CHAPTER

5

Advanced SQL

Objectives

- To write queries using nested select statements.
- To write quires using joins.
- To use table aliases to join same tables.
- To use the in, all, any, exists and unique operators for conditions involving sets.
- To understand the various join operations: natural join, inner join, left outer join, right outer join, and full outer join.
- To use the union, union all, intersect, and minus set operators.
- To create and use views.
- To understand the Oracle rowId and rowNum columns.
- To create and use indexes and clusters.
- To create and use database links for distributed quires.
- To create and use synonyms for database objects.
- To create and use sequences.

5.1 Introduction

In the preceding chapter, you learned simple SQL statements, SQL functions, the order by clause, the group by clause, and the having clause. The SQL statements in the preceding chapter use one select clause that involves a single table.

This chapter introduces nested queries and the queries that involve multiple tables. You will also learn how to create and use views, indexes, clusters, sequences, and database links.

5.2 Nested Queries

A select clause can be nested inside another select clause. This is known as *nested queries or subqueries*. Sometimes, a simple select statement cannot get the query. You have to use nested queries.

Example 5.1: Find the faculty with the highest salary.

You may attempt to solve the query using the following statement:

```
select lastName || ', ' || firstName as "Name", salary
from Faculty
where salary = max(salary);
```

This is erroneous because the aggregate function max cannot be used in the where clause. To fix it, use a nested query as follows:

```
select lastName || ', ' || firstName as "Name", salary
from Faculty
where salary = (select max(salary) from Faculty);
```

The result is shown in Figure 5.1.

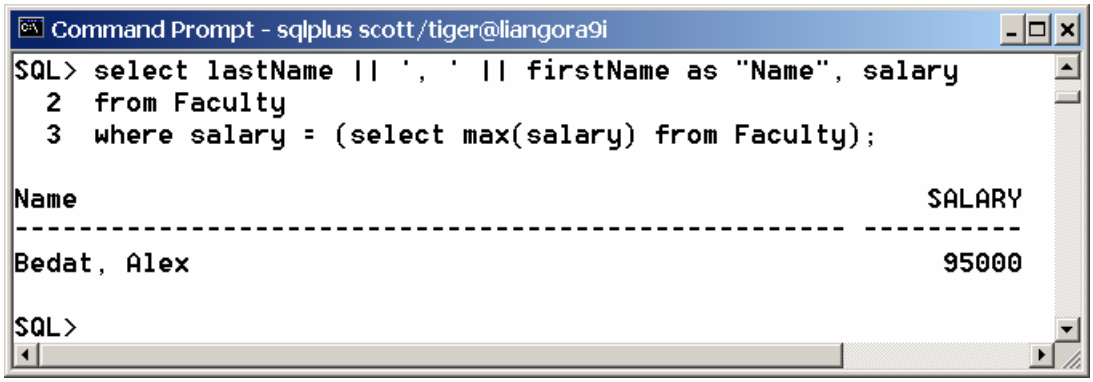

Figure 5.1

Example 5.1 demonstrates the nested queries.

Example 5.2: Find the department with the largest total salary. The result is shown in Figure 5.2.

```
select deptId, sum(salary)
from Faculty
```

```
group by deptId
having sum(salary) =
    (select max(sum(salary)) from Faculty group by deptId);
```

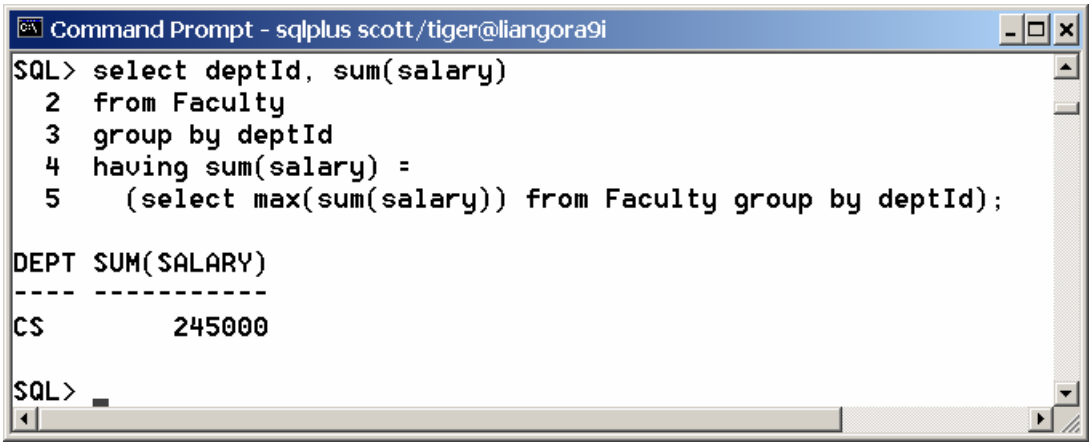

Figure 5.2

Example 5.2 demonstrates the nested queries with the group by and having clauses.

5.3 Joining Tables

Often you need to get the information from multiple tables as demonstrated in the following query.

Example 5.3: List courses taken by student Jacob Smith. To solve this query, you need to join tables Student and Enrollment as shown in Figure 5.3.

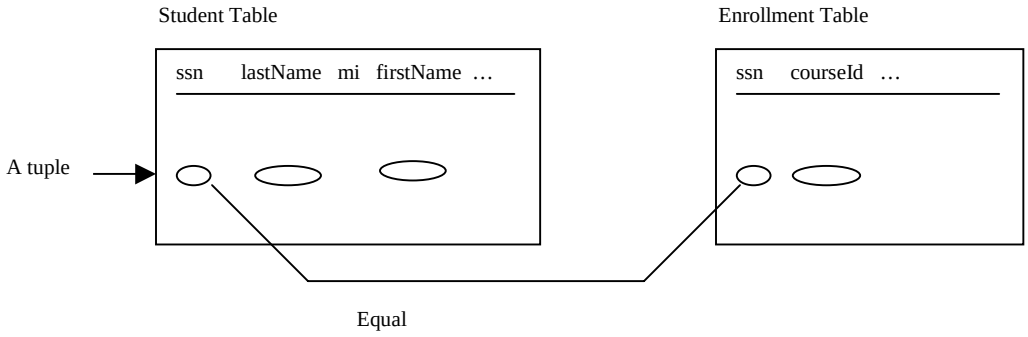

Figure 5.3

Student and Enrollment are joined on ssn.

You can write the query in SQL as follows:

```
select distinct lastName || ', ' || firstName as "Name", courseId
from Student, Enrollment
where Student.ssn = Enrollment.ssn and
  lastName = 'Smith' and firstName = 'Jacob';
```

The tables Student and Enrollment are listed in the from clause. The query examines all pairs of rows, one from Student and the other from Enrollment and selects the pairs that satisfy the condition in the where clause. The rows in Student have last name Smith and first name Jacob and both rows from Student and Enrollment have the same ssn values. For each pair selected, lastName and firstName from Student and courseId from Enrollment are used to produce the result, as shown in Figure 5.4.

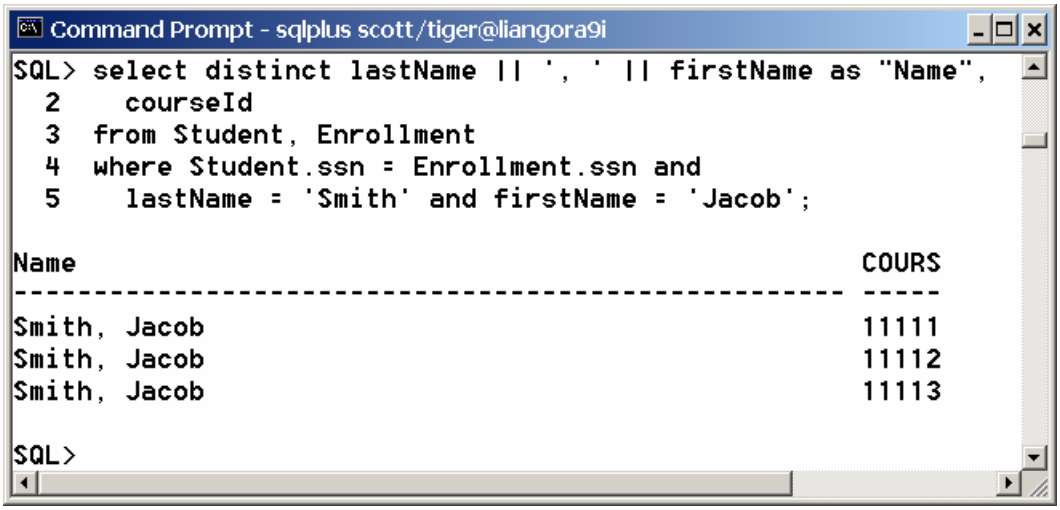

Figure 5.4

Example 5.3 demonstrates queries involving multiple tables.

