate description of the expected workload. A **workload description** includes the following elements:

1. A list of queries and their frequencies, as a fraction of all queries and updates.
2. A list of updates and their frequencies.
3. Performance goals for each type of query and update.

For each query in the workload, we must identify:

- Which relations are accessed.
- Which attributes are retained (in the `SELECT` clause).
- Which attributes have selection or join conditions expressed on them (in the `WHERE` clause) and how selective these conditions are likely to be.

Similarly, for each update in the workload, we must identify:

- Which attributes have selection or join conditions expressed on them (in the `WHERE` clause) and how selective these conditions are likely to be.
- The type of update (`INSERT`, `DELETE`, or `UPDATE`) and the updated relation.
- For `UPDATE` commands, the fields that are modified by the update.

Remember that queries and updates typically have parameters, for example, a debit or credit operation involves a particular account number. The values of these parameters determine selectivity of selection and join conditions.

Updates have a query component that is used to find the target tuples. This component can benefit from a good physical design and the presence of indexes. On the other hand, updates typically require additional work to maintain indexes on the attributes that they modify. Thus, while queries can only benefit from the presence of an index, an index may either speed up or slow down a given update. Designers should keep this trade-off in mind when creating indexes.

16.1.2 Physical Design and Tuning Decisions

Important decisions made during physical database design and database tuning include the following:

1. *Which indexes to create.*
   - Which relations to index and which field or combination of fields to choose as index search keys.
   - For each index, should it be clustered or unclustered? Should it be dense or sparse?
2. *Whether we should make changes to the conceptual schema in order to enhance performance.* For example, we have to consider:
   - *Alternative normalized schemas:* We usually have more than one way to decompose a schema into a desired normal form (BCNF or 3NF). A choice can be made on the basis of performance criteria.

- *Denormalization:* We might want to reconsider schema decompositions carried out for normalization during the conceptual schema design process to improve the performance of queries that involve attributes from several previously decomposed relations.
- *Vertical partitioning:* Under certain circumstances we might want to further decompose relations to improve the performance of queries that involve only a few attributes.
- *Views:* We might want to add some views to mask the changes in the conceptual schema from users.

3. *Whether frequently executed queries and transactions should be rewritten to run faster.*

In parallel or distributed databases, which we discuss in Chapter 21, there are additional choices to consider, such as whether to partition a relation across different sites or whether to store copies of a relation at multiple sites.

16.1.3 Need for Database Tuning

Accurate, detailed workload information may be hard to come by while doing the initial design of the system. Consequently, tuning a database after it has been designed and deployed is important—we must refine the initial design in the light of actual usage patterns to obtain the best possible performance.

The distinction between database design and database tuning is somewhat arbitrary. We could consider the design process to be over once an initial conceptual schema is designed and a set of indexing and clustering decisions is made. Any subsequent changes to the conceptual schema or the indexes, say, would then be regarded as a tuning activity. Alternatively, we could consider some refinement of the conceptual schema (and physical design decisions affected by this refinement) to be part of the physical design process.

Where we draw the line between design and tuning is not very important, and we will simply discuss the issues of index selection and database tuning without regard to when the tuning activities are carried out.

16.2 GUIDELINES FOR INDEX SELECTION

In considering which indexes to create, we begin with the list of queries (including queries that appear as part of update operations). Obviously, only relations accessed by some query need to be considered as candidates for indexing, and the choice of attributes to index on is guided by the conditions that appear in the `WHERE` clauses of

the queries in the workload. The presence of suitable indexes can significantly improve the evaluation plan for a query, as we saw in Chapter 13.

One approach to index selection is to consider the most important queries in turn, and for each to determine which plan the optimizer would choose given the indexes that are currently on our list of (to be created) indexes. Then we consider whether we can arrive at a substantially better plan by adding more indexes; if so, these additional indexes are candidates for inclusion in our list of indexes. In general, range retrievals will benefit from a B+ tree index, and exact-match retrievals will benefit from a hash index. Clustering will benefit range queries, and it will benefit exact-match queries if several data entries contain the same key value.

Before adding an index to the list, however, we must consider the impact of having this index on the updates in our workload. As we noted earlier, although an index can speed up the query component of an update, all indexes on an updated attribute—on *any* attribute, in the case of inserts and deletes—must be updated whenever the value of the attribute is changed. Therefore, we must sometimes consider the trade-off of slowing some update operations in the workload in order to speed up some queries.

Clearly, choosing a good set of indexes for a given workload requires an understanding of the available indexing techniques, and of the workings of the query optimizer. The following guidelines for index selection summarize our discussion:

Guideline 1 (whether to index): The obvious points are often the most important. Don't build an index unless some query—including the query components of updates—will benefit from it. Whenever possible, choose indexes that speed up more than one query.

Guideline 2 (choice of search key): Attributes mentioned in a `WHERE` clause are candidates for indexing.

- An exact-match selection condition suggests that we should consider an index on the selected attributes, ideally, a hash index.
- A range selection condition suggests that we should consider a B+ tree (or ISAM) index on the selected attributes. A B+ tree index is usually preferable to an ISAM index. An ISAM index may be worth considering if the relation is infrequently updated, but we will assume that a B+ tree index is always chosen over an ISAM index, for simplicity.

Guideline 3 (multiple-attribute search keys): Indexes with multiple-attribute search keys should be considered in the following two situations:

- A `WHERE` clause includes conditions on more than one attribute of a relation.

- They enable index-only evaluation strategies (i.e., accessing the relation can be avoided) for important queries. (This situation could lead to attributes being in the search key even if they do not appear in `WHERE` clauses.)

When creating indexes on search keys with multiple attributes, if range queries are expected, be careful to order the attributes in the search key to match the queries.

Guideline 4 (whether to cluster): At most one index on a given relation can be clustered, and clustering affects performance greatly; so the choice of clustered index is important.

- As a rule of thumb, range queries are likely to benefit the most from clustering. If several range queries are posed on a relation, involving different sets of attributes, consider the selectivity of the queries and their relative frequency in the workload when deciding which index should be clustered.
- If an index enables an index-only evaluation strategy for the query it is intended to speed up, the index need not be clustered. (Clustering matters only when the index is used to retrieve tuples from the underlying relation.)

Guideline 5 (hash versus tree index): A B+ tree index is usually preferable because it supports range queries as well as equality queries. A hash index is better in the following situations:

- The index is intended to support index nested loops join; the indexed relation is the inner relation, and the search key includes the join columns. In this case, the slight improvement of a hash index over a B+ tree for equality selections is magnified, because an equality selection is generated for each tuple in the outer relation.
- There is a very important equality query, and there are no range queries, involving the search key attributes.

Guideline 6 (balancing the cost of index maintenance): After drawing up a 'wishlist' of indexes to create, consider the impact of each index on the updates in the workload.

- If maintaining an index slows down frequent update operations, consider dropping the index.
- Keep in mind, however, that adding an index may well speed up a given update operation. For example, an index on employee ids could speed up the operation of increasing the salary of a given employee (specified by id).

16.3 BASIC EXAMPLES OF INDEX SELECTION

The following examples illustrate how to choose indexes during database design. The schemas used in the examples are not described in detail; in general they contain the attributes named in the queries. Additional information is presented when necessary.

Let us begin with a simple query:

```
SELECT  E.ename, D.mgr
FROM    Employees E, Departments D
WHERE   D.dname='Toy' AND E.dno=D.dno
```

The relations mentioned in the query are Employees and Departments, and both conditions in the `WHERE` clause involve equalities. Our guidelines suggest that we should build hash indexes on the attributes involved. It seems clear that we should build a hash index on the *dname* attribute of Departments. But consider the equality *E.dno=D.dno*. Should we build an index (hash, of course) on the *dno* attribute of Departments or of Employees (or both)? Intuitively, we want to retrieve Departments tuples using the index on *dname* because few tuples are likely to satisfy the equality selection *D.dname='Toy'*.[1] For each qualifying Departments tuple, we then find matching Employees tuples by using an index on the *dno* attribute of Employees. Thus, we should build an index on the *dno* field of Employees. (Note that nothing is gained by building an additional index on the *dno* field of Departments because Departments tuples are retrieved using the *dname* index.)

Our choice of indexes was guided by a query evaluation plan that we wanted to utilize. This consideration of a potential evaluation plan is common while making physical design decisions. Understanding query optimization is very useful for physical design. We show the desired plan for this query in Figure 16.1.

As a variant of this query, suppose that the `WHERE` clause is modified to be `WHERE` *D.dname='Toy'* `AND` *E.dno=D.dno* `AND` *E.age=25*. Let us consider alternative evaluation plans. One good plan is to retrieve Departments tuples that satisfy the selection on *dname* and to retrieve matching Employees tuples by using an index on the *dno* field; the selection on *age* is then applied on-the-fly. However, unlike the previous variant of this query, we do not really need to have an index on the *dno* field of Employees if we have an index on *age*. In this case we can retrieve Departments tuples that satisfy the selection on *dname* (by using the index on *dname*, as before), retrieve Employees tuples that satisfy the selection on *age* by using the index on *age*, and join these sets of tuples. Since the sets of tuples we join are small, they fit in memory and the join method is not important. This plan is likely to be somewhat poorer than using an

[1] This is only a heuristic. If *dname* is not the key, and we do not have statistics to verify this claim, it is possible that several tuples satisfy this condition!

