8.NESTED SUBQUERIES

SQL provides a mechanism for nesting subqueries. A subquery is a select-fromwhere expression that is nested within another query. A common use of subqueries is to perform tests for set membership, make set comparisons, and determine set cardinality, by nesting subqueries in the where clause. we see how a class of subqueries called scalar subqueries can appear wherever an expression returning a value can occur.

3.8.1 Set Membership

SQL allows testing tuples for membership in a relation. The in connective tests for set membership, where the set is a collection of values produced by a select clause. The not in connective tests for the absence of set membership.

As an illustration, reconsider the query "Find all the courses taught in the both the Fall 2009 and Spring 2010 semesters." Earlier, we wrote such a query by intersecting two sets: the set of courses taught in Fall 2009 and the set of courses taught in Spring 2010. We can take the alternative approach of finding all courses that were taught in Fall 2009 and that are also members of the set of courses taught in Spring 2010. We begin by finding all courses taught in Spring 2010, and we write the subquery

```
(select course_ id
 from section
 where semester = 'Spring' and year= 2010)
```

We then need to find those courses that were taught in the Fall 2009 and that appear in the set of courses obtained in the subquery. We do so by nesting the subquery in the where clause of an outer query. The resulting query is :

```
select distinct course_ id
 from section
 where semester = 'Fall' and year= 2009 and
 course id in (select course id
 from section
 where semester = 'Spring' and year= 2010);
```

We use the not in construct in a way similar to the in construct. For example, to find all the courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we can write:

```
select distinct course_ id
from section
where semester = 'Fall' and year= 2009 and
course_ id not in
 (select course_ id
 from section
where semester = 'Spring' and year= 2010);
```

3.8.2 Set Comparison

As an example of the ability of a nested subquery to compare sets, consider the query "Find the names of all instructors whose salary is greater than at least one instructor in the Biology department." we wrote this query as follows:

```
select distinct T.name
 from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

SQL does, however, offer an alternative style for writing the preceding query. The phrase "greater than at least one" is represented in SQL by > some. This construct allows us to rewrite the query in a form that resembles closely our formulation of the query in English.

```
select name
from instructor
where salary > some (select salary
                     from instructor
                    where dept name = 'Biology');
```

The subquery generates the set of all salary values of all instructors in the Biology department. The > some comparison in the where clause of the outer select is true

2.VIEWS

SQL allows a "virtual relation" to be defined by a query, and the relation conceptually contains the result of the query. The virtual relation is not precomputed and stored, but instead is computed by executing the query whenever the virtual relation is used.

Any such relation that is not part of the logical model, but is made visible to a user as a virtual relation, is called a **view**. It is possible to support a large number of views on top of any given set of actual relations.

4.2.1 View Definition

We define a view in SQL by using the **create view** command. To define a view, we must give the view a name and must state the query that computes the view. The form of the **create view** command is:

create view v as <query expression>;

where <query expression> is any legal query expression. The view name is represented by v.

Consider again the clerk who needs to access all data in the instructor relation, except salary. The clerk should not be authorized to access the instructor relation (we see later, in Section 4.6, how authorizations can be specified). Instead, a view relation faculty can be made available to the clerk, with the view defined as follows :

```
create view faculty as
select ID, name, dept_name
from instructor;
```

As explained earlier, the view relation conceptually contains the tuples in the query result, but is not precomputed and stored. Instead, the database system stores the query expression associated with the view relation. Whenever the view relation is accessed, its tuples are created by computing the query result. Thus, the view relation is created whenever needed, on demand.

To create a view that lists all course sections offered by the Physics department in the Fall 2009 semester with the building and room number of each section, we write:

```
create view physics_fall_2009 as
    select course.course_id, sec_id, building, room_number
    from course, section
    where course.course_id = section.course_id
                and course.dept_name = 'Physics'
                and section.semester = 'Fall'
                and section.year = '2009';
```

4.2.2 Using Views in SQL Queries

Once we have defined a view, we can use the view name to refer to the virtual relation that the view generates. Using the view *physics_fall_2009*, we can find all Physics courses offered in the Fall 2009 semester in the Watson building by writing:

```
select course_id
from physics_fall_2009
where building= 'Watson';
```

View names may appear in a query any place where a relation name may appear. The attribute names of a view can be specified explicitly as follows:

```
create view departments_total_salary(dept_name, total_salary) as
    select dept_name, sum (salary)
    from instructor
    group by dept_name;
```

4.2.3 Materialized Views

Certain database systems allow view relations to be stored, but they make sure that, if the actual relations used in the view definition change, the view is kept up-to-date. Such views are called **materialized views.**

The process of keeping the materialized view up-to-date is called **materialized view maintenance.**

View maintenance can be done immediately when any of the relations on which the view is defined is updated.

Applications that use a view frequently may benefit if the view is materialized. Applications that demand fast response to certain queries that compute aggregates over large relations can also benefit greatly by creating materialized views corresponding to the queries. In this case, the aggregated result is likely to be much smaller than the large relations on which the view is defined; as a result the materialized view can be used to answer the query very quickly, avoiding reading the large underlying relations. Of course, the benefits to queries from the materialization of a view must be weighed against the storage costs and the added overhead for updates.

SQL does not define a standard way of specifying that a view is materialized, but many database systems provide their own SQL extensions for this task. Some database systems always keep materialized views up-to-date when the underlying

relations change, while others permit them to become out of date, and periodically recompute them.

4.2.4 Update of a View

In general, an SQL view is said to be **updatable** (that is, inserts, updates or deletes can be applied on the view) if the following conditions are all satisfied by the query defining the view:

- The **from** clause has only one database relation.
- The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or distinct specification.
- Any attribute not listed in the select clause can be set to null; that is, it does not have a **not null** constraint and is not part of a primary key.
- The query does not have a **group by** or **having** clause.

Under these constraints, the update, insert, and delete operations would be allowed on the following view:

```
create view history_instructors as
select *
from instructor
where dept_name= 'History';
```

The predicate is not null succeeds if the value on which it is applied is not null. Some implementations of SQL also allow us to test whether the result of a comparison is unknown, rather than true or false, by using the clauses is unknown and is not unknown.

7. AGGREGATE FUNCTIONS

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions:

- Average: avg
- Minimum: min
- Maximum: max
- Total: sum
- Count: count

The input to sum and avg must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.

3.7.1 Basic Aggregation

Consider the query "Find the average salary of instructors in the Computer Science department." We write this query as follows:

```
select avg (salary)
from instructor
where dept _name= 'Comp. Sci.';
```

The result of this query is a relation with a single attribute, containing a single tuple with a numerical value corresponding to the average salary of instructors in the Computer Science department.