Student and Enrollment have the same attribute ssn. To distinguish them in a query, use Student.ssn and Enrollment.ssn.

When joining two tables without the where clause, the result is a Cartesian product, i.e. all pairs of two rows from each table are selected.

> NOTE: SQL2 introduced the natural join and join, operators that can be used in the from clause to specify a join of two tables. The syntax *A natural join B* joins A with B on their common attributes. The syntax *A join B on condition* joins A with B with a specified condition. Example 5.3 can be rewritten using the natural join or join operators as follows:

```
select distinct lastName || ', ' || firstName as "Name", courseId
from Student natural join Enrollment
where lastName = 'Smith' and firstName = 'Jacob';

select distinct lastName || ', ' || firstName as "Name", courseId
```

```
from Student join Enrollment on Student.ssn = Enrollment.ssn and
  lastName = 'Smith' and firstName = 'Jacob';
```

******End of NOTE***

5.4 Tuple Aliases

In the preceding query, you distinguish attributes by prefixing table names before the attributes using the *dot* notation (e.g. Student.ssn). Occasionally, you need to join rows from the same table, as in the following query.

Example 5.4: Find all pairs of students with the same phone number. List the students and their common phone number. To solve this query, you need to join tables Student with Student as shown in Figure 5.5.

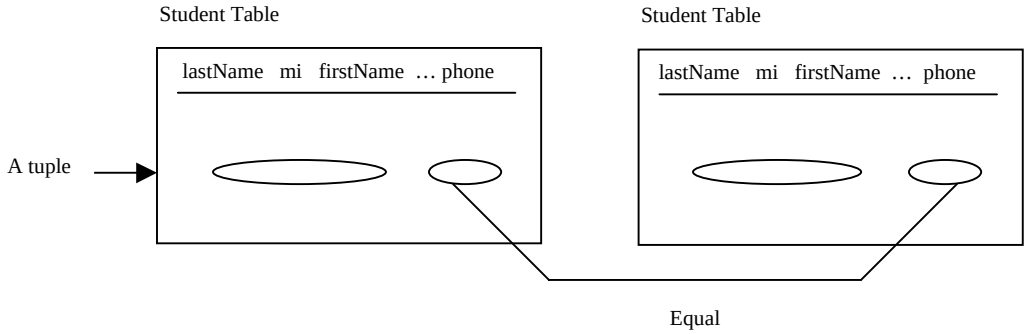

Figure 5.5

Student is joined with Student on the attribute phone.

To distinguish the rows from both Student tables, use table aliases. You can alias the first Student table as s1 and the second as s2. The query can be written as follows:

```
select s1.firstName || ' ' || s1.lastName as "Student 1",
  s2.firstName || ' ' || s2.lastName as "Student 2", s1.phone
from Student s1, Student s2
where s1.phone = s2.phone and s1.ssn < s2.ssn;
```

s1 and s2 are the aliases for Student. The query examines all pairs of rows, one from s1 and the other from s2 and selects the pairs whose rows have the same phone number. For each pair selected, lastName and firstName from s1 and s2 are used to produce the result, as shown in Figure 5.6.

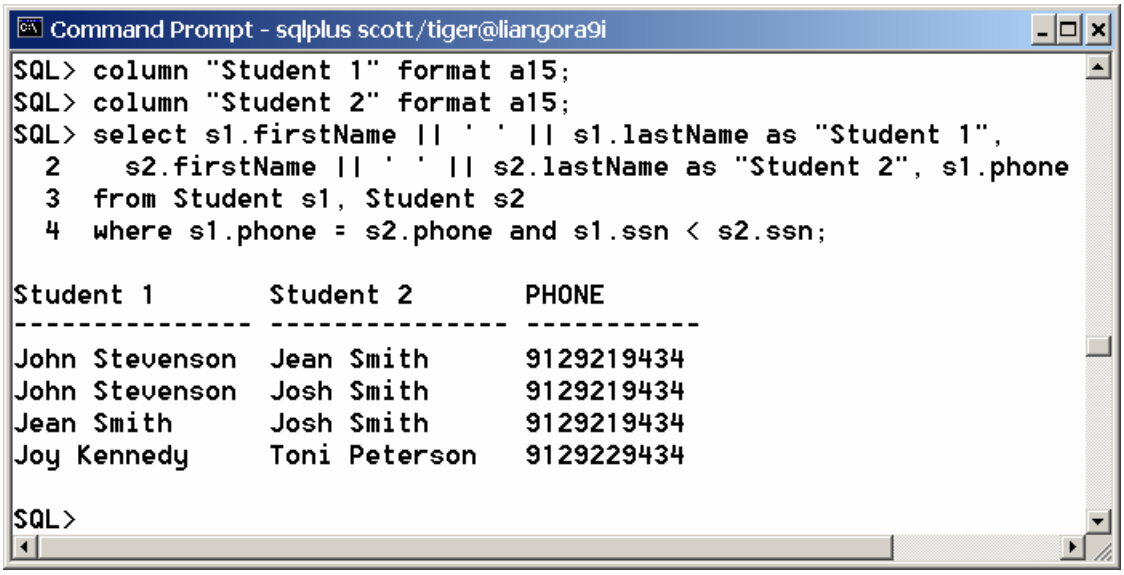

Figure 5.6

Example 5.4 demonstrates quires involving join of same tables and using table aliases.

> NOTE: The where clause has another condition s1.ssn < s2.ssn. Without this condition, two students (e.g. Joy Kennedy and Toni Peterson) with the same phone number would appear in two pairs (Joy Kennedy, Toni Peterson, 9129229434) and (Toni Peterson, Joy Kennedy, 9129229434).

5.5 Conditions Involving Sets

SQL provides the operators *in*, *all*, *any*, *exists*, and *unique* that checks values in a set to produce a Boolean result.

- Expression e in (S) checks whether value e is in set S.
- Expression e *op* all (S) uses the comparison operator op (=, <, <=, >, >=, <>) to compare e with every value in S. The result is true if and only if every comparison evaluates to true.
- Expression e *op* any (S) uses the comparison operator op (=, <, <=, >, >=, <>) to compare e with every value in S. The result is true if one of the comparisons evaluates to true.
- Expression exists (S) returns true if and only if S is not empty.
- Expression unique (S) returns true if and only if S has

no duplicate tuples. The unique condition is currently not supported in Oracle. However, you can write queries without using the unique condition.

NOTE: All the Boolean expressions can be negated by putting the not keyword in front of the expression.

NOTE: You can use e op S provided that S has a single value. If S is a query that results in more than one value, an error would occur.