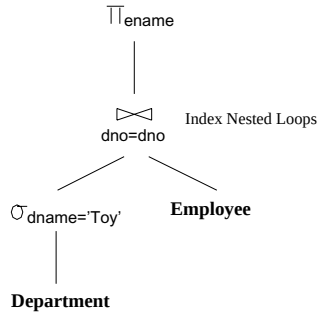

Figure 16.1 A Desirable Query Evaluation Plan

index on *dno*, but it is a reasonable alternative. Therefore, if we have an index on *age* already (prompted by some other query in the workload), this variant of the sample query does not justify creating an index on the *dno* field of Employees.

Our next query involves a range selection:

```
SELECT  E.ename, D.dname
FROM    Employees E, Departments D
WHERE   E.sal BETWEEN 10000 AND 20000
        AND E.hobby='Stamps' AND E.dno=D.dno
```

This query illustrates the use of the `BETWEEN` operator for expressing range selections. It is equivalent to the condition:

$10000 \leq$ E.sal `AND` E.sal ≤ 20000

The use of `BETWEEN` to express range conditions is recommended; it makes it easier for both the user and the optimizer to recognize both parts of the range selection.

Returning to the example query, both (nonjoin) selections are on the Employees relation. Therefore, it is clear that a plan in which Employees is the outer relation and Departments is the inner relation is the best, as in the previous query, and we should build a hash index on the *dno* attribute of Departments. But which index should we build on Employees? A B+ tree index on the *sal* attribute would help with the range selection, especially if it is clustered. A hash index on the *hobby* attribute would help with the equality selection. If one of these indexes is available, we could retrieve Employees tuples using this index, retrieve matching Departments tuples using the index on *dno*, and apply all remaining selections and projections on-the-fly. If both indexes are available, the optimizer would choose the more selective access path for the given query; that is, it would consider which selection (the range condition on *salary* or the equality on *hobby*) has fewer qualifying tuples. In general, which access path is more

selective depends on the data. If there are very few people with salaries in the given range and many people collect stamps, the B+ tree index is best. Otherwise, the hash index on *hobby* is best.

If the query constants are known (as in our example), the selectivities can be estimated if statistics on the data are available. Otherwise, as a rule of thumb, an equality selection is likely to be more selective, and a reasonable decision would be to create a hash index on *hobby*. Sometimes, the query constants are not known—we might obtain a query by expanding a query on a view at run-time, or we might have a query in dynamic SQL, which allows constants to be specified as *wild-card variables* (e.g., *%X*) and instantiated at run-time (see Sections 5.9 and 5.10). In this case, if the query is very important, we might choose to create a B+ tree index on *sal* and a hash index on *hobby* and leave the choice to be made by the optimizer at run-time.

16.4 CLUSTERING AND INDEXING *

Range queries are good candidates for improvement with a clustered index:

```
SELECT   E.dno
FROM     Employees E
WHERE    E.age > 40
```

If we have a B+ tree index on *age*, we can use it to retrieve only tuples that satisfy the selection *E.age*> 40. Whether such an index is worthwhile depends first of all on the selectivity of the condition. What fraction of the employees are older than 40? If virtually everyone is older than 40, we don't gain much by using an index on *age*; a sequential scan of the relation would do almost as well. However, suppose that only 10 percent of the employees are older than 40. Now, is an index useful? The answer depends on whether the index is clustered. If the index is unclustered, we could have one page I/O per qualifying employee, and this could be more expensive than a sequential scan even if only 10 percent of the employees qualify! On the other hand, a clustered B+ tree index on *age* requires only 10 percent of the I/Os for a sequential scan (ignoring the few I/Os needed to traverse from the root to the first retrieved leaf page and the I/Os for the relevant index leaf pages).

As another example, consider the following refinement of the previous query:

```
SELECT   E.dno, COUNT(*)
FROM     Employees E
WHERE    E.age > 10
GROUP BY E.dno
```

If a B+ tree index is available on *age*, we could retrieve tuples using it, sort the retrieved tuples on *dno*, and so answer the query. However, this may not be a good

plan if virtually all employees are more than 10 years old. This plan is especially bad if the index is not clustered.

Let us consider whether an index on *dno* might suit our purposes better. We could use the index to retrieve all tuples, grouped by *dno*, and for each *dno* count the number of tuples with $age > 10$. (This strategy can be used with both hash and B+ tree indexes; we only require the tuples to be *grouped*, not necessarily *sorted*, by *dno*.) Again, the efficiency depends crucially on whether the index is clustered. If it is, this plan is likely to be the best if the condition on *age* is not very selective. (Even if we have a clustered index on *age*, if the condition on *age* is not selective, the cost of sorting qualifying tuples on *dno* is likely to be high.) If the index is not clustered, we could perform one page I/O per tuple in Employees, and this plan would be terrible. Indeed, if the index is not clustered, the optimizer will choose the straightforward plan based on sorting on *dno*. Thus, this query suggests that we build a clustered index on *dno* if the condition on *age* is not very selective. If the condition is very selective, we should consider building an index (not necessarily clustered) on *age* instead.

Clustering is also important for an index on a search key that does not include a candidate key, that is, an index in which several data entries can have the same key value. To illustrate this point, we present the following query:

```
SELECT E.dno
FROM   Employees E
WHERE  E.hobby='Stamps'
```

If many people collect stamps, retrieving tuples through an unclustered index on *hobby* can be very inefficient. It may be cheaper to simply scan the relation to retrieve all tuples and to apply the selection on-the-fly to the retrieved tuples. Therefore, if such a query is important, we should consider making the index on *hobby* a clustered index. On the other hand, if we assume that *eid* is a key for Employees, and replace the condition *E.hobby='Stamps'* by *E.eid=552*, we know that at most one Employees tuple will satisfy this selection condition. In this case, there is no advantage to making the index clustered.

Clustered indexes can be especially important while accessing the inner relation in an index nested loops join. To understand the relationship between clustered indexes and joins, let us revisit our first example.

```
SELECT E.ename, D.mgr
FROM   Employees E, Departments D
WHERE  D.dname='Toy' AND E.dno=D.dno
```

We concluded that a good evaluation plan is to use an index on *dname* to retrieve Departments tuples satisfying the condition on *dname* and to find matching Employees

tuples using an index on *dno*. Should these indexes be clustered? Given our assumption that the number of tuples satisfying *D.dname='Toy'* is likely to be small, we should build an unclustered index on *dname*. On the other hand, Employees is the inner relation in an index nested loops join, and *dno* is not a candidate key. This situation is a strong argument that the index on the *dno* field of Employees should be clustered. In fact, because the join consists of repeatedly posing equality selections on the *dno* field of the inner relation, this type of query is a stronger justification for making the index on *dno* be clustered than a simple selection query such as the previous selection on *hobby*. (Of course, factors such as selectivities and frequency of queries have to be taken into account as well.)

The following example, very similar to the previous one, illustrates how clustered indexes can be used for sort-merge joins.

```
SELECT  E.ename, D.mgr
FROM    Employees E, Departments D
WHERE   E.hobby='Stamps' AND E.dno=D.dno
```

This query differs from the previous query in that the condition *E.hobby='Stamps'* replaces *D.dname='Toy'*. Based on the assumption that there are few employees in the Toy department, we chose indexes that would facilitate an indexed nested loops join with Departments as the outer relation. Now let us suppose that many employees collect stamps. In this case, a block nested loops or sort-merge join might be more efficient. A sort-merge join can take advantage of a clustered B+ tree index on the *dno* attribute in Departments to retrieve tuples and thereby avoid sorting Departments. Note that an unclustered index is not useful—since all tuples are retrieved, performing one I/O per tuple is likely to be prohibitively expensive. If there is no index on the *dno* field of Employees, we could retrieve Employees tuples (possibly using an index on *hobby*, especially if the index is clustered), apply the selection *E.hobby='Stamps'* on-the-fly, and sort the qualifying tuples on *dno*.

As our discussion has indicated, when we retrieve tuples using an index, the impact of clustering depends on the number of retrieved tuples, that is, the number of tuples that satisfy the selection conditions that match the index. An unclustered index is just as good as a clustered index for a selection that retrieves a single tuple (e.g., an equality selection on a candidate key). As the number of retrieved tuples increases, the unclustered index quickly becomes more expensive than even a sequential scan of the entire relation. Although the sequential scan retrieves all tuples, it has the property that each page is retrieved exactly once, whereas a page may be retrieved as often as the number of tuples it contains if an unclustered index is used. If blocked I/O is performed (as is common), the relative advantage of sequential scan versus an unclustered index increases further. (Blocked I/O also speeds up access using a clustered index, of course.)

We illustrate the relationship between the number of retrieved tuples, viewed as a percentage of the total number of tuples in the relation, and the cost of various access methods in Figure 16.2. We assume that the query is a selection on a single relation, for simplicity. (Note that this figure reflects the cost of writing out the result; otherwise, the line for sequential scan would be flat.)

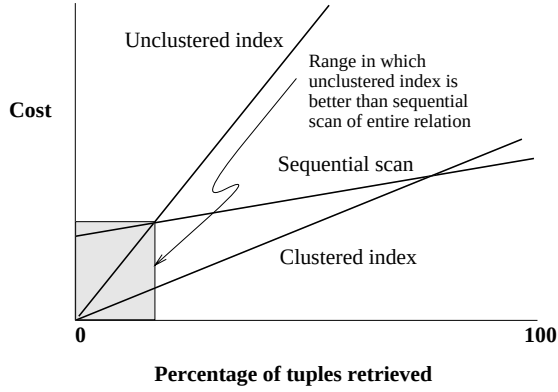

Figure 16.2 The Impact of Clustering

16.4.1 Co-clustering Two Relations

In our description of a typical database system architecture in Chapter 7, we explained how a relation is stored as a file of records. Although a file usually contains only the records of some one relation, some systems allow records from more than one relation to be stored in a single file. The database user can request that the records from two relations be interleaved physically in this manner. This data layout is sometimes referred to as **co-clustering** the two relations. We now discuss when co-clustering can be beneficial.