```
select avg (salary) as avg _salary
from instructor
where dept_ name= 'Comp. Sci.';
```

In the instructor relation of Figure 2.1, the salaries in the Computer Science department are $75,000, $65,000, and $92,000. The average balance is $232,000/3 = $77,333.33.

There are cases where we must eliminate duplicates before computing an aggregate function. If we do want to eliminate duplicates, we use the keyword distinct in the aggregate expression. An example arises in the query "Find the total number of instructors who teach a course in the Spring 2010 semester."

The required information is contained in the relation teaches, and we write this query as follows:

```
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2010;
```

We use the aggregate function count frequently to count the number of tuples in a relation. The notation for this function in SQL is count (*). Thus, to find the number of tuples in the course relation, we write

```
select count (*)
```

from course;

ID	name	dept_name	salary
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

Figure 3.14 Tuples of the *instructor* relation, grouped by the *dept_name* attribute.

3.7.2 Aggregation with Grouping

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this wish in SQL using the group by clause. The attribute or attributes given in the group by clause are used to form groups. Tuples with the same value on all attributes in the group by clause are placed in one group.

As an illustration, consider the query "Find the average salary in each department." We write this query as follows:

```
select dept name, avg (salary) as avg_salary
 from instructor
 group by dept _name;
```

The specified aggregate is computed for each group, and the result of the query is shown in Figure 3.15.

In contrast, consider the query "Find the average salary of all instructors." We write this query as follows:

```
select avg (salary)
```

from instructor;

dept_name	avg_salary
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Figure 3.15 The result relation for the query "Find the average salary in each department".

3.7.3 The Having Clause

At times, it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those departments where the average salary of the instructors is more than $42,000. This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause. To express such a query, we use the **having** clause of SQL. SQL applies predicates in the **having** clause after groups have been formed, so aggregate functions may be used. We express this query in SQL as follows:

select *dept_name*, **avg** (*salary*) **as** *avg_salary*
from *instructor*
group by *dept_name*
having avg (*salary*) > 42000;

The result is shown in Figure 3.17.

As was the case for the **select** clause, any attribute that is present in the **having** clause without being aggregated must appear in the **group by** clause, otherwise the query is treated as erroneous.

The meaning of a query containing aggregation, **group by**, or **having** clauses is defined by the following sequence of operations:

1. As was the case for queries without aggregation, the from clause is first evaluated to get a relation.

dept_name	avg(avg_salary)
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

Figure 3.17 The result relation for the query "Find the average salary of instructors in those departments where the average salary is more than $42,000."

2. If a where clause is present, the predicate in the where clause is applied on the result relation of the from clause.

3. Tuples satisfying the where predicate are then placed into groups by the group by clause if it is present. If the group by clause is absent, the entire set of tuples satisfying the where predicate is treated as being in one group.

4. The having clause, if it is present, is applied to each group; the groups that do not satisfy the having clause predicate are removed.

5. The select clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.

3.7.4 Aggregation with Null and Boolean Values

Null values, when they exist, complicate the processing of aggregate operators. For example, assume that some tuples in the instructor relation have a null value for salary. Consider the following query to total all salary amounts:

```
select sum (salary) from instructor;
```

The values to be summed in the preceding query include null values, since some tuples have a null value for salary. Rather than say that the overall sum is itself null, the SQL standard says that the sum operator should ignore null values in its input.

A Boolean data type that can take values true, false, and unknown, was introduced in SQL:1999. The aggregate functions some and every, which mean exactly what you would intuitively expect, can be applied on a collection of Boolean values.

```
update instructor
set salary = salary * 1.05
where salary < (select avg (salary)
                from instructor);
```

SQL provides a case construct that we can use to perform both the updates with a single **update** statement, avoiding the problem with the order of updates.

INTERMEDIATE SQL

1. Join Expressions

we introduced the natural join operation. SQL provides other forms of the join operation, including the ability to specify an explicit join predicate, and the ability to include in the result tuples that are excluded by natural join. We shall discuss these forms of join in this section.

The examples in this section involve the two relations student and takes, shown in Figures 4.1 and 4.2, respectively. Observe that the attribute grade has a value null for the student with ID 98988, for the course BIO-301, section 1, taken in Summer 2010. The null value indicates that the grade has not been awarded yet.

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Figure 4.1 The *student* relation.

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

Figure 4.2 The *takes* relation.

4.1.1 Join Conditions

The **on** condition allows a general predicate over the relations being joined. Th predicate is written like a **where** clause predicate except for the use of the ke on rather than **where**. Like the **using** condition, the **on** condition appears at the of the join expression.

Consider the following query, which has a join expression containing the on condition.

```
select *
from student join takes on student.ID= takes.ID;
```

The **on** condition above specifies that a tuple from student matches a tuple from takes if their *ID* values are equal. The join expression in this case is almost the same as the join expression student **natural join** takes, since the natural join operation also requires that for a student tuple and a takes tuple to match. The or difference is that the result has the ID attribute listed twice, in the join result, on for student and once for takes, even though their ID values must be the same.

In fact, the above query is equivalent to the following query:

```
select *
from student, takes
where student.ID= takes.ID;
```

As we have seen earlier, the relation name is used to disambiguate the attribute name *ID*, and thus the two occurrences can be referred to as *student.ID* and *takes.ID* respectively. A version of this query that displays the *ID* value only once is as follows:

```
select student.ID as ID, name, dept_name, tot_cred,
       course_id, sec_id, semester, year, grade
from student join takes on student.ID= takes.ID;
```

The result of the above query is shown in Figure 4.3.

The **on** condition can express any SQL predicate, and thus a join expressions using the **on** condition can express a richer class of join conditions than **natural** join. However, as illustrated by our preceding example, a query using a join expression with an **on** condition can be replaced by an equivalent expression without the **on** condition, with the predicate in the **on** clause moved to the **where** clause. Thus, it may appear that the **on** condition is a redundant feature of SQL.

However, there are two good reasons for introducing the **on** condition. First, we shall see shortly that for a kind of join called an outer join, **on** conditions do behave in a manner different from **where** conditions. Second, an SQL query is often more readable by humans if the join condition is specified in the **on** clause and the rest of the conditions appear in the **where** clause.

4.2 Outer Joins

More generally, some tuples in either or both of the relations being joined may be "lost" in this way. The **outer join** operation works in a manner similar to the join operations we have already studied, but preserve those tuples that would be lost in a join, by creating tuples in the result containing null values.

ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2010	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2009	A
19991	Brandt	History	80	HIS-351	1	Spring	2010	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B-
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	null

Figure 4.3 The result of *student* join *takes* on *student.ID= takes.ID* with second occurrence of *ID* omitted

There are in fact three forms of outer join:

- The **left outer join** preserves tuples only in the relation named before (to the left of) the **left outer join** operation.
- The **right outer join** preserves tuples only in the relation named after (to the right of) the **right outer join** operation.
- The **full outer join** preserves tuples in both relations.

In contrast, the join operations we studied earlier that do not preserve nonmatched tuples are called **inner join** operations, to distinguish them from the outer-join operations.

We now explain exactly how each form of outer join operates. We can compute the left outer-join operation as follows. First, compute the result of the inner join as before. Then, for every tuple t in the left-hand-side relation that does not match any

tuple in the right-hand-side relation in the inner join, add a tuple r to the result of the join constructed as follows:

- The attributes of tuple r that are derived from the left-hand-side relation are filled in with the values from tuple t.
- The remaining attributes of r are filled with null values.

Figure 4.4 shows the result of:

```
select *
from student natural left outer join takes;
```

That result includes student Snow (*ID* 70557), unlike the result of an inner join, but the tuple for Snow includes nulls for the attributes that appear only in the schema of the *takes* relation.

As another example of the use of the outer-join operation, we can write the query "Find all students who have not taken a course" as:

```
select ID
from student natural left outer join takes
where course_id is null;
```

The right outer join is symmetric to the **left outer join**. Tuples from the right-hand-side relation that do not match any tuple in the left-hand-side relation are padded with nulls and are added to the result of the right outer join. Thus, if we rewrite our above query using a right outer join and swapping the order in which we list the relations as follows:

```
select *
from takes natural right outer join student;
```

we get the same result except for the order in which the attributes appear in the result (see Figure 4.5).

The full outer join is a combination of the left and right outer-join types.

As an example of the use of full outer join, consider the following query: "Display a list of all students in the Comp. Sci. department, along with the course sections, if any, that they have taken in Spring 2009; all course sections from Spring 2009 must be displayed, even if no student from the Comp. Sci. department has taken the course section." This query can be written as:

```
select *
from (select *
      from student
      where dept_name= 'Comp. Sci')
     natural full outer join
     (select *
      from takes
      where semester = 'Spring' and year = 2009);
```

ID	name	dept_name	tot_cred	course_id	sec_id	semester	year	grade
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2010	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2009	A
19991	Brandt	History	80	HIS-351	1	Spring	2010	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2010	C-
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B-
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
70557	Snow	Physics	0	*null*	*null*	*null*	*null*	*null*
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	*null*

Figure 4.4 Result of *student* **natural left outer join** *takes*.

ID	course_id	sec_id	semester	year	grade	name	dept_name	tot_cred
00128	CS-101	1	Fall	2009	A	Zhang	Comp. Sci.	102
00128	CS-347	1	Fall	2009	A-	Zhang	Comp. Sci.	102
12345	CS-101	1	Fall	2009	C	Shankar	Comp. Sci.	32
12345	CS-190	2	Spring	2009	A	Shankar	Comp. Sci.	32
12345	CS-315	1	Spring	2010	A	Shankar	Comp. Sci.	32
12345	CS-347	1	Fall	2009	A	Shankar	Comp. Sci.	32
19991	HIS-351	1	Spring	2010	B	Brandt	History	80
23121	FIN-201	1	Spring	2010	C+	Chavez	Finance	110
44553	PHY-101	1	Fall	2009	B-	Peltier	Physics	56
45678	CS-101	1	Fall	2009	F	Levy	Physics	46
45678	CS-101	1	Spring	2010	B+	Levy	Physics	46
45678	CS-319	1	Spring	2010	B	Levy	Physics	46
54321	CS-101	1	Fall	2009	A-	Williams	Comp. Sci.	54
54321	CS-190	2	Spring	2009	B+	Williams	Comp. Sci.	54
55739	MU-199	1	Spring	2010	A-	Sanchez	Music	38
70557	*null*	*null*	*null*	*null*	*null*	Snow	Physics	0
76543	CS-101	1	Fall	2009	A	Brown	Comp. Sci.	58
76543	CS-319	2	Spring	2010	A	Brown	Comp. Sci.	58
76653	EE-181	1	Spring	2009	C	Aoi	Elec. Eng.	60
98765	CS-101	1	Fall	2009	C-	Bourikas	Elec. Eng.	98
98765	CS-315	1	Spring	2010	B	Bourikas	Elec. Eng.	98
98988	BIO-101	1	Summer	2009	A	Tanaka	Biology	120
98988	BIO-301	1	Summer	2010	*null*	Tanaka	Biology	120

Figure 4.5 The result of *takes* **natural right outer join** *student*.

4.1.3 Join Types and Conditions

To distinguish normal joins from outer joins, normal joins are called **inner joins** in SQL. A join clause can thus specify **inner join** instead of **outer join** to specify that a normal join is to be used. The keyword **inner** is, however, optional. The default join type, when the **join** clause is used without the **outer** prefix is the **inner join**. Thus,

```
select *
from student join takes using (ID);
```

is equivalent to:

```
select *
from student inner join takes using (ID);
```

Similarly, **natural join** is equivalent to **natural inner join**.

Figure 4.6 shows a full list of the various types of join that we have discussed. As can be seen from the figure, any form of join (inner, left outer, right outer, or full outer) can be combined with any join condition (natural, using, or on).

1.4.1 Data-Manipulation Language

A data-manipulation language (DML) is a language that enables users to access or manipulate data as organized by the appropriate data model. The types of access are:

- Retrieval of information stored in the database
- Insertion of new information into the database
- Deletion of information from the database
- Modification of information stored in the database

There are basically two types:

- Procedural DMLs require a user to specify *what* data are needed and *how* to get those data.
- Declarative DMLs (also referred to as nonprocedural DMLs) require a user to specify *what* data are needed *without* specifying how to get those data.

Declarative DMLs are usually easier to learn and use than are procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data.

A query is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a query language. Although technically incorrect, it is common practice to use the terms *query language* and *data-manipulation language* synonymously.

1.4.2 Data-Definition Language (DDL)

We specify a database schema by a set of definitions expressed by a special language called a **data-definition language (DDL)**. The DDL is also used to specify additional properties of the data.

We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a data storage and definition language. These statements define the implementation details of the database schemas, which are usually hidden from the users.

The data values stored in the database must satisfy certain consistency constraints. For example, suppose the university requires that the account balance of a department must never be negative. The DDL provides facilities to specify such constraints. The database system checks these constraints every time the database is updated. In general, a constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, database systems implement integrity constraints that can be tested with minimal overhead:

- Domain Constraints. A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.
- Referential Integrity. There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity). For example, the department listed for each course must be one that actually exists. More precisely, the *dept_name* value in a *course* record must appear in the *dept_name* attribute of some record of the *department* relation. Database modifications can cause violations of referential integrity. When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- Assertions. An assertion is any condition that the database must always satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. However, there are many constraints that we cannot express by using only these special forms. For example, "Every department must have at least five courses offered every semester" must be expressed as an assertion. When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated.
- Authorization. We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of authorization, the most common being: read authorization, which allows reading, but not modification, of data; insert authorization, which allows insertion of new data, but not modification of existing data; update authorization, which allows modification, but not deletion, of data; and delete authorization, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization.

8.NESTED SUBQUERIES

SQL provides a mechanism for nesting subqueries. A subquery is a select-fromwhere expression that is nested within another query. A common use of subqueries is to perform tests for set membership, make set comparisons, and determine set cardinality, by nesting subqueries in the where clause. we see how a class of subqueries called scalar subqueries can appear wherever an expression returning a value can occur.

3.8.1 Set Membership

SQL allows testing tuples for membership in a relation. The in connective tests for set membership, where the set is a collection of values produced by a select clause. The not in connective tests for the absence of set membership.

As an illustration, reconsider the query "Find all the courses taught in the both the Fall 2009 and Spring 2010 semesters." Earlier, we wrote such a query by intersecting two sets: the set of courses taught in Fall 2009 and the set of courses taught in Spring 2010. We can take the alternative approach of finding all courses that were taught in Fall 2009 and that are also members of the set of courses taught in Spring 2010. We begin by finding all courses taught in Spring 2010, and we write the subquery

```
(select course_ id
from section
where semester = 'Spring' and year= 2010)
```

We then need to find those courses that were taught in the Fall 2009 and that appear in the set of courses obtained in the subquery. We do so by nesting the subquery in the where clause of an outer query. The resulting query is :

```
select distinct course_ id
from section
where semester = 'Fall' and year= 2009 and
course id in (select course id
from section
where semester = 'Spring' and year= 2010);
```

We use the not in construct in a way similar to the in construct. For example, to find all the courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we can write:

```
select distinct course_ id
from section
where semester = 'Fall' and year= 2009 and
course_ id not in
 (select course_ id
 from section
where semester = 'Spring' and year= 2010);
```

3.8.2 Set Comparison

As an example of the ability of a nested subquery to compare sets, consider the query "Find the names of all instructors whose salary is greater than at least one instructor in the Biology department." we wrote this query as follows:

```
select distinct T.name
 from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

SQL does, however, offer an alternative style for writing the preceding query. The phrase "greater than at least one" is represented in SQL by > some. This construct allows us to rewrite the query in a form that resembles closely our formulation of the query in English.

```
select name
from instructor
where salary > some (select salary
                     from instructor
                    where dept name = 'Biology');
```

The subquery generates the set of all salary values of all instructors in the Biology department. The > some comparison in the where clause of the outer select is true

if the salary value of the tuple is greater than at least one member of the set of all salary values for instructors in Biology.

SQL also allows < some, <= some, >= some, = some, and <> some comparisons. As an exercise, verify that = some is identical to in, whereas <> some is not the same as not in.

The construct > all corresponds to the phrase "greater than all." Using this construct, we write the query as follows:

```
select name
from instructor
where salary > all (select salary
from instructor
where dept name = 'Biology');
```

As it does for some, SQL also allows < all, <= all, >= all, = all, and <> all comparisons. As an exercise, verify that <> all is identical to not in, whereas = all is not the same as in.

3.8.3 Test for Empty Relations

SQL includes a feature for testing whether a subquery has any tuples in its result. The exists construct returns the value true if the argument subquery is nonempty. Using the exists construct, we can write the query "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester" in still another way:

```
select course_ id
from section as S
where semester = 'Fall' and year= 2009 and
exists (select *
                from section as T
                where semester = 'Spring' and year=
        2010 and S.course _id= T.course_ id);
```

A subquery that uses a correlation name from an outer query is called a **correlated subquery.**

We can test for the nonexistence of tuples in a subquery by using the **not exists** construct. We can use the **not exists** construct to simulate the set containment (th is, superset) operation: We can write "relation Acontains relation B" as "**not exis** (B **except** A)." To illustrate the **not exists** operator, consider the query "Find all students who have taken all courses offered in the Biology department." Using th **except** construct, we can write the query as follows:

```
select distinct S.ID, S.name
from student as S
where not exists ((select course _id
from course
where dept_ name = 'Biology')
except
(select T.course_ id
from takes as T
where S.ID = T.ID));
```

Here, the subquery finds the set of all courses offered in the Biology department. The subquery finds all the courses that student S.ID has taken. Thus, the outer select takes each student and tests whether the set of all courses that the student has taken contains the set of all courses offered in the Biology department.

3.8.4 Test for the Absence of Duplicate Tuples

SQL includes a boolean function for testing whether a subquery has duplicate tuples in its result. The **unique** construct returns the value true if the argument subquery contains no duplicate tuples. Using the unique construct, we can write the query "Find all courses that were offered at most once in 2009" as follows:

```
select T.course_ id
from course as T
where unique (select R.course _id
from section as R
where T.course_ id= R.course_ id and R.year = 2009);
```

Note that if a course is not offered in 2009, the subquery would return an empty result, and the **unique** predicate would evaluate to true on the empty set. An equivalent version of the above query not using the **unique** construct is:

```
select T.course_id
from course as T
where 1 <= (select count(R.course_id)
            from section as R
            where T.course_id= R.course_id and
                  R.year = 2009);
```

We can test for the existence of duplicate tuples in a subquery by using the **not unique** construct. To illustrate this construct, consider the query "Find all courses that were offered at least twice in 2009" as follows.

```
select T.course_id
from course as T
where not unique (select R.course_id
                  from section as R
                  where T.course_id= R.course_id and
                        R.year = 2009);
```

Formally, the **unique** test on a relation is defined to fail if and only if the relation contains two tuples t1 and t2 such that t1 = t2. Since the test t1 = t2 fails if any of the fields of t1 or t2 are null, it is possible for **unique** to be true even if there are multiple copies of a tuple, as long as at least one of the attributes of the tuple is null.