Example 5.5: Find all faculty in the Computer Science and Mathematics departments. The query result is shown in Figure 5.7.

```
select firstName || ' ' || lastName as "CS/MATH Faculty"
from Faculty
where deptId in (
  select deptId
  from Department
  where name = 'Computer Science' or name = 'Mathematics');
```

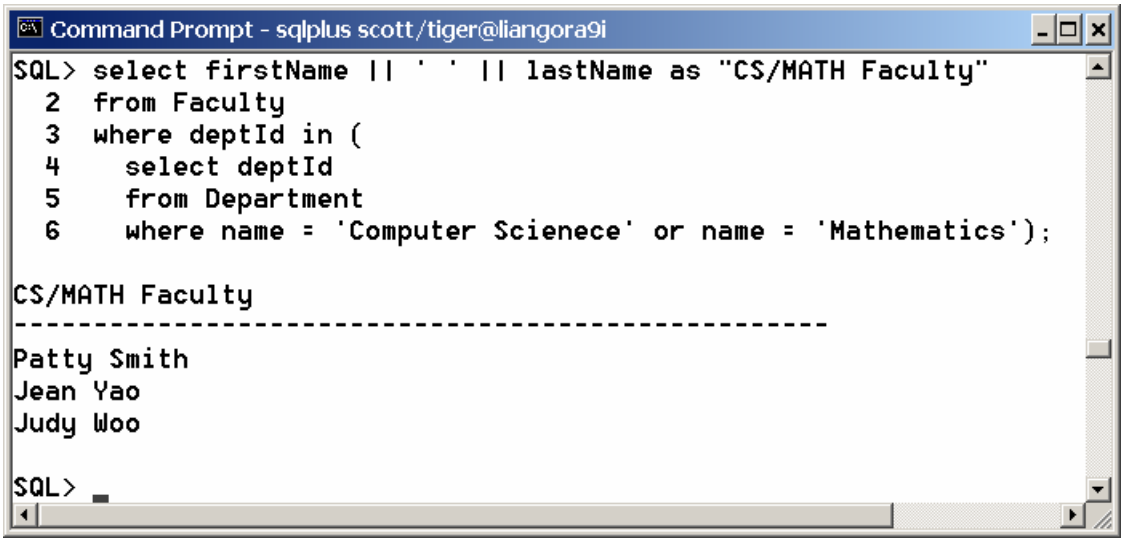

Figure 5.7

Example 5.5 demonstrates the in operator.

Many quires involving the in operator are nested queries. They can often be rewritten using joins. Example 5.5 can be rewritten using joins as follows:

```
select firstName || ' ' || lastName as "CS/MATH Faculty"
from Faculty, Department
where Faculty.deptId = Department.deptId and
  (name = 'Computer Science' or name = 'Mathematics');
```

There are many ways to write queries. All solutions are acceptable and there is no difference in the performance.

SQL is a non-procedural language. Each query is translated to procedures and optimized by the DBMS. The advantage of using joins is that you can display attributes from both tables. For example, you can display department name in the query as follows:

```
select firstName || ' ' || lastName as "CS/MATH Faculty",
  Department.name
from Faculty, Department
where Faculty.deptId = Department.deptId and
  (name = 'Computer Science' or name = 'Mathematics');
```

Example 5.6: Find all faculty whose salary is greater than the salary of all the faculty in the MATH department. The query result is shown in Figure 5.8.

```
select firstName || ' ' || lastName as "Faculty"
from Faculty
where salary > all (
  select salary
  from Faculty
  where deptId = 'MATH');
```

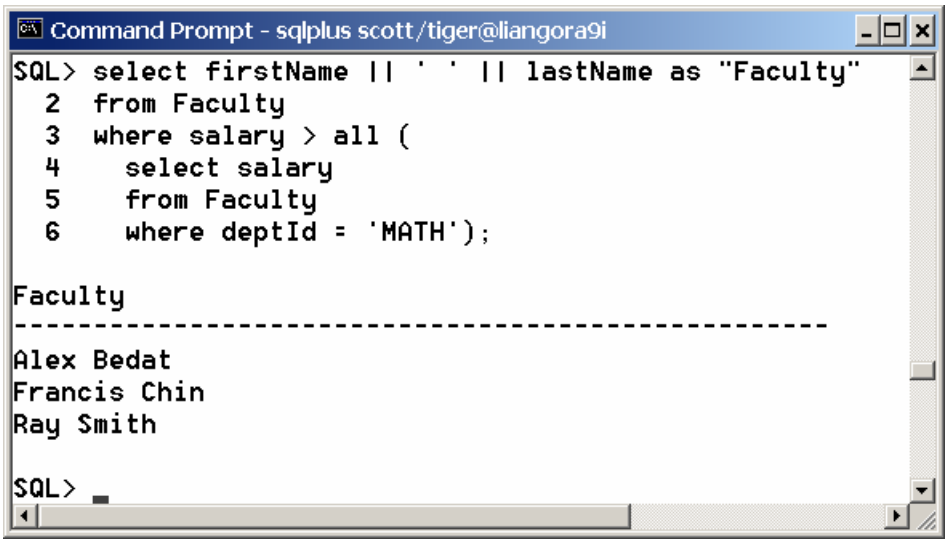

Figure 5.8

Example 5.6 demonstrates the all operator.

This query is equivalent to the following:

```
select firstName || ' ' || lastName as "Faculty"
from Faculty
where salary > (select max(salary)
  from Faculty
  where deptId = 'MATH');
```

Example 5.7: Find all students who take at least one course taught by Alex Bedat. List the student names and course titles. The query result is shown in Figure 5.9.

```
select firstName || ' ' || lastName as "Student", Course.title
from Student, Enrollment, Course
where Course.courseId = Enrollment.courseId and
  Student.ssn = Enrollment.ssn and Enrollment.courseId = any (
  select courseId
  from TaughtBy, Faculty
  where Faculty.ssn = TaughtBy.ssn and
  firstName = 'Alex' and lastName = 'Bedat')
order by lastName;
```

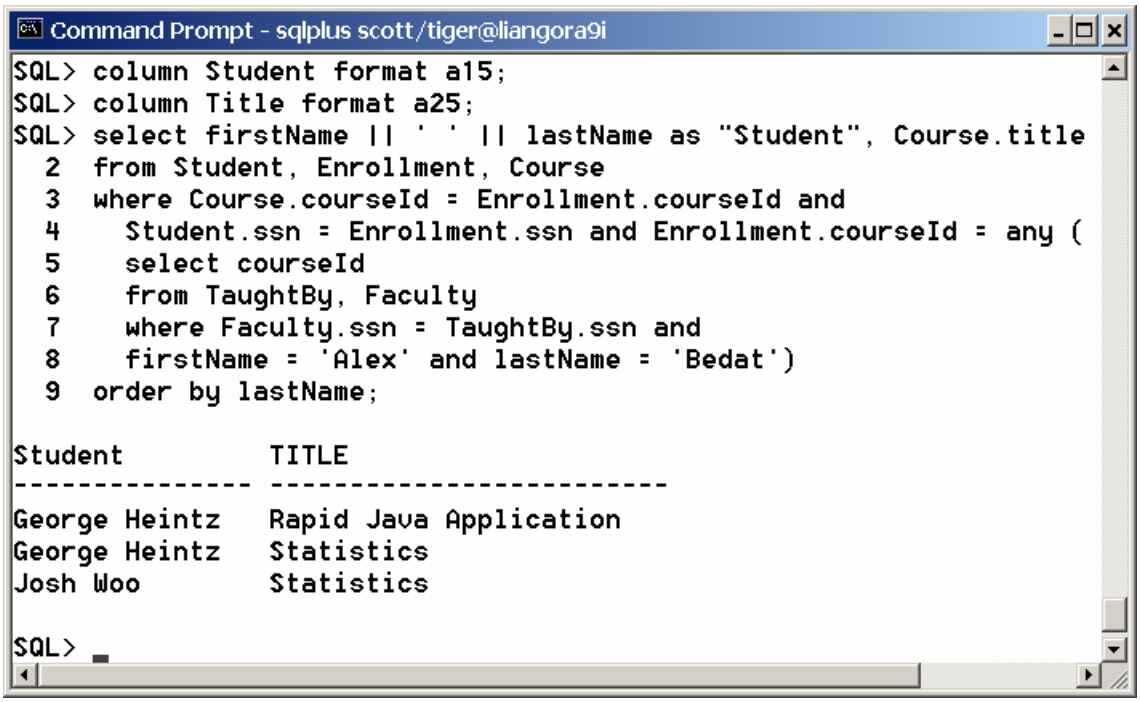

Figure 5.9

Example 5.7 demonstrates the any operator.