As an example, consider two relations with the following schemas:

Parts(*pid:* `integer`, *pname:* `string`, *cost:* `integer`, *supplierid:* `integer`)
Assembly(*partid:* `integer`, *componentid:* `integer`, *quantity:* `integer`)

In this schema the *componentid* field of Assembly is intended to be the *pid* of some part that is used as a component in assembling the part with *pid* equal to *partid*. Thus, the Assembly table represents a 1:N relationship between parts and their subparts; a part can have many subparts, but each part is the subpart of at most one part. In the Parts table *pid* is the key. For composite parts (those assembled from other parts, as indicated by the contents of Assembly), the *cost* field is taken to be the cost of assembling the part from its subparts.

Suppose that a frequent query is to find the (immediate) subparts of all parts that are supplied by a given supplier:

```
SELECT  P.pid, A.componentid
FROM    Parts P, Assembly A
WHERE   P.pid = A.partid AND  P.supplierid = 'Acme'
```

A good evaluation plan is to apply the selection condition on Parts and to then retrieve matching Assembly tuples through an index on the *partid* field. Ideally, the index on *partid* should be clustered. This plan is reasonably good. However, if such selections are common and we want to optimize them further, we can *co-cluster* the two tables. In this approach we store records of the two tables together, with each Parts record P followed by all the Assembly records A such that $P.pid = A.partid$. This approach improves on storing the two relations separately and having a clustered index on *partid* because it doesn't need an index lookup to find the Assembly records that match a given Parts record. Thus, for each selection query, we save a few (typically two or three) index page I/Os.

If we are interested in finding the immediate subparts of *all* parts (i.e., the above query without the selection on *supplierid*), creating a clustered index on *partid* and doing an index nested loops join with Assembly as the inner relation offers good performance. An even better strategy is to create a clustered index on the *partid* field of Assembly and the *pid* field of Parts, and to then do a sort-merge join, using the indexes to retrieve tuples in sorted order. This strategy is comparable to doing the join using a co-clustered organization, which involves just one scan of the set of tuples (of Parts and Assembly, which are stored together in interleaved fashion).

The real benefit of co-clustering is illustrated by the following query:

```
SELECT  P.pid, A.componentid
FROM    Parts P, Assembly A
WHERE   P.pid = A.partid AND  P.cost=10
```

Suppose that many parts have $cost = 10$. This query essentially amounts to a collection of queries in which we are given a Parts record and want to find matching Assembly records. If we have an index on the *cost* field of Parts, we can retrieve qualifying Parts tuples. For each such tuple we have to use the index on Assembly to locate records with the given *pid*. The index access for Assembly is avoided if we have a co-clustered organization. (Of course, we still require an index on the *cost* attribute of Parts tuples.)

Such an optimization is especially important if we want to traverse several levels of the part-subpart hierarchy. For example, a common query is to find the total cost of a part, which requires us to repeatedly carry out joins of Parts and Assembly. Incidentally, if we don't know the number of levels in the hierarchy in advance, the

number of joins varies and the query cannot be expressed in SQL. The query can be answered by embedding an SQL statement for the join inside an iterative host language program. How to express the query is orthogonal to our main point here, which is that co-clustering is especially beneficial when the join in question is carried out very frequently (either because it arises repeatedly in an important query such as finding total cost, or because the join query is itself asked very frequently).

To summarize co-clustering:

- It can speed up joins, in particular key–foreign key joins corresponding to 1:N relationships.
- A sequential scan of either relation becomes slower. (In our example, since several Assembly tuples are stored in between consecutive Parts tuples, a scan of all Parts tuples becomes slower than if Parts tuples were stored separately. Similarly, a sequential scan of all Assembly tuples is also slower.)
- Inserts, deletes, and updates that alter record lengths all become slower, thanks to the overheads involved in maintaining the clustering. (We will not discuss the implementation issues involved in co-clustering.)

16.5 INDEXES ON MULTIPLE-ATTRIBUTE SEARCH KEYS *

It is sometimes best to build an index on a search key that contains more than one field. For example, if we want to retrieve Employees records with *age=30* and *sal=4000*, an index with search key $\langle age, sal \rangle$ (or $\langle sal, age \rangle$) is superior to an index with search key *age* or an index with search key *sal*. If we have two indexes, one on *age* and one on *sal*, we could use them both to answer the query by retrieving and intersecting rids. However, if we are considering what indexes to create for the sake of this query, we are better off building one composite index.

Issues such as whether to make the index clustered or unclustered, dense or sparse, and so on are orthogonal to the choice of the search key. We will call indexes on multiple-attribute search keys *composite indexes*. In addition to supporting equality queries on more than one attribute, composite indexes can be used to support multidimensional range queries.

Consider the following query, which returns all employees with $20 < age < 30$ and $3000 < sal < 5000$:

```
SELECT E.eid
FROM   Employees E
WHERE  E.age BETWEEN 20 AND 30
       AND E.sal BETWEEN 3000 AND 5000
```

A composite index on $\langle age, sal \rangle$ could help if the conditions in the `WHERE` clause are fairly selective. Obviously, a hash index will not help; a B+ tree (or ISAM) index is required. It is also clear that a clustered index is likely to be superior to an unclustered index. For this query, in which the conditions on *age* and *sal* are equally selective, a composite, clustered B+ tree index on $\langle age, sal \rangle$ is as effective as a composite, clustered B+ tree index on $\langle sal, age \rangle$. However, the order of search key attributes can sometimes make a big difference, as the next query illustrates:

```
SELECT E.eid
FROM   Employees E
WHERE  E.age = 25
       AND E.sal BETWEEN 3000 AND 5000
```

In this query a composite, clustered B+ tree index on $\langle age, sal \rangle$ will give good performance because records are sorted by *age* first and then (if two records have the same *age* value) by *sal*. Thus, all records with $age = 25$ are clustered together. On the other hand, a composite, clustered B+ tree index on $\langle sal, age \rangle$ will not perform as well. In this case, records are sorted by *sal* first, and therefore two records with the same *age* value (in particular, with $age = 25$) may be quite far apart. In effect, this index allows us to use the range selection on *sal*, but not the equality selection on *age*, to retrieve tuples. (Good performance on both variants of the query can be achieved using a single *spatial* index. We discuss spatial indexes in Chapter 26.)

Some points about composite indexes are worth mentioning. Since data entries in the index contain more information about the data record (i.e., more fields than a single-attribute index), the opportunities for index-only evaluation strategies are increased (see Section 16.6). On the negative side, a composite index must be updated in response to any operation (insert, delete, or update) that modifies *any* field in the search key. A composite index is likely to be larger than a single-attribute search key index because the size of entries is larger. For a composite B+ tree index, this also means a potential increase in the number of levels, although key compression can be used to alleviate this problem (see Section 9.8.1).

16.6 INDEXES THAT ENABLE INDEX-ONLY PLANS *

This section considers a number of queries for which we can find efficient plans that avoid retrieving tuples from one of the referenced relations; instead, these plans scan an associated index (which is likely to be much smaller). An index that is used (only) for index-only scans does *not* have to be clustered because tuples from the indexed relation are not retrieved! However, only dense indexes can be used for the index-only strategies discussed here.

This query retrieves the managers of departments with at least one employee:

```
SELECT D.mgr
FROM   Departments D, Employees E
WHERE  D.dno=E.dno
```

Observe that no attributes of Employees are retained. If we have a dense index on the *dno* field of Employees, the optimization of doing an index nested loops join using an index-only scan for the inner relation is applicable; this optimization is discussed in Section 14.7. Note that it does not matter whether this index is clustered because we do not retrieve Employees tuples anyway. Given this variant of the query, the correct decision is to build an unclustered, dense index on the *dno* field of Employees, rather than a (dense or sparse) clustered index.

The next query takes this idea a step further:

```
SELECT D.mgr, E.eid
FROM   Departments D, Employees E
WHERE  D.dno=E.dno
```

If we have an index on the *dno* field of Employees, we can use it to retrieve Employees tuples during the join (with Departments as the outer relation), but unless the index is clustered, this approach will not be efficient. On the other hand, suppose that we have a dense B+ tree index on $\langle dno, eid \rangle$. Now all the information we need about an Employees tuple is contained in the data entry for this tuple in the index. We can use the index to find the first data entry with a given *dno*; all data entries with the same *dno* are stored together in the index. (Note that a hash index on the composite key $\langle dno, eid \rangle$ cannot be used to locate an entry with just a given *dno*!) We can therefore evaluate this query using an index nested loops join with Departments as the outer relation and an index-only scan of the inner relation.

The next query shows how aggregate operations can influence the choice of indexes:

```
SELECT    E.dno, COUNT(*)
FROM      Employees E
GROUP BY E.dno
```

A straightforward plan for this query is to sort Employees on *dno* in order to compute the count of employees for each *dno*. However, if a dense index—hash or B+ tree—is available, we can answer this query by scanning only the index. For each *dno* value, we simply count the number of data entries in the index with this value for the search key. Note that it does not matter whether the index is clustered because we never retrieve tuples of Employees.