3.8.5 Subqueries in the From Clause

SQL allows a subquery expression to be used in the **from** clause. The key concept applied here is that any **select-from-where** expression returns a relation as a result and, therefore, can be inserted into another **select-from-where** anywhere that a relation can appear.

Consider the query "Find the average instructors' salaries of those departments where the average salary is greater than $42,000." We wrote this query in Section 3.7 by using the **having** clause. We can now rewrite this query, without using the **having** clause, by using a subquery in the **from** clause, as follows:

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 42000;
```

We note that nested subqueries in the **from** clause cannot use correlation variables from other relations in the **from** clause. However, SQL:2003 allows a subquery in the **from** clause that is prefixed by the **lateral** keyword to access attributes of preceding tables or subqueries in the **from** clause. For example, if we wish to print the names of each instructor, along with their salary and the average salary in their department, we could write the query as follows:

```
select name, salary, avg_salary
from instructor I1, lateral (select avg(salary) as avg_salary
                             from instructor I2
                             where I2.dept_name= I1.dept_name);
```

Without the **lateral** clause, the subquery cannot access the correlation variable *I1* from the outer query. Currently, only a few SQL implementations, such as IBM DB2, support the **lateral** clause.

3.8.6 The with Clause

The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs. Consider the following query, which finds those departments with the maximum budget.

```
with max_budget (value) as
    (select max(budget)
    from department)
select budget
from department, max_budget
where department.budget = max_budget.value;
```

The **with** clause defines the temporary relation *max_budget*, which is used in the immediately following query. The **with** clause, introduced in SQL:1999, is supported by many, but not all, database systems.

We could have written the above query by using a nested subquery in either the **from** clause or the **where** clause. However, using nested subqueries would have made the query harder to read and understand. The **with** clause makes the query logic clearer; it also permits a view definition to be used in multiple places within a query.

3.8.7 Scalar Subqueries

SQL allows subqueries to occur wherever an expression returning a value is permitted, provided the subquery returns only one tuple containing a single attribute; such subqueries are called scalar subqueries. For example, a subquery

can be used in the **select** clause as illustrated in the following example that lists all departments along with the number of instructors in each department:

```
select dept_name,
       (select count(*)
        from instructor
        where department.dept_name = instructor.dept_name)
       as num_instructors
from department;
```

The subquery in the above example is guaranteed to return only a single value since it has a **count(*)** aggregate without a **group by**. The example also illustrates the usage of correlation variables, that is, attributes of relations in the **from** clause of the outer query, such as *department.dept_name* in the above example.

Scalar subqueries can occur in **select, where,** and **having** clauses.

9. Modification of the Database

We have restricted our attention until now to the extraction of information from the

where salary is null;

The predicate is **not null** succeeds if the value on which it is applied is not null. Some implementations of SQL also allow us to test whether the result of a comparison is unknown, rather than true or false, by using the clauses is **unknow** and is not unknown.

7. AGGREGATE FUNCTIONS

Aggregate functions are functions that take a collection (a set or multiset) of value as input and return a single value. SQL offers five built-in aggregate functions:

- Average: avg
- Minimum: min
- Maximum: max
- Total: sum
- Count: count

The input to sum and avg must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.

6. NULL VALUES

Null values present special problems in relational operations, including arithmetic operations, comparison operations, and set operations.

The result of an arithmetic expression (involving, for example +, −, *, or /) is null if any of the input values is null. For example, if a query has an expression $r.A + 5$, and $r.A$ is null for a particular tuple, then the expression result must also be null for that tuple.

Comparisons involving nulls are more of a problem. For example, consider the comparison "1 < **null**". It would be wrong to say this is true since we do not know what the null value represents. But it would likewise be wrong to claim this expression is false; if we did, "**not** (1 < **null**)" would evaluate to true, which does not make sense. SQL therefore treats as **unknown** the result of any comparison involving a *null* value (other than predicates **is null** and **is not null**, which are described later in this section). This creates a third logical value in addition to *true* and *false*.

Since the predicate in a **where** clause can involve Boolean operations such as **and**, **or**, and **not** on the results of comparisons, the definitions of the Boolean operations are extended to deal with the value **unknown**.

8.NESTED SUBQUERIES

SQL provides a mechanism for nesting subqueries. A subquery is a select-fromwhere expression that is nested within another query. A common use of subqueries is to perform tests for set membership, make set comparisons, and determine set cardinality, by nesting subqueries in the where clause. we see how a class of subqueries called scalar subqueries can appear wherever an expression returning a value can occur.

3.8.1 Set Membership

SQL allows testing tuples for membership in a relation. The in connective tests for set membership, where the set is a collection of values produced by a select clause. The not in connective tests for the absence of set membership.

As an illustration, reconsider the query "Find all the courses taught in the both the Fall 2009 and Spring 2010 semesters." Earlier, we wrote such a query by intersecting two sets: the set of courses taught in Fall 2009 and the set of courses taught in Spring 2010. We can take the alternative approach of finding all courses that were taught in Fall 2009 and that are also members of the set of courses taught in Spring 2010. We begin by finding all courses taught in Spring 2010, and we write the subquery

```
(select course_ id
 from section
 where semester = 'Spring' and year= 2010)
```

We then need to find those courses that were taught in the Fall 2009 and that appear in the set of courses obtained in the subquery. We do so by nesting the subquery in the where clause of an outer query. The resulting query is :

```
select distinct course_ id
 from section
 where semester = 'Fall' and year= 2009 and
 course id in (select course id
 from section
 where semester = 'Spring' and year= 2010);
```

We use the not in construct in a way similar to the in construct. For example, to find all the courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we can write:

```
select distinct course_ id
from section
where semester = 'Fall' and year= 2009 and
course_ id not in
 (select course_ id
 from section
where semester = 'Spring' and year= 2010);
```

3.8.2 Set Comparison

As an example of the ability of a nested subquery to compare sets, consider the query "Find the names of all instructors whose salary is greater than at least one instructor in the Biology department." we wrote this query as follows:

```
select distinct T.name
 from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

SQL does, however, offer an alternative style for writing the preceding query. The phrase "greater than at least one" is represented in SQL by > some. This construct allows us to rewrite the query in a form that resembles closely our formulation of the query in English.

```
select name
from instructor
where salary > some (select salary
                     from instructor
                    where dept name = 'Biology');
```

The subquery generates the set of all salary values of all instructors in the Biology department. The > some comparison in the where clause of the outer select is true

if the salary value of the tuple is greater than at least one member of the set of all salary values for instructors in Biology.