> NOTE: Expression e = any S is equivalent to e in S. So the previous query can be written as

```
select firstName || ' ' || lastName as "Student", Course.title
from Student, Enrollment, Course
where Course.courseId = Enrollment.courseId and
  Student.ssn = Enrollment.ssn and Enrollment.courseId in (
  select courseId
  from TaughtBy, Faculty
  where Faculty.ssn = TaughtBy.ssn and
  firstName = 'Alex' and lastName = 'Bedat')
order by lastName;
```

******End of NOTE***

Example 5.8: Find all faculty who currently don't teach any courses. The query result is shown in Figure 5.10.

```
select distinct firstName || ' ' || lastName as "Faculty"
from Faculty f
where not exists (
  select courseId
  from TaughtBy
```

```
  where f.ssn = TaughtBy.ssn);
```

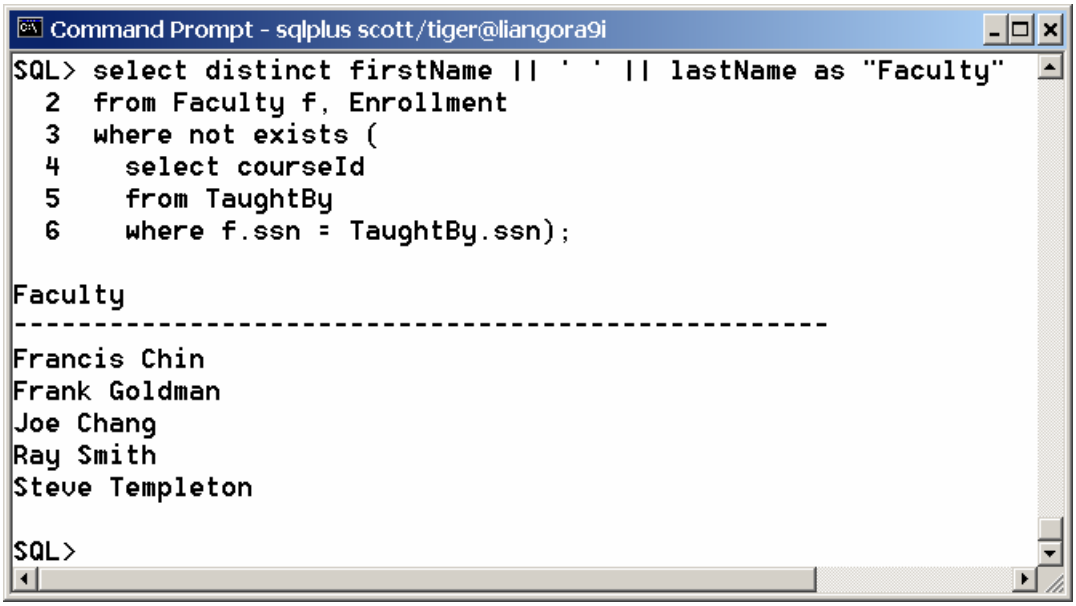

Figure 5.10

Example 5.8 demonstrates the exist operator.

Example 5.9: Find the department with unique faculty last names.

```
select Department.name
from Dapartment d
where unique (
  select lastName
  from Faculty
  where d.deptId = Faculty.deptId);
```

The unique condition currently not supported in Oracle. However, you can write this query without using the unique condition as follows:

```
select name
from Department d
where (select count(lastName)
  from Faculty
  where d.deptId = Faculty.deptId)
  = (select count(distinct lastName)
  from Faculty
  where d.deptId = Faculty.deptId);
```

The output the query is shown in Figure 5.11. Note that the CS department has two faculty Frank Goldman and Kim Goldman with the same last name. So, the CS department is not selected in the query.

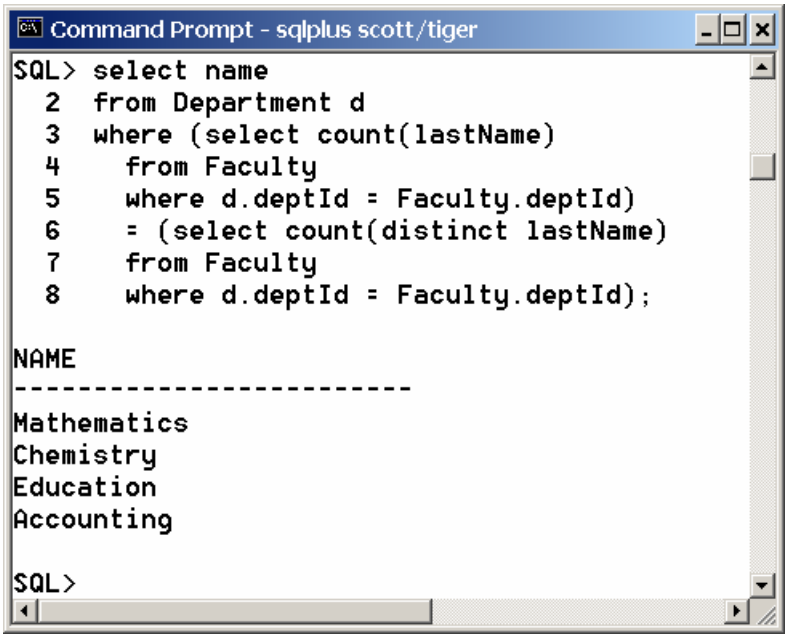

Figure 5.11

Example 5.9 demonstrates quires without using the unique keyword.

5.6 Outer Join (Optional)

A join of two tables returns only those rows that satisfy the join condition. This is known as *inner join* or *simple join*. Sometimes it is convenient to list all tuples from one or both tables in the result even though they are not matched in the inner join. SQL 92 provides the *outer join* operation, which can be used for this purpose. An outer join extends the result of an inner join and returns all rows that satisfy the join condition and also some or all of those rows from one or both tables for which no rows satisfy the join condition.

To write a query that performs an outer join of tables A and B and returns all rows from A (a left outer join), use the syntax *A left [outer] join B on condition*, For all rows in A that have no matching rows in B, Oracle returns null for any select list expressions containing columns of B.

To write a query that performs an outer join of tables A and B and returns all rows from B (a right outer join), use the syntax *A right [outer] join B on condition*. For all rows in A that have no matching rows in B, Oracle returns null for any select list expressions containing columns of A.

To write a query that performs an outer join and returns all rows from A and B, extended with nulls if they do not satisfy the join condition (a full outer join), use the

syntax *A full [outer] join B on condition*.

Example 5.10: Display all courses taught by CS faculty. If a course is not taught by a CS faculty, display it too. The query result is shown in Figure 5.12.

```
select Course.courseId, title,
  firstName || ' ' || lastName as "Name", deptId
from (Course left join TaughtBy
  on Course.courseId = TaughtBy.courseId) left join Faculty
  on Faculty.ssn = TaughtBy.ssn and deptId = 'CS';
```