Here is a variation of the previous example:

```
SELECT    E.dno, COUNT(*)
```

```
FROM      Employees E
WHERE     E.sal=10,000
GROUP BY  E.dno
```

An index on *dno* alone will not allow us to evaluate this query with an index-only scan, because we need to look at the *sal* field of each tuple to verify that $sal = 10,000$.

However, we can use an index-only plan if we have a composite B+ tree index on $\langle sal, dno \rangle$ or $\langle dno, sal \rangle$. In an index with key $\langle sal, dno \rangle$, all data entries with $sal = 10,000$ are arranged contiguously (whether or not the index is clustered). Further, these entries are sorted by *dno*, making it easy to obtain a count for each *dno* group. Note that we need to retrieve only data entries with $sal = 10,000$. It is worth observing that this strategy will not work if the `WHERE` clause is modified to use $sal > 10,000$. Although it suffices to retrieve only index data entries—that is, an index-only strategy still applies—these entries must now be sorted by *dno* to identify the groups (because, for example, two entries with the same *dno* but different *sal* values may not be contiguous).

In an index with key $\langle dno, sal \rangle$, data entries with a given *dno* value are stored together, and each such group of entries is itself sorted by *sal*. For each *dno* group, we can eliminate the entries with *sal* not equal to 10,000 and count the rest. We observe that this strategy works even if the `WHERE` clause uses $sal > 10,000$. Of course, this method is less efficient than an index-only scan with key $\langle sal, dno \rangle$ because we must read all data entries.

As another example, suppose that we want to find the minimum *sal* for each *dno*:

```
SELECT    E.dno, MIN(E.sal)
FROM      Employees E
GROUP BY  E.dno
```

An index on *dno* alone will not allow us to evaluate this query with an index-only scan. However, we can use an index-only plan if we have a composite B+ tree index on $\langle dno, sal \rangle$. Notice that all data entries in the index with a given *dno* value are stored together (whether or not the index is clustered). Further, this group of entries is itself sorted by *sal*. An index on $\langle sal, dno \rangle$ would enable us to avoid retrieving data records, but the index data entries must be sorted on *dno*.

Finally consider the following query:

```
SELECT AVG (E.sal)
FROM   Employees E
WHERE  E.age = 25
       AND E.sal BETWEEN 3000 AND 5000
```

A dense, composite B+ tree index on $\langle age, sal \rangle$ allows us to answer the query with an index-only scan. A dense, composite B+ tree index on $\langle sal, age \rangle$ will also allow us to answer the query with an index-only scan, although more index entries are retrieved in this case than with an index on $\langle age, sal \rangle$.

16.7 OVERVIEW OF DATABASE TUNING

After the initial phase of database design, actual use of the database provides a valuable source of detailed information that can be used to refine the initial design. Many of the original assumptions about the expected workload can be replaced by observed usage patterns; in general, some of the initial workload specification will be validated, and some of it will turn out to be wrong. Initial guesses about the size of data can be replaced with actual statistics from the system catalogs (although this information will keep changing as the system evolves). Careful monitoring of queries can reveal unexpected problems; for example, the optimizer may not be using some indexes as intended to produce good plans.

Continued database tuning is important to get the best possible performance. In this section, we introduce three kinds of tuning: *tuning indexes, tuning the conceptual schema,* and *tuning queries*. Our discussion of index selection also applies to index tuning decisions. Conceptual schema and query tuning are discussed further in Sections 16.8 and 16.9.

16.7.1 Tuning Indexes

The initial choice of indexes may be refined for one of several reasons. The simplest reason is that the observed workload reveals that some queries and updates considered important in the initial workload specification are not very frequent. The observed workload may also identify some new queries and updates that *are* important. The initial choice of indexes has to be reviewed in light of this new information. Some of the original indexes may be dropped and new ones added. The reasoning involved is similar to that used in the initial design.

It may also be discovered that the optimizer in a given system is not finding some of the plans that it was expected to. For example, consider the following query, which we discussed earlier:

```
SELECT  D.mgr
FROM    Employees E, Departments D
WHERE   D.dname='Toy' AND E.dno=D.dno
```

A good plan here would be to use an index on *dname* to retrieve Departments tuples with *dname='Toy'* and to use a dense index on the *dno* field of Employees as the inner

relation, using an index-only scan. Anticipating that the optimizer would find such a plan, we might have created a dense, unclustered index on the *dno* field of Employees.

Now suppose that queries of this form take an unexpectedly long time to execute. We can ask to see the plan produced by the optimizer. (Most commercial systems provide a simple command to do this.) If the plan indicates that an index-only scan is not being used, but that Employees tuples are being retrieved, we have to rethink our initial choice of index, given this revelation about our system's (unfortunate) limitations. An alternative to consider here would be to drop the unclustered index on the *dno* field of Employees and to replace it with a clustered index.

Some other common limitations of optimizers are that they do not handle selections involving string expressions, arithmetic, or *null* values effectively. We discuss these points further when we consider query tuning in Section 16.9.

In addition to re-examining our choice of indexes, it pays to periodically reorganize some indexes. For example, a static index such as an ISAM index may have developed long overflow chains. Dropping the index and rebuilding it—if feasible, given the interrupted access to the indexed relation—can substantially improve access times through this index. Even for a dynamic structure such as a B+ tree, if the implementation does not merge pages on deletes, space occupancy can decrease considerably in some situations. This in turn makes the size of the index (in pages) larger than necessary, and could increase the height and therefore the access time. Rebuilding the index should be considered. Extensive updates to a clustered index might also lead to overflow pages being allocated, thereby decreasing the degree of clustering. Again, rebuilding the index may be worthwhile.

Finally, note that the query optimizer relies on statistics maintained in the system catalogs. These statistics are updated only when a special utility program is run; be sure to run the utility frequently enough to keep the statistics reasonably current.

16.7.2 Tuning the Conceptual Schema

In the course of database design, we may realize that our current choice of relation schemas does not enable us meet our performance objectives for the given workload with any (feasible) set of physical design choices. If so, we may have to redesign our conceptual schema (and re-examine physical design decisions that are affected by the changes that we make).

We may realize that a redesign is necessary during the initial design process or later, after the system has been in use for a while. Once a database has been designed and populated with tuples, changing the conceptual schema requires a significant effort in terms of mapping the contents of relations that are affected. Nonetheless, it may

sometimes be necessary to revise the conceptual schema in light of experience with the system. (Such changes to the schema of an operational system are sometimes referred to as **schema evolution**.) We now consider the issues involved in conceptual schema (re)design from the point of view of performance.

The main point to understand is that *our choice of conceptual schema should be guided by a consideration of the queries and updates in our workload, in addition to the issues of redundancy that motivate normalization* (which we discussed in Chapter 15). Several options must be considered while tuning the conceptual schema:

- We may decide to settle for a 3NF design instead of a BCNF design.
- If there are two ways to decompose a given schema into 3NF or BCNF, our choice should be guided by the workload.
- Sometimes we might decide to further decompose a relation that is *already* in BCNF.
- In other situations we might *denormalize*. That is, we might choose to replace a collection of relations obtained by a decomposition from a larger relation with the original (larger) relation, even though it suffers from some redundancy problems. Alternatively, we might choose to add some fields to certain relations to speed up some important queries, even if this leads to a redundant storage of some information (and consequently, a schema that is in neither 3NF nor BCNF).
- This discussion of normalization has concentrated on the technique of *decomposition*, which amounts to vertical partitioning of a relation. Another technique to consider is *horizontal partitioning* of a relation, which would lead to our having two relations with identical schemas. Note that we are not talking about physically partitioning the tuples of a single relation; rather, we want to create two distinct relations (possibly with different constraints and indexes on each).

Incidentally, when we redesign the conceptual schema, especially if we are tuning an existing database schema, it is worth considering whether we should create views to mask these changes from users for whom the original schema is more natural. We will discuss the choices involved in tuning the conceptual schema in Section 16.8.

16.7.3 Tuning Queries and Views

If we notice that a query is running much slower than we expected, we have to examine the query carefully to find the problem. Some rewriting of the query, perhaps in conjunction with some index tuning, can often fix the problem. Similar tuning may be called for if queries on some view run slower than expected. We will not discuss view tuning separately; just think of queries on views as queries in their own right

(after all, queries on views are expanded to account for the view definition before being optimized) and consider how to tune them.

When tuning a query, the first thing to verify is that the system is using the plan that you expect it to use. It may be that the system is not finding the best plan for a variety of reasons. Some common situations that are not handled efficiently by many optimizers follow.

- A selection condition involving *null* values.
- Selection conditions involving arithmetic or string expressions or conditions using the `OR` connective. For example, if we have a condition *E.age = 2*D.age* in the `WHERE` clause, the optimizer may correctly utilize an available index on *E.age* but fail to utilize an available index on *D.age*. Replacing the condition by *E.age/2 = D.age* would reverse the situation.
- Inability to recognize a sophisticated plan such as an index-only scan for an aggregation query involving a `GROUP BY` clause. Of course, virtually no optimizer will look for plans outside the plan space described in Chapters 12 and 13, such as nonleft-deep join trees. So a good understanding of what an optimizer typically does is important. In addition, the more aware you are of a given system's strengths and limitations, the better off you are.