SQL also allows < some, <= some, >= some, = some, and <> some comparisons. As an exercise, verify that = some is identical to in, whereas <> some is not the same as not in.

The construct > all corresponds to the phrase "greater than all." Using this construct, we write the query as follows:

```
select name
from instructor
where salary > all (select salary
from instructor
where dept name = 'Biology');
```

As it does for some, SQL also allows < all, <= all, >= all, = all, and <> all comparisons. As an exercise, verify that <> all is identical to not in, whereas = all is not the same as in.

3.8.3 Test for Empty Relations

SQL includes a feature for testing whether a subquery has any tuples in its result. The exists construct returns the value true if the argument subquery is nonempty. Using the exists construct, we can write the query "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester" in still another way:

```
select course_ id
from section as S
where semester = 'Fall' and year= 2009 and
exists (select *
        from section as T
        where semester = 'Spring' and year=
2010 and S.course _id= T.course_ id);
```

A subquery that uses a correlation name from an outer query is called a **correlated subquery.**

We can test for the nonexistence of tuples in a subquery by using the **not exists** construct. We can use the **not exists** construct to simulate the set containment (that is, superset) operation: We can write "relation A contains relation B" as "**not exists** (B **except** A)." To illustrate the **not exists** operator, consider the query "Find all students who have taken all courses offered in the Biology department." Using the **except** construct, we can write the query as follows:

```
select distinct S.ID, S.name
from student as S
where not exists ((select course_id
from course
where dept_name = 'Biology')
except
(select T.course_id
from takes as T
where S.ID = T.ID));
```

Here, the subquery finds the set of all courses offered in the Biology department. The subquery finds all the courses that student S.ID has taken. Thus, the outer select takes each student and tests whether the set of all courses that the student has taken contains the set of all courses offered in the Biology department.

3.8.4 Test for the Absence of Duplicate Tuples

SQL includes a boolean function for testing whether a subquery has duplicate tuples in its result. The **unique** construct returns the value true if the argument subquery contains no duplicate tuples. Using the unique construct, we can write the query "Find all courses that were offered at most once in 2009" as follows:

```
select T.course_id
from course as T
where unique (select R.course_id
from section as R
where T.course_id= R.course_id and R.year = 2009);
```

Note that if a course is not offered in 2009, the subquery would return an empty result, and the **unique** predicate would evaluate to true on the empty set. An equivalent version of the above query not using the **unique** construct is:

```
select T.course_id
from course as T
where 1 <= (select count(R.course_id)
            from section as R
            where T.course_id= R.course_id and
                  R.year = 2009);
```

We can test for the existence of duplicate tuples in a subquery by using the **not unique** construct. To illustrate this construct, consider the query "Find all courses that were offered at least twice in 2009" as follows:

```
select T.course_id
from course as T
where not unique (select R.course_id
                  from section as R
                  where T.course_id= R.course_id and
                        R.year = 2009);
```

Formally, the **unique** test on a relation is defined to fail if and only if the relation contains two tuples t1 and t2 such that t1 = t2. Since the test t1 = t2 fails if any of the fields of t1 or t2 are null, it is possible for **unique** to be true even if there are multiple copies of a tuple, as long as at least one of the attributes of the tuple is null.

3.8.5 Subqueries in the From Clause

SQL allows a subquery expression to be used in the **from** clause. The key concept applied here is that any **select-from-where** expression returns a relation as a result and, therefore, can be inserted into another **select-from-where** anywhere that a relation can appear.

Consider the query "Find the average instructors' salaries of those departments where the average salary is greater than $42,000." We wrote this query in Section 3.7 by using the **having** clause. We can now rewrite this query, without using the **having** clause, by using a subquery in the **from** clause, as follows:

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 42000;
```

We note that nested subqueries in the **from** clause cannot use correlation variables from other relations in the **from** clause. However, SQL:2003 allows a subquery in the **from** clause that is prefixed by the **lateral** keyword to access attributes of preceding tables or subqueries in the **from** clause. For example, if we wish to print the names of each instructor, along with their salary and the average salary in their department, we could write the query as follows:

```
select name, salary, avg_salary
from instructor I1, lateral (select avg(salary) as avg_salary
                             from instructor I2
                             where I2.dept_name= I1.dept_name);
```

Without the **lateral** clause, the subquery cannot access the correlation variable *I1* from the outer query. Currently, only a few SQL implementations, such as IBM DB2, support the **lateral** clause.

3.8.6 The with Clause

The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs. Consider the following query, which finds those departments with the maximum budget.

```
with max_budget (value) as
    (select max(budget)
    from department)
select budget
from department, max_budget
where department.budget = max_budget.value;
```

The **with** clause defines the temporary relation *max_budget*, which is used in the immediately following query. The **with** clause, introduced in SQL:1999, is supported by many, but not all, database systems.

We could have written the above query by using a nested subquery in either the **from** clause or the **where** clause. However, using nested subqueries would have made the query harder to read and understand. The **with** clause makes the query logic clearer; it also permits a view definition to be used in multiple places within a query.

3.8.7 Scalar Subqueries

SQL allows subqueries to occur wherever an expression returning a value is permitted, provided the subquery returns only one tuple containing a single attribute; such subqueries are called scalar subqueries. For example, a subquery

can be used in the **select** clause as illustrated in the following example that lists all departments along with the number of instructors in each department:

```
select dept_name,
        (select count(*)
          from instructor
          where department.dept_name = instructor.dept_name)
        as num_instructors
from department;
```

The subquery in the above example is guaranteed to return only a single value since it has a **count**(*) aggregate without a **group by**. The example also illustrates the usage of correlation variables, that is, attributes of relations in the **from** clause of the outer query, such as *department.dept_name* in the above example.

Scalar subqueries can occur in **select, where,** and **having** clauses.

9. Modification of the Database

We have restricted our attention until now to the

Online Analytical processing uses complex Queries to analyse aggregated historical data from OLTP systems.

Online captures, stores and process data from transaction in real time.

Features	OLTP	OLAP
(i) characteristics	Handles a large no. of small transactions	Handles large volumes of data with complex Queries
(ii) Query types	Simple standardized Queries.	Complex Queries.
(iii) Response time	milli seconds.	seconds or, minutes, hours depending on the amount of data to Process.
(iv) Design	Industry specific, such as retail, manufacturing or banking.	subject specific, such as sales, inventory or marketing
(v) Source	Transactions	Aggregate data from transactions.
(vi) Purpose	Control and run essential bussiness operations in real time	Plan, solve problems, support decision, discover hidden insites
(vii) Data updates	Short, fast updates initiated by user	Data periodically refreshed with schedule of long running batch jobs

vii. Space requirements	Generally small if historical data is archeive.	Generally large due to aggregating large data sets
viii) Backup & recovery	Regular backups requir -ed. (last ~~data can be reloaded from OLTP database as needed~~) x	~~Productivity~~ last data can be reloaded from OLTP database as needed
x, Productivity	increase productivity of users	increase productivity of bussiness manger and data analyst
xi, Data view	list day-to-day bussiness transactions	Multidimensional view of enterprise data
xii, User example	Chile shoppers,	Data Analyst
xiii, Data base design	normalised design.	