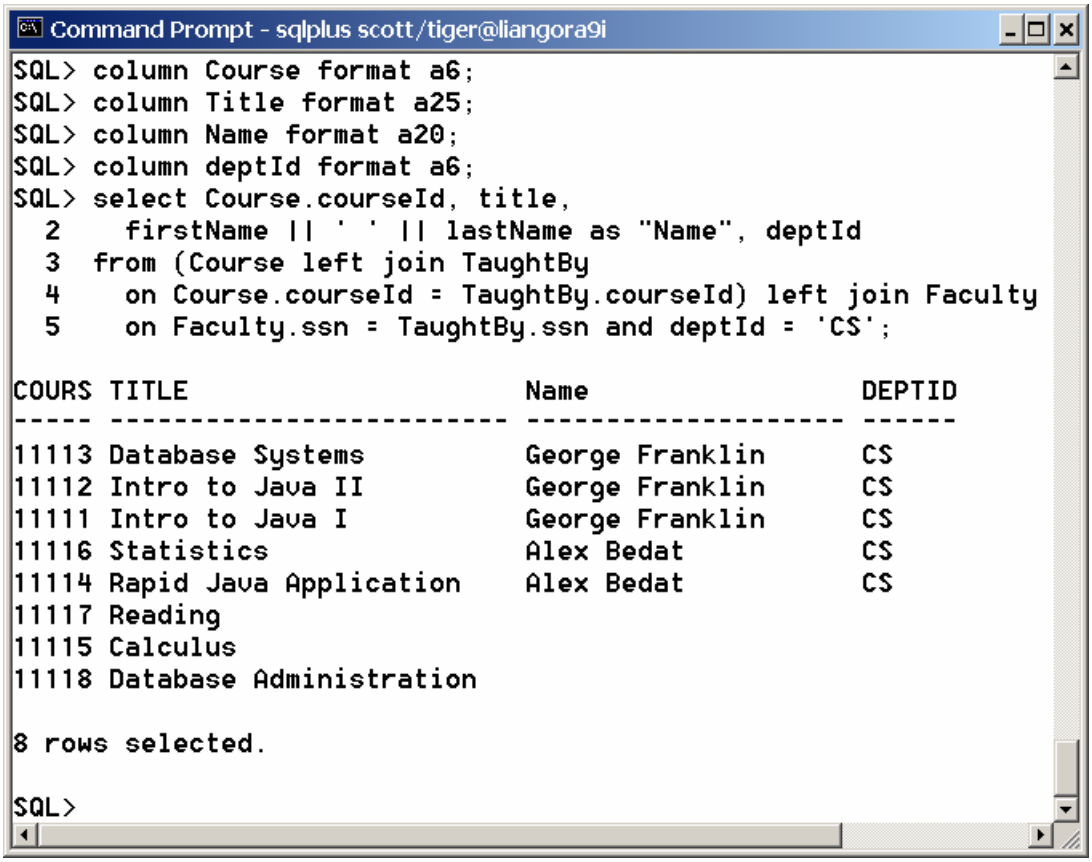

Figure 5.12

Example 5.10 demonstrates left outer join.

5.7 Set Operators

In Chapter 1, “Introduction to Database Systems,” you learned about the relational operators union, difference, and intersection. SQL supports these operators to combine the results of two queries into a single result.

- *Q1 union Q2* returns all distinct rows from both queries.

- *Q1 union all Q2* returns all rows from both queries (duplicates are not eliminated).
- *Q1 intersect Q2* returns all rows that appear in both queries.
- *Q1 minus Q2* returns all distinct rows selected by the first query but not the second.

Example 5.11: Display all faculty in the CS department and all faculty who teach the CS courses. A CS course may be taught by a faculty in other departments. You can use the union operator to solve the query. The query result is shown in Figure 5.13.

```
select firstName || ' ' || lastName as "Name"
from Faculty
where deptId = 'CS'
union
select firstName || ' ' || lastName as "Name"
from Faculty, TaughtBy, Course, Subject
where Faculty.ssn = TaughtBy.ssn and
  TaughtBy.courseId = Course.courseId and
  Course.subjectId = Subject.subjectId and
  Subject.deptId = 'CS';
```

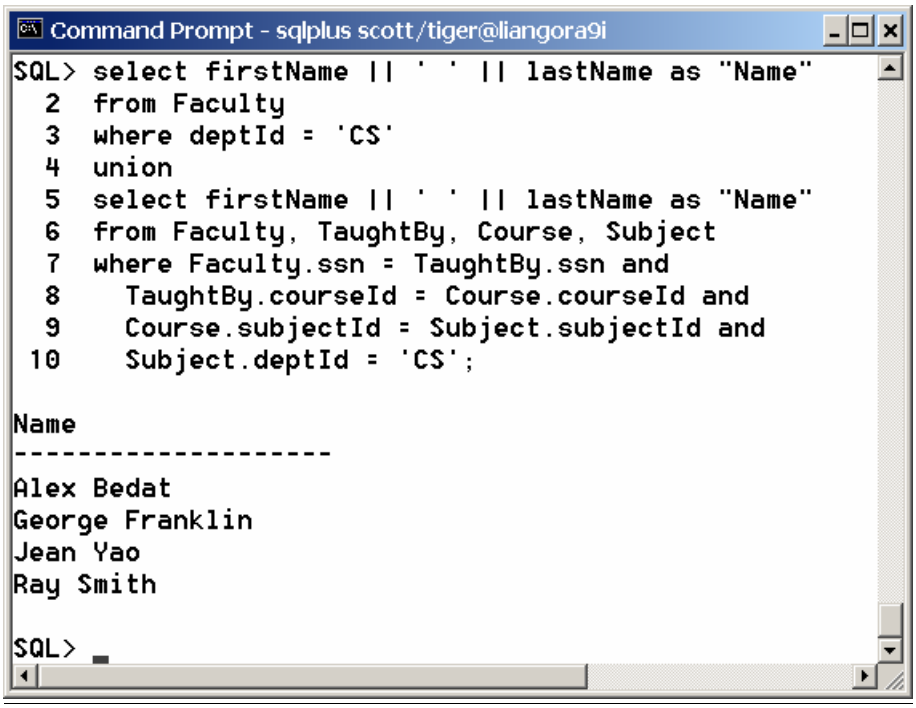

Figure 5.13

Example 5.11 demonstrates the union operator.

The query that uses the union operator can always be replaced by a query that uses the or operator. For example, the preceding query is equivalent to the following:

```
select distinct firstName || ' ' || lastName as "Name"
from Faculty, Subject, Course, TaughtBy
where Faculty.deptId = 'CS' or (Faculty.ssn = TaughtBy.ssn and
  TaughtBy.courseId = Course.courseId and
  Course.subjectId = Subject.subjectId and
  Subject.deptId = 'CS');
```

Example 5.12: Display all faculty in the CS department who currently teach a CS course. You can solve the query using the intersect operator. The result is the intersection of all CS faculty and all faculty who teach a CS course. The query result is shown in Figure 5.14.

```
select firstName || ' ' || lastName as "Name"
from Faculty
where deptId = 'CS'
intersect
select firstName || ' ' || lastName as "Name"
from Faculty, TaughtBy, Course, Subject
where Faculty.ssn = TaughtBy.ssn and
  TaughtBy.courseId = Course.courseId and
  Course.subjectId = Subject.subjectId and
  Subject.deptId = 'CS';
```

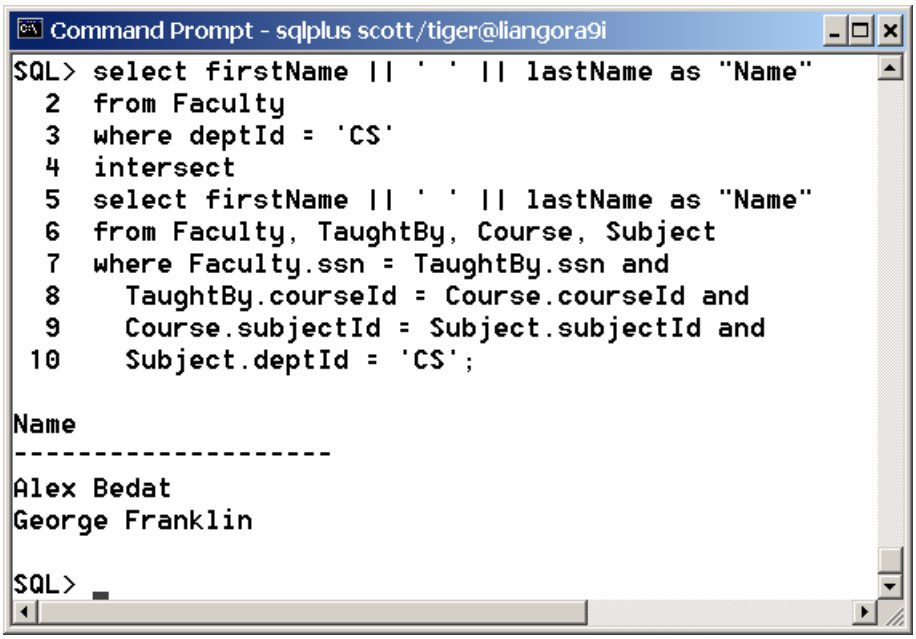

Figure 5.14

Example 5.12 demonstrates the intersect operator.