If the optimizer is not smart enough to find the best plan (using access methods and evaluation strategies supported by the DBMS), some systems allow users to guide the choice of a plan by providing hints to the optimizer; for example, users might be able to force the use of a particular index or choose the join order and join method. A user who wishes to guide optimization in this manner should have a thorough understanding of both optimization and the capabilities of the given DBMS. We will discuss query tuning further in Section 16.9.

16.8 CHOICES IN TUNING THE CONCEPTUAL SCHEMA *

We now illustrate the choices involved in tuning the conceptual schema through several examples using the following schemas:

Contracts(*cid:* `integer`, *supplierid:* `integer`, *projectid:* `integer`,
deptid: `integer`, *partid:* `integer`, *qty:* `integer`, *value:* `real`)
Departments(*did:* `integer`, *budget:* `real`, *annualreport:* `varchar`)
Parts(*pid:* `integer`, *cost:* `integer`)
Projects(*jid:* `integer`, *mgr:* `char(20)`)
Suppliers(*sid:* `integer`, *address:* `char(50)`)

For brevity, we will often use the common convention of denoting attributes by a single character and denoting relation schemas by a sequence of characters. Consider

the schema for the relation Contracts, which we will denote as CSJDPQV, with each letter denoting an attribute. The meaning of a tuple in this relation is that the contract with *cid* C is an agreement that supplier S (with *sid* equal to *supplierid*) will supply Q items of part P (with *pid* equal to *partid*) to project J (with *jid* equal to *projectid*) associated with department D (with *deptid* equal to *did*), and that the value V of this contract is equal to *value*.[2]

There are two known integrity constraints with respect to Contracts. A project purchases a given part using a single contract; thus, there will not be two distinct contracts in which the same project buys the same part. This constraint is represented using the FD $JP \rightarrow C$. Also, a department purchases at most one part from any given supplier. This constraint is represented using the FD $SD \rightarrow P$. In addition, of course, the contract id C is a key. The meaning of the other relations should be obvious, and we will not describe them further because our focus will be on the Contracts relation.

16.8.1 Settling for a Weaker Normal Form

Consider the Contracts relation. Should we decompose it into smaller relations? Let us see what normal form it is in. The candidate keys for this relation are C and JP. (C is given to be a key, and JP functionally determines C.) The only nonkey dependency is $SD \rightarrow P$, and P is a *prime* attribute because it is part of candidate key JP. Thus, the relation is not in BCNF—because there is a nonkey dependency—but it is in 3NF.

By using the dependency $SD \rightarrow P$ to guide the decomposition, we get the two schemas SDP and CSJDQV. This decomposition is lossless, but it is not dependency-preserving. However, by adding the relation scheme CJP, we obtain a lossless-join and dependency-preserving decomposition into BCNF. Using the guideline that a dependency-preserving, lossless-join decomposition into BCNF is good, we might decide to replace Contracts by three relations with schemas CJP, SDP, and CSJDQV.

However, suppose that the following query is very frequently asked: Find the number of copies Q of part P ordered in contract C. This query requires a join of the decomposed relations CJP and CSJDQV (or of SDP and CSJDQV), whereas it can be answered directly using the relation Contracts. The added cost for this query could persuade us to settle for a 3NF design and not decompose Contracts further.

16.8.2 Denormalization

The reasons motivating us to settle for a weaker normal form may lead us to take an even more extreme step: deliberately introduce some redundancy. As an example,

[2]If this schema seems complicated, note that real-life situations often call for considerably more complex schemas!

consider the Contracts relation, which is in 3NF. Now, suppose that a frequent query is to check that the value of a contract is less than the budget of the contracting department. We might decide to add a budget field B to Contracts. Since *did* is a key for Departments, we now have the dependency $D \rightarrow B$ in Contracts, which means Contracts is not in 3NF any more. Nonetheless, we might choose to stay with this design if the motivating query is sufficiently important. Such a decision is clearly subjective and comes at the cost of significant redundancy.

16.8.3 Choice of Decompositions

Consider the Contracts relation again. Several choices are possible for dealing with the redundancy in this relation:

- We can leave Contracts as it is and accept the redundancy associated with its being in 3NF rather than BCNF.
- We might decide that we want to avoid the anomalies resulting from this redundancy by decomposing Contracts into BCNF using one of the following methods:
  - We have a lossless-join decomposition into PartInfo with attributes SDP and ContractInfo with attributes CSJDQV. As noted previously, this decomposition is not dependency-preserving, and to make it dependency-preserving would require us to add a third relation CJP, whose sole purpose is to allow us to check the dependency $JP \rightarrow C$.
  - We could choose to replace Contracts by just PartInfo and ContractInfo even though this decomposition is not dependency-preserving.

Replacing Contracts by just PartInfo and ContractInfo does not prevent us from enforcing the constraint $JP \rightarrow C$; it only makes this more expensive. We could create an assertion in SQL-92 to check this constraint:

```
CREATE ASSERTION checkDep
CHECK     ( NOT EXISTS
          ( SELECT  *
          FROM      PartInfo PI, ContractInfo CI
          WHERE     PI.supplierid=CI.supplierid
                    AND PI.deptid=CI.deptid
          GROUP BY CI.projectid, PI.partid
          HAVING    COUNT (cid) > 1 ) )
```

This assertion is expensive to evaluate because it involves a join followed by a sort (to do the grouping). In comparison, the system can check that JP is a primary key for table CJP by maintaining an index on JP. This difference in integrity-checking cost is the motivation for dependency-preservation. On the other hand, if updates are

infrequent, this increased cost may be acceptable; therefore, we might choose not to maintain the table CJP (and quite likely, an index on it).

As another example illustrating decomposition choices, consider the Contracts relation again, and suppose that we also have the integrity constraint that a department uses a given supplier for at most one of its projects: $SPQ \rightarrow V$. Proceeding as before, we have a lossless-join decomposition of Contracts into SDP and CSJDQV. Alternatively, we could begin by using the dependency $SPQ \rightarrow V$ to guide our decomposition, and replace Contracts with SPQV and CSJDPQ. We can then decompose CSJDPQ, guided by $SD \rightarrow P$, to obtain SDP and CSJDQ.

Thus, we now have two alternative lossless-join decompositions of Contracts into BCNF, neither of which is dependency-preserving. The first alternative is to replace Contracts with the relations SDP and CSJDQV. The second alternative is to replace it with SPQV, SDP, and CSJDQ. The addition of CJP makes the second decomposition (but not the first!) dependency-preserving. Again, the cost of maintaining the three relations CJP, SPQV, and CSJDQ (versus just CSJDQV) may lead us to choose the first alternative. In this case, enforcing the given FDs becomes more expensive. We might consider not enforcing them, but we then risk a violation of the integrity of our data.

16.8.4 Vertical Decomposition

Suppose that we have decided to decompose Contracts into SDP and CSJDQV. These schemas are in BCNF, and there is no reason to decompose them further from a normalization standpoint. However, suppose that the following queries are very frequent:

- Find the contracts held by supplier S.
- Find the contracts placed by department D.

These queries might lead us to decompose CSJDQV into CS, CD, and CJQV. The decomposition is lossless, of course, and the two important queries can be answered by examining much smaller relations.

Whenever we decompose a relation, we have to consider which queries the decomposition might adversely affect, especially if the only motivation for the decomposition is improved performance. For example, if another important query is to find the total value of contracts held by a supplier, it would involve a join of the decomposed relations CS and CJQV. In this situation we might decide against the decomposition.

16.8.5 Horizontal Decomposition

Thus far, we have essentially considered how to replace a relation with a collection of vertical decompositions. Sometimes, it is worth considering whether to replace a relation with two relations that have the same attributes as the original relation, each containing a subset of the tuples in the original. Intuitively, this technique is useful when different subsets of tuples are queried in very distinct ways.

For example, different rules may govern large contracts, which are defined as contracts with values greater than 10,000. (Perhaps such contracts have to be awarded through a bidding process.) This constraint could lead to a number of queries in which Contracts tuples are selected using a condition of the form $value > 10,000$. One way to approach this situation is to build a clustered B+ tree index on the *value* field of Contracts. Alternatively, we could replace Contracts with two relations called LargeContracts and SmallContracts, with the obvious meaning. If this query is the only motivation for the index, horizontal decomposition offers all the benefits of the index without the overhead of index maintenance. This alternative is especially attractive if other important queries on Contracts also require clustered indexes (on fields other than *value*).

If we replace Contracts by two relations LargeContracts and SmallContracts, we could mask this change by defining a view called Contracts:

```
CREATE VIEW Contracts(cid, supplierid, projectid, deptid, partid, qty, value)
       AS ((SELECT *
           FROM     LargeContracts)
           UNION
           (SELECT  *
           FROM     SmallContracts))
```

However, any query that deals solely with LargeContracts should be expressed directly on LargeContracts, and not on the view. Expressing the query on the view Contracts with the selection condition $value > 10,000$ is equivalent to expressing the query on LargeContracts, but less efficient. This point is quite general: Although we can mask changes to the conceptual schema by adding view definitions, users concerned about performance have to be aware of the change.

As another example, if Contracts had an additional field *year* and queries typically dealt with the contracts in some one year, we might choose to partition Contracts by year. Of course, queries that involved contracts from more than one year might require us to pose queries against each of the decomposed relations.

16.9 CHOICES IN TUNING QUERIES AND VIEWS *

The first step in tuning a query is to understand the plan that is used by the DBMS to evaluate the query. Systems usually provide some facility for identifying the plan used to evaluate a query. Once we understand the plan selected by the system, we can consider how to improve performance. We can consider a different choice of indexes or perhaps co-clustering two relations for join queries, guided by our understanding of the old plan and a better plan that we want the DBMS to use. The details are similar to the initial design process.