```
CREATE TABLE Employee
(
 Empno    number(10),
 Name     varchar2(20),
 Empcity  varchar2(10),
 Primary key(empno),
 foreign key(empno) ~~references~~.
);
desc employee
```

Name	Null &	Type.
Empno		Number(10)
Name		varchar2(20)
Empcity		varchar2(20)

```
CREATE TABLE company
(
  Empno  number(10),
  companyname varchar2(20),
  Salary      number(10),
  Primary key (Empno),
  foreign key(Empno) references Employee
);
desc company
```

Name	Null &	Type
Empno		Number(10)
company name		varchar2(20)
Salary		Number (10)

```
Insert All
Into employee (Empno, Name, Empcity) values (5421, 'Ram', 'chennai')
Into employee (Empno, Name, Empcity) values (5821, 'Hari', 'Nellore')
Into employee (Empno, Name, Empcity) values (5351, 'Kavya', 'chennai')
Into employee (Empno, Name, Empcity) values (5451, 'Yash', 'Guntur')
Select 1 from dual;
```

Empno	Name	Empcity
5421	Ram	chennai
5821	Hari	Nellore
5351	Kavya	chennai
5451	Yash	Guntur

```
Insert All
Into company (Empno, Company name, Salary)
    values (5421, 'XYZ', 50,000)
Into company (Empno, company name, salary)
    values (5821, 'ABC', 70000)
Into company (Empno, company name, salary)
    values (5351, 'XYZ', 75000)
Into company (Empno, company name, salary)
    values (5451, 'EFG', 80000)
Select 1 from dual;
```

Empno	companyname	salary
5421	XYZ	50000
5821	ABC	70000
5351	XYZ	75000
5451	EFG	80000

```
select name, company name
foram Employee, company;
```

Name	company name
Ram.	XYZ
Hari	ABC
Kavya	XYZ
Yash.	EFG

```
select name, empcity, company name, salary
from employee, company
where salary > 10000;
```

Name	Empcity	company name	salary.
Ram	chennai	XYZ	50000
Hari	Nellore	ABC	70000
Kavya	Chennai	XYZ	75000
Yash	Guntur	EFG	80000

```
Select Name
From employee
Where company name = 'xyz';
```

Name
Ram
Kavya

For instance, consider a banking application, where we need to update from one bank account to another in the same bank. To do so, we need to update two account balances, subtracting the amount transferred from one, and adding it to the other. If the system crashes after subtracting the amount from the first account, but before adding it to the second account, the bank balances would be inconsistent. A similar problem would occur, if the second account is credited before subtracting the amount from the first account, and the system crashes just after crediting the amount

By either committing the actions of a transaction after all its steps are completed, or rolling back all its actions in case the transaction could not complete all its actions successfully, the database provides an abstraction of a transaction as being **atomic**, that is, indivisible. Either all the effects of the transaction are reflected in the database, or none are (after rollback)

4. Integrity Constraints

Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database.

Examples of integrity constraints are:

- An instructor name cannot be null.
- No two instructors can have the same instructor ID.

• Every department name in the course relation must have a matching department name in the department relation.

• The budget of a department must be greater than $0.00

In general, an integrity constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, most database systems allow one to specify integrity constraints that can be tested with minimal overhead.

Another form of integrity constraint, called **functional dependencies**, that is used primarily in the process of schema design.

4.4.1 Constraints on a Single Relation

We described in Section 3.2 how to define tables using the **create table** command. The **create table** command may also include integrity-constraint statements. In addition to the primary-key constraint, there are a number of other ones that can be included in the **create table** command. The allowed integrity constraints include

- **not null**
- **unique**
- **check**(<predicate>)

We cover each of these types of constraints in the following sections.

4.4.2 Not Null Constraint

For certain attributes, however, null values may be inappropriate. Consider a tuple in the student relation where name is null. Such a tuple gives student information for an unknown student; thus, it does not contain useful information. Similarly, we would not want the department budget to be null. In cases such as this, we wish to forbid null values, and we can do so by restricting the domain of the attributes name and budget to exclude null values, by declaring it as follows

name **varchar**(20) not null

budget **numeric**(12,2) not null

The not null specification prohibits the insertion of a null value for the attribute. Any database modification that would cause a null to be inserted in an attribute declared to be not null generates an error diagnostic.

4.4.3 Unique Constraint:

SQL also supports an integrity constraint:

unique (A_{j1} , A_{j2} ,..., A_{jm})

The **unique** specification says that attributes A_{j1} , A_{j2} ,..., A_{jm} form a candidate ke
that is, no two tuples in the relation can be equal on all the listed attributes.
However, candidate key attributes are permitted to be null unless they have
explicitly been declared to be not null. Recall that a null value does not equal an
other value.

4.4.4 The check Clause

When applied to a relation declaration, the clause **check**(P) specifies a predicat
P that must be satisfied by every tuple in a relation.

A common use of the **check** clause is to ensure that attribute values satisf
specified conditions, in effect creating a powerful type system. For instance,
clause **check** (*budget* > 0) in the **create table** command for relation *departmen*
would ensure that the value of **budget** is nonnegative.

As another example, consider the following:

```
create table section
  (course_id      varchar (8),
   sec_id         varchar (8),
   semester       varchar (6),
   year           numeric (4,0),
   building       varchar (15),
   room_number    varchar (7),
   time_slot_id   varchar (4),
   primary key (course_id, sec_id, semester, year),
   check (semester in ('Fall', 'Winter', 'Spring', 'Summer')));
```

Here, we use the **check** clause to simulate an enumerated type, by specifying that
semester must be one of 'Fall', 'Winter', 'Spring', or 'Summer'. Thus, the check
clause permits attribute domains to be restricted in powerful ways that most
programming-language type systems do not permit.

4.4.5 Referential Integrity:

Often, we wish to ensure that a value that appears in one relation for a given set
attributes also appears for a certain set of attributes in another relation. This
condition is called **referential integrity**.

Foreign keys can be specified as part of the SQL create table statement by using
the **foreign key** clause. We illustrate foreign-key declarations by using the SQL

QL definition of part of our university database, shown in Figure 4.8. The finition of the course table has a declaration "**foreign key**(dept_name)**references** partment". This foreign-key declaration specifies that for each course tuple, the partment name specified in the tuple must exist in the department relation. ithout this constraint, it is possible for a course to specify a nonexistent partment name.

y default, in SQL a foreign key references the primary-key attributes of the eferenced table. SQL also supports a version of the references clause where a list f attributes of the referenced relation can be specified explicitly. The specified list f attributes must, however, be declared as a candidate key of the referenced elation, using either a primary key constraint, or a unique constraint.

We can use the following short form as part of an attribute definition to declare hat the attribute forms a foreign key:

dept_ name varchar(20) references department

```
create table classroom
    (building       varchar (15),
     room_number    varchar (7),
     capacity       numeric (4,0),
     primary key (building, room_number))

create table department
    (dept_name      varchar (20),
     building       varchar (15),
     budget         numeric (12,2) check (budget > 0),
     primary key (dept_name))

create table course
    (course_id      varchar (8),
     title          varchar (50),
     dept_name      varchar (20),
     credits        numeric (2,0) check (credits > 0),
     primary key (course_id),
     foreign key (dept_name) references department)

create table instructor
    (ID             varchar (5),
     name           varchar (20), not null
     dept_name      varchar (20),
     salary         numeric (8,2), check (salary > 29000),
     primary key (ID),
     foreign key (dept_name) references department)

create table section
    (course_id      varchar (8),
     sec_id         varchar (8),
     semester       varchar (6), check (semester in
                                ('Fall', 'Winter', 'Spring', 'Summer'),
     year           numeric (4,0), check (year > 1759 and year < 2100)
     building       varchar (15),
     room_number    varchar (7),
     time_slot_id   varchar (4),
     primary key (course_id, sec_id, semester, year),
     foreign key (course_id) references course,
     foreign key (building, room_number) references classroom)
```

Figure 4.8 SQL data definition for part of the university database.

```
create table course
    (...
    foreign key (dept_name) references department
                on delete cascade
                on update cascade,
    ...);
```

Because of the clause **on delete cascade** associated with the foreign-key declaration, if a delete of a tuple in *department* results in this referential-integrity constraint being violated, the system does not reject the delete. Instead, the delete "cascades" to the *course* relation, deleting the tuple that refers to the department that was deleted. Similarly, the system does not reject an update to a field referenced by the constraint if it violates the constraint; instead, the system updates the field *dept_name* in the referencing tuples in *course* to the new value as well. SQL also allows the **foreign key** clause to specify actions other than **cascade**, if the constraint is violated: The referencing field (here, *dept_name*) can be set to *null* (by using **set null** in place of **cascade**), or to the default value for the domain (by using **set default**).

4.4.6 Integrity Constraint Violation During a Transaction

Transactions may consist of several steps, and integrity constraints may be violated temporarily after one step, but a later step may remove the violation. For instance, suppose we have a relation person with primary key name, and an attribute spouse, and suppose that spouse is a foreign key on person. That is, the constraint says that the spouse attribute must contain a name that is present in the person table. Suppose we wish to note the fact that John and Mary are married to each other by inserting two tuples, one for John and one for Mary, in the above relation, with the spouse attributes set to Mary and John, respectively. The insertion of the first tuple would violate the foreign-key constraint, regardless of which of the two tuples is inserted first. After the second tuple is inserted the foreign-key constraint would hold again.

To handle such situations, the SQL standard allows a clause **initially deferred** to be added to a constraint specification; the constraint would then be checked at the end of a transaction, and not at intermediate steps. A constraint can alternatively be specified as **deferrable**, which means it is checked immediately by default, but can be deferred when desired. For constraints declared as deferrable, executing a statement **set constraints** constraint-list **deferred** as part of a transaction causes the checking of the specified constraints to be deferred to the end of that transaction.

4.4.7 Complex Check Conditions and Assertions

As defined by the SQL standard, the predicate in the **check** clause can be a arbitrary predicate, which can include a subquery. If a database implementation supports subqueries in the **check** clause, we could specify the following referential-integrity constraint on the relation section:

check (time_ slot_ id in (select time_ slot_ id from time_ slot))

The check condition verifies that the time slot id in each tuple in the section relation is actually the identifier of a time slot in the time slot relation. Thus, the condition has to be checked not only when a tuple is inserted or modified in section, but also when the relation time slot changes (in this case, when a tuple is deleted or modified in relation time slot).

An **assertion** is a predicate expressing a condition that we wish the database always to satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. We have paid substantial attention to these forms of assertions because they are easily tested and apply to a wide range of database applications. However, there are many constraints that we cannot express by using only these special forms. Two examples of such constraints are:

- For each tuple in the student relation, the value of the attribute tot cred must equal the sum of credits of courses that the student has completed successfully.
- An instructor cannot teach in two different classrooms in a semester in the same time slot.1
- An assertion in SQL takes the form:

  create assertion <assertion_name>check<predicate> ;

Currently, none of the widely used database systems supports either subqueries in the check clause predicate, or the create assertion construct. However, equivalent functionality can be implemented using triggers

5. SQL Data Types and Schemas

A number of built-in

- Advantage and Disadvantage

What is a Trigger?

Triggers are automatically executed in response to certain events on a particular table. These are used to maintain the integrity of the data. A trigger in SQL codes that are SQL works similar to a real-world trigger. For example, when the gun trigger is pulled a bullet is fired. We all know this, but how this is related to Triggers in SQL? To understand this let's consider a hypothetical situation.

Advantages

- Forcing **security** approvals on the table that are present in the database
- Triggers provide another way to check the **integrity of data**
- **Counteracting invalid** exchanges
- Triggers **handle errors** from the database layer
- Normally triggers can be useful for **inspecting the data** changes in tables
- Triggers give an alternative way to run **scheduled tasks**. Using triggers, we don't have to wait for the scheduled events to run because the triggers are invoked

automatically before or after a change is made to the data in a table

Disadvantages

- Triggers can only provide extended **validations**, i.e, not all kinds of validations. For simple validations, you can use the NOT NULL, UNIQUE, CHECK, and FOREIGN KEY constraints
- Triggers may increase the **overhead** of the database
- Triggers can be difficult to **troubleshoot** because they execute automatically in the database, which may not invisible to the client applications