The query that uses the intersect operator can always be replaced by a query that uses the and operator. For example, the preceding query is equivalent to the following:

```
select distinct firstName || ' ' || lastName as "Name"
```

```
from Faculty, TaughtBy, Course, Subject
where Faculty.deptId = 'CS' and (Faculty.ssn = TaughtBy.ssn and
  TaughtBy.courseId = Course.courseId and
  Course.subjectId = Subject.subjectId and
  Subject.deptId = 'CS');
```

Example 5.13: Display all faculty not in the CS department who currently teach a CS course. You can solve the query using the minus operator. The result is all faculty who teach a CS course minus all CS faculty. The query result is shown in Figure 5.15.

```
select firstName || ' ' || lastName as "Name"
from Faculty, TaughtBy, Course, Subject
where Faculty.ssn = TaughtBy.ssn and
  TaughtBy.courseId = Course.courseId and
  Course.subjectId = Subject.subjectId and
  Subject.deptId = 'CS'
minus
select firstName || ' ' || lastName as "Name"
from Faculty
where deptId = 'CS';
```

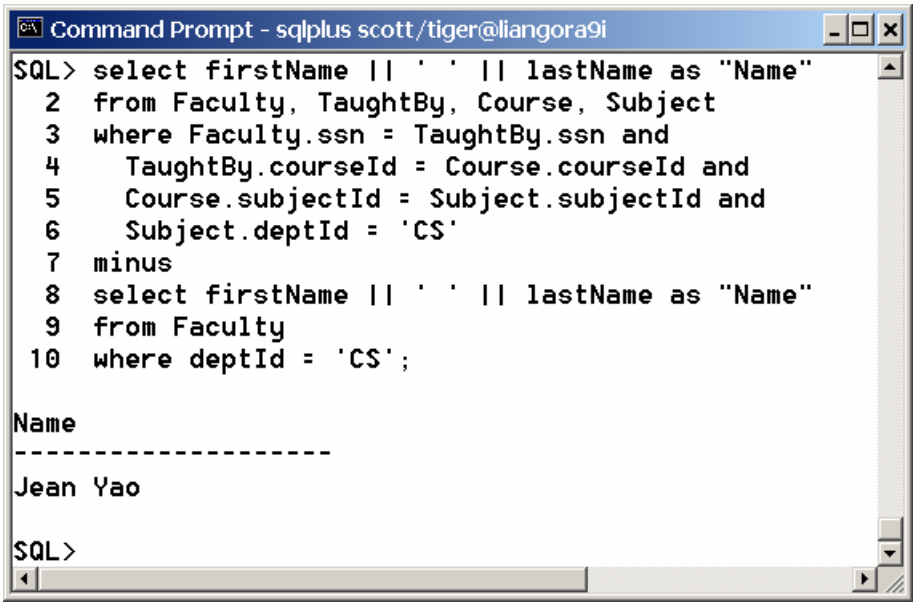

Figure 5.15

Example 5.13 demonstrates the minus operator.

The query that uses the minus operator can always be replaced by a query that uses the and not operator. For example, the preceding query is equivalent to the following:

```
select firstName || ' ' || lastName as "Name"
from Faculty, TaughtBy, Course, Subject
where Faculty.ssn = TaughtBy.ssn and
```

```
  TaughtBy.courseId = Course.courseId and
  Course.subjectId = Subject.subjectId and
  Subject.deptId = 'CS' and not (Faculty.deptId = 'CS');
```

5.8 Using Queries in the create table, insert, update, and delete Statements

The SQL select statement is the most used statement. You can use a nested select query to create a table, insert rows, update rows, and delete rows.

The general syntax for creating a table using a select query is:

```
create table TableName
  as select-statement;
```

For example, the following statement creates a table that contains the CS faculty.

```
create table TempFaculty
  as select ssn, firstName || ' ' || lastName as "Name",
       phone, rank, email, deptId, salary
     from Faculty where deptId = 'CS';
```

The general syntax for inserting into a table using a select query is:

```
insert into table TableName [column-list]
  select-statement;
```

To insert rows into a table, the table must already exist. For example, the following statement inserts Math faculty into TempFaculty.

```
insert into TempFaculty
  select ssn, firstName || ' ' || lastName,
    phone, rank, email, deptId, salary
  from Faculty where deptId = 'MATH';
```

The general syntax for updating rows in a table using a select query is:

```
update TableName
set columnName-list = (select-statement)
where condition;
```

For example, the following statement changes the salary of all full professors to the maximum faculty salary:

```
update TempFaculty
set salary = (select max(salary) from TempFaculty)
where rank = 'Full Professor';
```

The following statement changes the salary of the faculty with the lowest to the highest:

```
update TempFaculty
set salary = (select max(salary) from TempFaculty)
where salary = (select min(salary) from TempFaculty);
```

The general syntax for deleting rows from a table using a select query is:

```
delete [from] TableName
where columnName = select-statement;
```

For example, the following statement deletes all CS Faculty from TempFaculty who don't teach any courses.

```
delete from TempFaculty
where ssn in (
  select Faculty.ssn
  from Faculty, Enrollment
  where not exists (
    select courseId
    from TaughtBy
    where Faculty.ssn = TaughtBy.ssn));
```

5.9 Solving Queries Using Multiple Statements

Sometimes, it is difficult to get a solution for a query in one statement. You may break a program into several subproblems and store result for subproblems into temporary tables.

Example 5.14: Find the students who take *all* the courses taught by Professor George Franklin. This problem can be divided into the following subproblems:

1. Find all the courseId for the courses taught by George Franklin and save the result into a temporary table Temp1:

```
create table Temp1
  as select courseId
     from TaughtBy, Faculty
     where TaughtBy.ssn = Faculty.ssn and
       firstName = 'George' and lastName = 'Franklin';
```

2. Store all the students and the courseId into table Temp2 for the students who take courses in Temp1.

```
create table Temp2
  as select Student.ssn, Enrollment.courseId
     from Student, Enrollment, Temp1
     where Student.ssn = Enrollment.ssn and
       Enrollment.courseId = Temp1.courseId;
```

3. Display the students in Temp2 whose count is the same as the count in Temp1.

```
select Student.ssn, firstName, mi, lastName
from Student, Temp2
where Student.ssn = Temp2.ssn
group by Student.ssn, firstName, mi, lastName
having count(*) = (
  select count(*) from Temp1);
```

4. Drop all temporary tables.

```
drop table Temp2;
drop table Temp1;
```

5.10 Views

The concept of external views was introduced in Chapter 1, “Introduction to Database Systems.” A view is a virtual table. It presents only the part of the database that is interested to the user. In many ways, a view can be used like a table. This section introduces defining and using views in SQL.

A view is defined based on one or more tables or other views. The general syntax for creating a view is:

```
create [or replace] [force/noforce] view ViewName [column-list]
as select-statement
[with check option [constraint constraintName]]
[with read only];
```

- The or replace option replaces an existing view with the same name if it already exists.
- The force option creates the view even if the underlying table does not exist. In this case, the view is created, but cannot be used. The view can be used once the table is created. The default is noforce, which does not create the view if the underlying table does not exist.
- The column-list specifies the column names in the view. The number of columns in the list must match the number of columns selected from the select-statement.
- With the with check option clause specified, DBMS checks whether the condition specified in the where clause of the select-statement is violated for every insert or update statement on the view. If violated, the insertion or update operation is rejected. You may specify a constraint name for the check option.
- With the with read only clause specified, the view is not updatable.