One point worth making is that before creating new indexes we should consider whether rewriting the query will achieve acceptable results with existing indexes. For example, consider the following query with an `OR` connective:

```
SELECT E.dno
FROM   Employees E
WHERE  E.hobby='Stamps' OR E.age=10
```

If we have indexes on both *hobby* and *age*, we can use these indexes to retrieve the necessary tuples, but an optimizer might fail to recognize this opportunity. The optimizer might view the conditions in the `WHERE` clause as a whole as not matching either index, do a sequential scan of Employees, and apply the selections on-the-fly. Suppose we rewrite the query as the union of two queries, one with the clause `WHERE` *E.hobby='Stamps'* and the other with the clause `WHERE` *E.age=10*. Now each of these queries will be answered efficiently with the aid of the indexes on *hobby* and *age*.

We should also consider rewriting the query to avoid some expensive operations. For example, including `DISTINCT` in the `SELECT` clause leads to duplicate elimination, which can be costly. Thus, we should omit `DISTINCT` whenever possible. For example, for a query on a single relation, we can omit `DISTINCT` whenever either of the following conditions holds:

- We do not care about the presence of duplicates.
- The attributes mentioned in the `SELECT` clause include a candidate key for the relation.

Sometimes a query with `GROUP BY` and `HAVING` can be replaced by a query without these clauses, thereby eliminating a sort operation. For example, consider:

```
SELECT   MIN (E.age)
FROM     Employees E
GROUP BY E.dno
HAVING   E.dno=102
```

This query is equivalent to

```
SELECT   MIN (E.age)
FROM     Employees E
WHERE    E.dno=102
```

Complex queries are often written in steps, using a temporary relation. We can usually rewrite such queries without the temporary relation to make them run faster. Consider the following query for computing the average salary of departments managed by Robinson:

```
SELECT   *
INTO     Temp
FROM     Employees E, Departments D
WHERE    E.dno=D.dno AND D.mgrname='Robinson'

SELECT   T.dno, AVG (T.sal)
FROM     Temp T
GROUP BY T.dno
```

This query can be rewritten as

```
SELECT   E.dno, AVG (E.sal)
FROM     Employees E, Departments D
WHERE    E.dno=D.dno AND D.mgrname='Robinson'
GROUP BY E.dno
```

The rewritten query does not materialize the intermediate relation Temp and is therefore likely to be faster. In fact, the optimizer may even find a very efficient index-only plan that never retrieves Employees tuples if there is a dense, composite B+ tree index on $\langle dno, sal \rangle$. This example illustrates a general observation: By rewriting queries to avoid unnecessary temporaries, we not only avoid creating the temporary relations, we also open up more optimization possibilities for the optimizer to explore.

In some situations, however, if the optimizer is unable to find a good plan for a complex query (typically a nested query with correlation), it may be worthwhile to rewrite the query using temporary relations to guide the optimizer toward a good plan.

In fact, nested queries are a common source of inefficiency because many optimizers deal poorly with them, as discussed in Section 14.5. Whenever possible, it is better to rewrite a nested query without nesting and to rewrite a correlated query without correlation. As already noted, a good reformulation of the query may require us to introduce new, temporary relations, and techniques to do so systematically (ideally, to

be done by the optimizer) have been widely studied. Often though, it is possible to rewrite nested queries without nesting or the use of temporary relations, as illustrated in Section 14.5.

16.10 IMPACT OF CONCURRENCY *

In a system with many concurrent users, several additional points must be considered. As we saw in Chapter 1, each user's program (*transaction*) obtains *locks* on the pages that it reads or writes. Other transactions cannot access locked pages until this transaction completes and releases the locks. This restriction can lead to *contention* for locks on heavily used pages.

- The duration for which transactions hold locks can affect performance significantly. Tuning transactions by writing to local program variables and deferring changes to the database until the end of the transaction (and thereby delaying the acquisition of the corresponding locks) can greatly improve performance. On a related note, performance can be improved by replacing a transaction with several smaller transactions, each of which holds locks for a shorter time.
- At the physical level, a careful partitioning of the tuples in a relation and its associated indexes across a collection of disks can significantly improve concurrent access. For example, if we have the relation on one disk and an index on another, accesses to the index can proceed without interfering with accesses to the relation, at least at the level of disk reads.
- If a relation is updated frequently, B+ tree indexes in particular can become a concurrency control bottleneck because all accesses through the index must go through the root; thus, the root and index pages just below it can become **hotspots**, that is, pages for which there is heavy contention. If the DBMS uses specialized locking protocols for tree indexes, and in particular, sets fine-granularity locks, this problem is greatly alleviated. Many current systems use such techniques. Nonetheless, this consideration may lead us to choose an ISAM index in some situations. Because the index levels of an ISAM index are static, we do not need to obtain locks on these pages; only the leaf pages need to be locked. An ISAM index may be preferable to a B+ tree index, for example, if frequent updates occur but we expect the relative distribution of records and the number (and size) of records with a given range of search key values to stay approximately the same. In this case the ISAM index offers a lower locking overhead (and reduced contention for locks), and the distribution of records is such that few overflow pages will be created.

  Hashed indexes do not create such a concurrency bottleneck, unless the data distribution is very skewed and many data items are concentrated in a few buckets. In this case the directory entries for these buckets can become a hotspot.
- The *pattern* of updates to a relation can also become significant. For example, if tuples are inserted into the Employees relation in *eid* order and we have a B+

tree index on *eid*, each insert will go to the last leaf page of the B+ tree. This leads to hotspots along the path from the root to the right-most leaf page. Such considerations may lead us to choose a hash index over a B+ tree index or to index on a different field. (Note that this pattern of access leads to poor performance for ISAM indexes as well, since the last leaf page becomes a hot spot.)

Again, this is not a problem for hash indexes because the hashing process randomizes the bucket into which a record is inserted.

- SQL features for specifying transaction properties, which we discuss in Section 19.4, can be used for improving performance. If a transaction does not modify the database, we should specify that its *access mode* is `READ ONLY`. Sometimes it is acceptable for a transaction (e.g., one that computes statistical summaries) to see some anomalous data due to concurrent execution. For such transactions, more concurrency can be achieved by controlling a parameter called the *isolation level*.

16.11 DBMS BENCHMARKING *

Thus far, we have considered how to improve the design of a database to obtain better performance. As the database grows, however, the underlying DBMS may no longer be able to provide adequate performance even with the best possible design, and we have to consider upgrading our system, typically by buying faster hardware and additional memory. We may also consider migrating our database to a new DBMS.

When evaluating DBMS products, performance is an important consideration. A DBMS is a complex piece of software, and different vendors may target their systems toward different market segments by putting more effort into optimizing certain parts of the system, or by choosing different system designs. For example, some systems are designed to run complex queries efficiently, while others are designed to run many simple transactions per second. Within each category of systems, there are many competing products. To assist users in choosing a DBMS that is well suited to their needs, several **performance benchmarks** have been developed. These include benchmarks for measuring the performance of a certain class of applications (e.g., the TPC benchmarks) and benchmarks for measuring how well a DBMS performs various operations (e.g., the Wisconsin benchmark).

Benchmarks should be portable, easy to understand, and scale naturally to larger problem instances. They should measure *peak performance* (e.g., *transactions per second*, or *tps*) as well as *price/performance ratios* (e.g., \$/*tps*) for typical workloads in a given application domain. The Transaction Processing Council (TPC) was created to define benchmarks for transaction processing and database systems. Other well-known benchmarks have been proposed by academic researchers and industry organizations. Benchmarks that are proprietary to a given vendor are not very useful for comparing

different systems (although they may be useful in determining how well a given system would handle a particular workload).

16.11.1 Well-Known DBMS Benchmarks

On-line Transaction Processing Benchmarks: The TPC-A and TPC-B benchmarks constitute the standard definitions of the *tps* and \$/*tps* measures. TPC-A measures the performance and price of a computer network in addition to the DBMS, whereas the TPC-B benchmark considers the DBMS by itself. These benchmarks involve a simple transaction that updates three data records, from three different tables, and appends a record to a fourth table. A number of details (e.g., transaction arrival distribution, interconnect method, system properties) are rigorously specified, ensuring that results for different systems can be meaningfully compared. The TPC-C benchmark is a more complex suite of transactional tasks than TPC-A and TPC-B. It models a warehouse that tracks items supplied to customers and involves five types of transactions. Each TPC-C transaction is much more expensive than a TPC-A or TPC-B transaction, and TPC-C exercises a much wider range of system capabilities, such as use of secondary indexes and transaction aborts. It has more or less completely replaced TPC-A and TPC-B as the standard transaction processing benchmark.

Query Benchmarks: The Wisconsin benchmark is widely used for measuring the performance of simple relational queries. The Set Query benchmark measures the performance of a suite of more complex queries, and the AS^3AP benchmark measures the performance of a mixed workload of transactions, relational queries, and utility functions. The TPC-D benchmark is a suite of complex SQL queries, intended to be representative of the decision-support application domain. The OLAP Council has also developed a benchmark for complex decision-support queries, including some queries that cannot be expressed easily in SQL; this is intended to measure systems for *on-line analytic processing (OLAP)*, which we discuss in Chapter 23, rather than traditional SQL systems. The Sequoia 2000 benchmark is designed to compare DBMS support for geographic information systems.