The following statement creates a read-only view that lists the faculty and the courses taught by the faculty.

```
create or replace view FacultyCourse (name, course)
```

```
as select firstName || ' ' || lastName, title
   from Faculty, TaughtBy, Course
   where Faculty.ssn = TaughtBy.ssn and
     TaughtBy.courseId = Course.courseId
with read only;
```

The following statement creates a view that lists the CS faculty.

```
create or replace view CSFaculty
  (ssn, firstName, lastName, deptId)
as select ssn, firstName, lastName, deptId
   from Faculty
   where deptId = 'CS'
with check option;
```

The view has the with check option specified. If you attempt to insert a row using the following statement, DBMS rejects it because DBMS checks the condition and catches the violation (deptId is 'MATH'). Without the with check option specified, the row would be inserted.

```
insert into CSFaculty values ('111224444', 'Tim', 'Jones', 'MATH');
```

There are many restrictions on updating views:

- A view cannot be updated if it contains derived columns. The following view has a column that is a combination of firstName and lastName. This view cannot be updated.

```
create or replace view CSFaculty
  (ssn, name, deptId)
as select ssn, firstName || ' ' || lastName, deptId
   from Faculty
   where deptId = 'CS'
```

- No insertion if its base table contains a column with the not null constraint, which is not selected in the view.
- No insertion if the select-statement in the view definition has an order by clause.
- No insertion if a view is defined on multiple tables.
- Update and delete operations can be performed on a view that is defined on multiple tables, provided that the table has a primary key and other restrictions are not violated.

  NOTE: To drop a view, use the command drop view ViewName. You can also alter a view using the alter view command. Often it is more convenient to use create or replace to alter a view rather than altering it.

5.10.1 Solving Queries Using Views

Section 5.9, “Solving Queries Using Multiple Statements,” uses temporary tables to help solve complex queries. You can also use views instead of temporary tables. You can rewrite Example 5.14 using views as follows:

1. Create a view Temp1 for the courses taught by George Franklin:

```
create view Temp1
  as select courseId
     from TaughtBy, Faculty
     where TaughtBy.ssn = Faculty.ssn and
       firstName = 'George' and lastName = 'Franklin';
```

2. Create a view Temp2 for the students who take courses in Temp1.

```
create view Temp2
  as select Student.ssn, Enrollment.courseId
     from Student, Enrollment, Temp1
     where Student.ssn = Enrollment.ssn and
       Enrollment.courseId = Temp1.courseId;
```

3. Display the students in Temp2 whose count is the same as the count in Temp1.

```
select Student.ssn, firstName, mi, lastName
from Student, Temp2
where Student.ssn = Temp2.ssn
group by Student.ssn, firstName, mi, lastName
having count(*) = (
  select count(*) from Temp1);
```

4. Drop all temporary views.

```
drop view Temp2;
drop view Temp1;
```

A view is a virtual table. Its contents are dynamically generated upon execution. The contents of the view may be different in a transaction due to the change of the base table. Therefore, you should be careful to watch for the changes in the view after the base tables are changed.

Example 5.15: Exercise 4.3 defined two tables Card and Customer. Delete the customer who has the most cards and also delete the cards owned by that customer.

You may attempt to solve the problem as follows:

1. Create a view Temp1 that contains the ssn of the customer and the number of the cards owned by the customer.

```
create view Temp1 (ssn, numberOfCards)
  as select ssn, count(cardNo)
     from Customer
     group by ssn;
```

2. Create a view Temp2 that contains the cardNo owned by the customer who has the most cards.

```
create view Temp2 (cardNo)
  as select cardNo
     from Customer, Temp1
     where Customer.ssn = Temp1.ssn and
       Temp1.numberOfCards =
         (select max(numberOfCards) from Temp1);
```

3. Delete the cards from the Customer table.

```
delete Customer
where cardNo in (select cardNo from Temp2);
```

4. Delete the cards from the Card table.

```
delete Card
where cardNo in (select cardNo from Temp2);
```

5. Drop the views.

```
drop view Temp2;
drop view Temp1;
```

The solution is wrong, because when the cards are deleted from Customer, the contents for Temp2 are changed. Therefore, after Step 3, Temp2 no longer contains the cardNo owned by the customer who has the most cards. To fix the problem, you can simply create temporary tables instead of views.

5.11 Oracle rowId and rowNum columns

For each table, Oracle maintains a pseudocolumn rowId. For each query result, Oracle maintains a pseudocolumn rowNum. Each row in the database has an address. You can examine a row's address by querying the pseudocolumn rowId, but cannot modify it. Values of this pseudocolumn are hexadecimal strings representing the address of each row.

The rowId values have several important uses:

- They are the fastest way to access a single row.
- They can show you how a table's rows are stored.
- They are unique identifiers for rows in a table.

For each row returned by a query, the rowNum pseudocolumn returns a number indicating the order of the row in the selected result. The first row selected has a rowNum of 1, the second has 2, and so on. You can use rowNum to limit the number of rows returned by a query. This is sometimes referred to as a "top-N query" as shown in the following example.

Example 5.16: Select three faculty with highest salary. The

query result is shown in Figure 5.16.

```
create table Temp1 (name, salary)
as select firstName || ' ' || lastName "Name", salary
from Faculty
where salary is not null
order by salary desc;

select * from Temp1 where rowNum <= 3;

drop table Temp1;
```

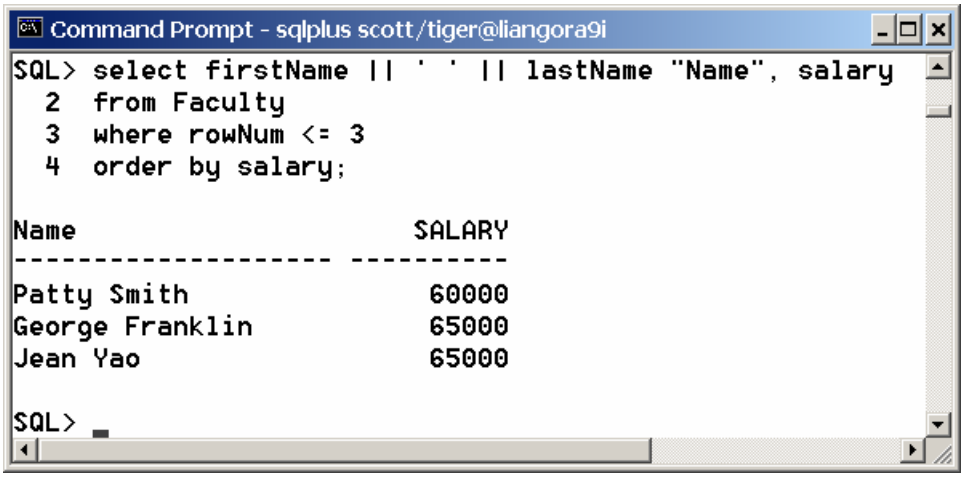

Figure 5.16

Example 5.16 demonstrates the rowNum column.

5.12 Indexes

Indexes are optional structures associated with tables. Indexes can be used to improve database performance. You can create indexes on one or more columns of a table to speed SQL statement execution on that. An index can be created using the create index statement. Once an index is created, the DBMS can utilize the index to search for a row quickly rather than scanning the entire table.

Indexes are logically and physically independent of the data in the associated table. You can create or drop an index at any time without affecting the base tables or other indexes. Your applications continue to work with or without indexes. Without indexes, your application may work slower. However, indexes require additional storage space.

> NOTE: By default, Oracle creates an index for the primary key and the unique constraints.

Oracle supports several types of indexes. Four frequently used indexes are:

1. *B-Tree indexes*. B-Tree is a variation of binary tree with multiple branches. It is commonly used in the database systems for creating indexes. By default, Oracle creates indexes using B-Tree.
2. *Bitmap indexes*. Bitmap indexes stores rowids associated with a key value as a bitmap.
3. *Unique indexes*. Unique indexes require that the indexes values are unique.
4. *Function-based indexes*. Function-based indexes create indexes based on expression or functions.

Here are some examples of creating indexes:

Example 5.17: Create a B-Tree index on the firstName and lastName columns of the Student table.

```
create index StudentIndex on Student (firstName, lastName);
```

To drop the index StudentIndex, use

```
drop index StudentIndex; (Oracle)

drop index StudentIndex on StudentIndex; (MySQL and Access)
```

Example 5.18: Create a Bitmap index on the firstName and lastName columns of the Student table.

```
create bitmap index StudentIndex on Student (firstName, lastName);
```

> NOTE: Bitmap indexes are only available in Oracle 9i Enterprise Edition.