Object-Database Benchmarks: The 001 and 007 benchmarks measure the performance of object-oriented database systems. The Bucky benchmark measures the performance of object-relational database systems. (We discuss object database systems in Chapter 25.)

16.11.2 Using a Benchmark

Benchmarks should be used with a good understanding of what they are designed to measure and the application environment in which a DBMS is to be used. When you

use benchmarks to guide your choice of a DBMS, keep the following guidelines in mind:

- **How meaningful is a given benchmark?** Benchmarks that try to distill performance into a single number can be overly simplistic. A DBMS is a complex piece of software used in a variety of applications. A good benchmark should have a suite of tasks that are carefully chosen to cover a particular application domain and to test DBMS features that are important for that domain.
- **How well does a benchmark reflect your workload?** You should consider your expected workload and compare it with the benchmark. Give more weight to the performance of those benchmark tasks (i.e., queries and updates) that are similar to important tasks in your workload. Also consider how benchmark numbers are measured. For example, elapsed times for individual queries might be misleading if considered in a multiuser setting: A system may have higher elapsed times because of slower I/O. On a multiuser workload, given sufficient disks for parallel I/O, such a system might outperform a system with a lower elapsed time.
- **Create your own benchmark:** Vendors often tweak their systems in ad hoc ways to obtain good numbers on important benchmarks. To counter this, create your own benchmark by modifying standard benchmarks slightly or by replacing the tasks in a standard benchmark with similar tasks from your workload.

16.12 POINTS TO REVIEW

- In *physical design*, we adjust the physical schema according to the typical query and update workload. A *workload description* contains detailed information about queries, updates, and their frequencies. During physical design, we might create indexes, make changes to the conceptual schema, and/or rewrite queries. **(Section 16.1)**
- There are guidelines that help us to decide whether to index, what to index, whether to use a multiple-attribute index, whether to create an unclustered or a clustered index, and whether to use a hash or a tree index. Indexes can speed up queries but can also slow down update operations. **(Section 16.2)**
- When choosing indexes, we must consider complete query plans including potential join methods that benefit from the indexes. It is not enough to just consider the conditions in the `WHERE` clause as selection criteria for accessing individual relations. **(Section 16.3)**
- Range queries can benefit from clustered indexes. When deciding which index to create, we have to take the selectivity of conditions in the `WHERE` clause into account. Some systems allow us to store records from more than one relation in a single file. This physical layout, called *co-clustering*, can speed up key–foreign key joins, which arise frequently in practice. **(Section 16.4)**

- If the WHERE condition has several conjunctions involving different attributes, an index on a search key with more than one field, called a *composite index*, can improve query performance. **(Section 16.5)**
- Query plans that do not have to retrieve records from an underlying relation are called *index-only plans*. Indexes that are used for index-only access do not need to be clustered. **(Section 16.6)**
- After an initial physical design, continuous database tuning is important to obtain best possible performance. Using the observed workload over time, we can reconsider our choice of indexes and our relation schema. Other tasks include periodic reorganization of indexes and updating the statistics in the system catalogs. **(Section 16.7)**
- We can tune the conceptual schema for increased performance by settling for a weaker normal form or denormalizing a relation to speed up some important query. Usually, we have several decomposition choices that we need to investigate carefully. In some cases we can increase performance through vertical or horizontal decomposition of a relation. **(Section 16.8)**
- When tuning queries, we first need to understand the query plan that the DBMS generates. Sometimes, query performance can be improved by rewriting the query in order to help the DBMS find a better query plan. **(Section 16.9)**
- If many users access the database concurrently, *lock contention* can decrease performance. Several possibilities exist for decreasing concurrency bottlenecks. **(Section 16.10)**
- A *DBMS benchmark* tests the performance of a class of applications or specific aspects of a DBMS to help users evaluate system performance. Well-known benchmarks include TPC-A, TPC-B, TPC-C, and TPC-D. **(Section 16.11)**

EXERCISES

Exercise 16.1 Consider the following relations:

Emp(*eid:* integer, *ename:* varchar, *sal:* integer, *age:* integer, *did:* integer)
Dept(*did:* integer, *budget:* integer, *floor:* integer, *mgr_eid:* integer)

Salaries range from $10,000 to $100,000, ages vary from 20 to 80, each department has about five employees on average, there are 10 floors, and budgets vary from $10,000 to $1,000,000. You can assume uniform distributions of values.

For each of the following queries, which of the listed index choices would you choose to speed up the query? If your database system does not consider index-only plans (i.e., data records are always retrieved even if enough information is available in the index entry), how would your answer change? Explain briefly.

1. Query: *Print ename, age, and sal for all employees.*
   (a) Clustered, dense hash index on $\langle ename, age, sal \rangle$ fields of Emp.
   (b) Unclustered hash index on $\langle ename, age, sal \rangle$ fields of Emp.
   (c) Clustered, sparse B+ tree index on $\langle ename, age, sal \rangle$ fields of Emp.
   (d) Unclustered hash index on $\langle eid, did \rangle$ fields of Emp.
   (e) No index.
2. Query: *Find the dids of departments that are on the 10th floor and that have a budget of less than $15,000.*
   (a) Clustered, dense hash index on the $floor$ field of Dept.
   (b) Unclustered hash index on the $floor$ field of Dept.
   (c) Clustered, dense B+ tree index on $\langle floor, budget \rangle$ fields of Dept.
   (d) Clustered, sparse B+ tree index on the *budget* field of Dept.
   (e) No index.
3. Query: *Find the names of employees who manage some department and have a salary greater than $12,000.*
   (a) Clustered, sparse B+ tree index on the *sal* field of Emp.
   (b) Clustered hash index on the *did* field of Dept.
   (c) Unclustered hash index on the *did* field of Dept.
   (d) Unclustered hash index on the *did* field of Emp.
   (e) Clustered B+ tree index on *sal* field of Emp and clustered hash index on the *did* field of Dept.
4. Query: *Print the average salary for each department.*
   (a) Clustered, sparse B+ tree index on the *did* field of Emp.
   (b) Clustered, dense B+ tree index on the *did* field of Emp.
   (c) Clustered, dense B+ tree index on $\langle did, sal \rangle$ fields of Emp.
   (d) Unclustered hash index on $\langle did, sal \rangle$ fields of Emp.
   (e) Clustered, dense B+ tree index on the *did* field of Dept.

Exercise 16.2 Consider the following relation:

Emp(*eid:* `integer`, *sal:* `integer`, *age:* `real`, *did:* `integer`)

There is a clustered index on *eid* and an unclustered index on *age*.

1. Which factors would you consider in deciding whether to make an index on a relation a clustered index? Would you always create at least one clustered index on every relation?
2. How would you use the indexes to enforce the constraint that *eid* is a key?
3. Give an example of an update that is *definitely speeded up* because of the available indexes. (English description is sufficient.)

4. Give an example of an update that is *definitely slowed down* because of the indexes. (English description is sufficient.)
5. Can you give an example of an update that is neither speeded up nor slowed down by the indexes?

Exercise 16.3 Consider the following BCNF schema for a portion of a simple corporate database (type information is not relevant to this question and is omitted):

Emp (*eid, ename, addr, sal, age, yrs, deptid*)
Dept (*did, dname, floor, budget*)

Suppose you know that the following queries are the six most common queries in the workload for this corporation and that all six are roughly equivalent in frequency and importance:

- List the id, name, and address of employees in a user-specified age range.
- List the id, name, and address of employees who work in the department with a user-specified department name.
- List the id and address of employees with a user-specified employee name.
- List the overall average salary for employees.
- List the average salary for employees of each age; that is, for each age in the database, list the age and the corresponding average salary.
- List all the department information, ordered by department floor numbers.

1. Given this information, and assuming that these queries are more important than any updates, design a physical schema for the corporate database that will give good performance for the expected workload. In particular, decide which attributes will be indexed and whether each index will be a clustered index or an unclustered index. Assume that B+ tree indexes are the only index type supported by the DBMS and that both single- and multiple-attribute keys are permitted. Specify your physical design by identifying the attributes that you recommend indexing on via clustered or unclustered B+ trees.
2. Redesign the physical schema assuming that the set of important queries is changed to be the following:
   - List the id and address of employees with a user-specified employee name.
   - List the overall maximum salary for employees.
   - List the average salary for employees by department; that is, for each *deptid* value, list the *deptid* value and the average salary of employees in that department.
   - List the sum of the budgets of all departments by floor; that is, for each floor, list the floor and the sum.

Exercise 16.4 Consider the following BCNF relational schema for a portion of a university database (type information is not relevant to this question and is omitted):

Prof(*ssno, pname, office, age, sex, specialty, dept_did*)
Dept(*did, dname, budget, num_majors, chair_ssno*)

Suppose you know that the following queries are the five most common queries in the workload for this university and that all five are roughly equivalent in frequency and importance:

- List the names, ages, and offices of professors of a user-specified sex (male or female) who have a user-specified research specialty (e.g., *recursive query processing*). Assume that the university has a diverse set of faculty members, making it very uncommon for more than a few professors to have the same research specialty.
- List all the department information for departments with professors in a user-specified age range.
- List the department id, department name, and chairperson name for departments with a user-specified number of majors.
- List the lowest budget for a department in the university.
- List all the information about professors who are department chairpersons.

These queries occur much more frequently than updates, so you should build whatever indexes you need to speed up these queries. However, you should not build any unnecessary indexes, as updates will occur (and would be slowed down by unnecessary indexes). Given this information, design a physical schema for the university database that will give good performance for the expected workload. In particular, decide which attributes should be indexed and whether each index should be a clustered index or an unclustered index. Assume that both B+ trees and hashed indexes are supported by the DBMS and that both single- and multiple-attribute index search keys are permitted.

1. Specify your physical design by identifying the attributes that you recommend indexing on, indicating whether each index should be clustered or unclustered and whether it should be a B+ tree or a hashed index.
2. Redesign the physical schema assuming that the set of important queries is changed to be the following:
   - List the number of different specialties covered by professors in each department, by department.
   - Find the department with the fewest majors.
   - Find the youngest professor who is a department chairperson.

Exercise 16.5 Consider the following BCNF relational schema for a portion of a company database (type information is not relevant to this question and is omitted):

Project(*pno, proj_name, proj_base_dept, proj_mgr, topic, budget*)
Manager(*mid, mgr_name, mgr_dept, salary, age, sex*)

Note that each project is based in some department, each manager is employed in some department, and the manager of a project need not be employed in the same department (in which the project is based). Suppose you know that the following queries are the five most common queries in the workload for this university and that all five are roughly equivalent in frequency and importance:

- List the names, ages, and salaries of managers of a user-specified sex (male or female) working in a given department. You can assume that while there are many departments, each department contains very few project managers.

- List the names of all projects with managers whose ages are in a user-specified range (e.g., younger than 30).
- List the names of all departments such that a manager in this department manages a project based in this department.
- List the name of the project with the lowest budget.
- List the names of all managers in the same department as a given project.

These queries occur much more frequently than updates, so you should build whatever indexes you need to speed up these queries. However, you should not build any unnecessary indexes, as updates will occur (and would be slowed down by unnecessary indexes). Given this information, design a physical schema for the company database that will give good performance for the expected workload. In particular, decide which attributes should be indexed and whether each index should be a clustered index or an unclustered index. Assume that both B+ trees and hashed indexes are supported by the DBMS, and that both single- and multiple-attribute index keys are permitted.

1. Specify your physical design by identifying the attributes that you recommend indexing on, indicating whether each index should be clustered or unclustered and whether it should be a B+ tree or a hashed index.
2. Redesign the physical schema assuming that the set of important queries is changed to be the following:
   - Find the total of the budgets for projects managed by each manager; that is, list *proj_mgr* and the total of the budgets of projects managed by that manager, for all values of *proj_mgr*.
   - Find the total of the budgets for projects managed by each manager but only for managers who are in a user-specified age range.
   - Find the number of male managers.
   - Find the average age of managers.

Exercise 16.6 The Globetrotters Club is organized into chapters. The president of a chapter can never serve as the president of any other chapter, and each chapter gives its president some salary. Chapters keep moving to new locations, and a new president is elected when (and only when) a chapter moves. The above data is stored in a relation G(C,S,L,P), where the attributes are chapters (C), salaries (S), locations (L), and presidents (P). Queries of the following form are frequently asked, and you *must* be able to answer them without computing a join: "Who was the president of chapter X when it was in location Y?"

1. List the FDs that are given to hold over G.
2. What are the candidate keys for relation G?
3. What normal form is the schema G in?
4. Design a good database schema for the club. (Remember that your design *must* satisfy the query requirement stated above!)
5. What normal form is your good schema in? Give an example of a query that is likely to run slower on this schema than on the relation G.

6. Is there a lossless-join, dependency-preserving decomposition of G into BCNF?
7. Is there ever a good reason to accept something less than 3NF when designing a schema for a relational database? Use this example, if necessary adding further constraints, to illustrate your answer.

Exercise 16.7 Consider the following BCNF relation, which lists the ids, types (e.g., nuts or bolts), and costs of various parts, along with the number that are available or in stock:

Parts (*pid, pname, cost, num_avail*)

You are told that the following two queries are extremely important:

- Find the total number available by part type, for all types. (That is, the sum of the *num_avail* value of all nuts, the sum of the *num_avail* value of all bolts, etc.)
- List the *pid*s of parts with the highest cost.

1. Describe the physical design that you would choose for this relation. That is, what kind of a file structure would you choose for the set of Parts records, and what indexes would you create?
2. Suppose that your customers subsequently complain that performance is still not satisfactory (given the indexes and file organization that you chose for the Parts relation in response to the previous question). Since you cannot afford to buy new hardware or software, you have to consider a schema redesign. Explain how you would try to obtain better performance by describing the schema for the relation(s) that you would use and your choice of file organizations and indexes on these relations.
3. How would your answers to the above two questions change, if at all, if your system did not support indexes with multiple-attribute search keys?

Exercise 16.8 Consider the following BCNF relations, which describe employees and departments that they work in:

Emp (*eid, sal, did*)
Dept (*did, location, budget*)

You are told that the following queries are extremely important:

- Find the location where a user-specified employee works.
- Check whether the budget of a department is greater than the salary of each employee in that department.

1. Describe the physical design that you would choose for this relation. That is, what kind of a file structure would you choose for these relations, and what indexes would you create?
2. Suppose that your customers subsequently complain that performance is still not satisfactory (given the indexes and file organization that you chose for the relations in response to the previous question). Since you cannot afford to buy new hardware or software, you have to consider a schema redesign. Explain how you would try to obtain better performance by describing the schema for the relation(s) that you would use and your choice of file organizations and indexes on these relations.

3. Suppose that your database system has very inefficient implementations of index structures. What kind of a design would you try in this case?

Exercise 16.9 Consider the following BCNF relations, which describe departments in a company and employees:

Dept(*did*, *dname*, *location*, *managerid*)
Emp(*eid*, *sal*)

You are told that the following queries are extremely important:

- List the names and ids of managers for each department in a user-specified location, in alphabetical order by department name.
- Find the average salary of employees who manage departments in a user-specified location. You can assume that no one manages more than one department.

1. Describe the file structures and indexes that you would choose.
2. You subsequently realize that updates to these relations are frequent. Because indexes incur a high overhead, can you think of a way to improve performance on these queries without using indexes?

Exercise 16.10 For each of the following queries, identify one possible reason why an optimizer might not find a good plan. Rewrite the query so that a good plan is likely to be found. Any available indexes or known constraints are listed before each query; assume that the relation schemas are consistent with the attributes referred to in the query.

1. An index is available on the *age* attribute.

```
SELECT E.dno
FROM   Employee E
WHERE  E.age=20 OR E.age=10
```

2. A B+ tree index is available on the *age* attribute.

```
SELECT E.dno
FROM   Employee E
WHERE  E.age<20 AND E.age>10
```

3. An index is available on the *age* attribute.

```
SELECT E.dno
FROM   Employee E
WHERE  2*E.age<20
```

4. No indexes are available.

```
SELECT DISTINCT *
FROM   Employee E
```

5. No indexes are available.

```
SELECT   AVG (E.sal)
FROM     Employee E
GROUP BY E.dno
HAVING   E.dno=22
```

6. *sid* in Reserves is a foreign key that refers to Sailors.

```
SELECT   S.sid
FROM     Sailors S, Reserves R
WHERE    S.sid=R.sid
```

Exercise 16.11 Consider the following two ways of computing the names of employees who earn more than $100,000 and whose age is equal to their manager's age. First, a nested query:

```
SELECT   E1.ename
FROM     Emp E1
WHERE    E1.sal > 100 AND E1.age = ( SELECT E2.age
                                     FROM   Emp E2, Dept D2
                                     WHERE  E1.dname = D2.dname
                                            AND D2.mgr = E2.ename )
```

Second, a query that uses a view definition:

```
SELECT   E1.ename
FROM     Emp E1, MgrAge A
WHERE    E1.dname = A.dname AND E1.sal > 100 AND E1.age = A.age

CREATE VIEW MgrAge (dname, age)
       AS SELECT D.dname, E.age
          FROM   Emp E, Dept D
          WHERE  D.mgr = E.ename
```

1. Describe a situation in which the first query is likely to outperform the second query.
2. Describe a situation in which the second query is likely to outperform the first query.
3. Can you construct an equivalent query that is likely to beat both these queries when every employee who earns more than $100,000 is either 35 or 40 years old? Explain briefly.

PROJECT-BASED EXERCISES

Exercise 16.12 Minibase's Designview tool does not provide any support for choosing indexes or, in general, physical database design. How do you see Designview being used, if at all, in the context of physical database design?

BIBLIOGRAPHIC NOTES

[572] is an early discussion of physical database design. [573] discusses the performance implications of normalization and observes that denormalization may improve performance for certain queries. The ideas underlying a physical design tool from IBM are described in

[234]. The Microsoft AutoAdmin tool that performs automatic index selection according to a query workload is described in [138]. Other approaches to physical database design are described in [125, 557]. [591] considers *transaction tuning*, which we discussed only briefly. The issue is how an application should be structured into a collection of transactions to maximize performance.

The following books on database design cover physical design issues in detail; they are recommended for further reading. [236] is largely independent of specific products, although many examples are based on DB2 and Teradata systems. [684] deals primarily with DB2. [589] is a very readable treatment of performance tuning and is not specific to any one system.

[284] contains several papers on benchmarking database systems and has accompanying software. It includes articles on the AS^3AP, Set Query, TPC-A, TPC-B, Wisconsin, and 001 benchmarks written by the original developers. The Bucky benchmark is described in [112], the 007 benchmark is described in [111], and the TPC-D benchmark is described in [648]. The Sequoia 2000 benchmark is described in [631].