Example 5.19: Create a unique index on the firstName and lastName columns of the Student table.

```
create unique index StudentIndex on Student (firstName, lastName);
```

In this case, the combination of firstName and lastName in the Student table must be unique. Otherwise, an index error would occur.

Example 5.20: Create a function-based index on the firstName and lastName columns of the Student table. Use the upper or lower function on firstName and lastName to enable case-insensitive search on firstName and lastName.

```
create index StudentIndex on Student (upper(firstName), upper(lastName));
```

> NOTE: You need the query rewrite privilege to be able to create function-bases indexes.

CAUTION: Indexes can be used to speed up queries. Indexes, however, add space and processing overhead to the database. Maintaining indexes may consume a significant CPU and I/O resources. In general, indexes can be used if your applications frequently retrieve small portion of the records in a large table.

TIP: By default, Oracle creates B-Tree indexes. B-Tree indexes provide excellent performance for quires, inserts, updates, and deletes, but it takes a lot of space to store B-Tree. Bitmap indexes can provide considerable storage savings over the use of B-Tree indexes. If most of the operations in your application involve queries, Bitmap indexes are generally appropriate.

5.12 Clusters

Tables are related through common columns. Often the common columns are used to join tables. To speed up the join operations of the tables on the common columns, you can create clusters to group these tables together. A cluster is a database object that physically stores the tables together through their common columns. Because the tables are stored together in the same data blocks, disk I/O is reduced for joins of clustered tables.

A cluster uses the cluster key values to group tables together. A cluster key value is the value of the cluster key columns for a particular row. Each cluster key value is stored only once in the cluster and the cluster index. For example, you may create a cluster to store Student and Enrollment together on the cluster key ssn. Figure 5.17 illustrates how these two tables are stored without using a cluster. Figure 5.18 illustrates how these two tables are stored using a cluster.

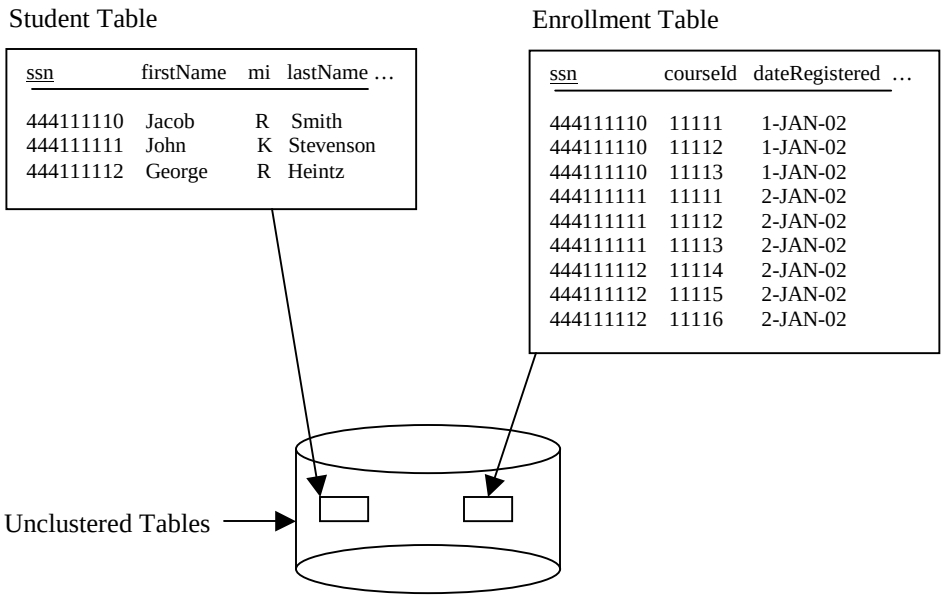

Figure 5.17

Tables are stored unclustered in the database.

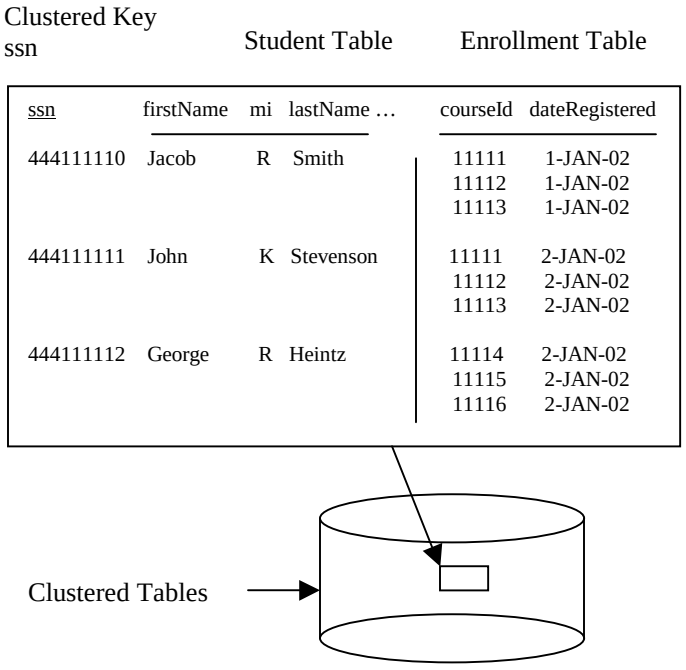

Figure 5.18

Tables are stored clustered in the database.

You can create a cluster using the create cluster statement. For example, the following statement creates a cluster on key ssn.

```
create cluster StudentEnrollment (ssn char(9));
```

Before you add tables into the cluster, the cluster key must be indexed using the create index statement. The following statement creates an index on the cluster StudentEnrollment.

```
create index idxStudentEnrollment on cluster StudentEnrollment;
```

Suppose the Student table and the Enrollment table are already created, to add them to the StudentEnrollment cluster, perform the following steps:

1. Create a temporary table for Student and a temp table for Enrollment.

```
create table TempStudent
  as select * from Student;

create table TempEnrollment
  as select * from Enrollment;
```

2. Drop tables Student and Enrollment.

```
drop table Student;

drop table Enrollment;
```

3. Create a new Student table and a new Enrollment table and add them to the cluster.

```
create table Student
  cluster StudentEnrollment (ssn)
  as select * from TempStudent;

create table Enrollment
  cluster StudentEnrollment (ssn)
  as select * from TempEnrollment;
```

4. Drop temporary tables TempStudent and TempEnrollment.

```
drop table TempStudent;

drop table TempEnrollment;
```

> TIP: Cluster tables that are accessed frequently if your application joins the tables on the common columns frequently. Don’t cluster the

tables if the tables are joined occasionally. It takes more time to insert, update, and delete records in the clustered tables.

5.13 Database Links and Distributed Queries

Database links are used to build distributed database systems. A distributed database system allows applications running on one database server to access data on other database servers over the network. A *database link* defines a schema object in the local database that enables the local user to access objects on a remote database. Once you have created a database link, you can use it to refer to tables and views on the remote database.

The syntax to create a database link is as follows:

```
create [public] database link dblink
connect to username identified by password using netservicename;
```

The public keyword specifies to create a public database link available to all users. If you omit this keyword, the database link is private and is available only to you. The dblink specifies the link name. The connect clause specifies the account with the username identified by the password on the remote server to be linked. The netservicename specifies the remote server. For the information on Oracle network service name, please see www.prenhall.com/liang/db.html.

Here is an example to create a database link named dblink_liang to scott/tiger on a remote server whose network service name is liang.

```
create public database link dblink_liang
connect to scott identified by tiger using 'liang';
```

You can now access the objects on the remote server through dblink_liang. The following displays the contents in the Address table on the remote server.

```
select * from Address@dblink_liang;
```

To drop a database link, use

```
drop [public] database link dblink;
```

> Note: A database link is created from one database to the other. You cannot have a link on the same databases.

5.14 Synonyms

Sometimes you want to give all database users the access to a table with a simple unified name. You can create a synonym using the create synonym statement. Synonyms provide both data independence and location transparency. Synonyms permit applications to function without modification regardless of which user owns the table or view and regardless of which database holds the table or view.

For example, the DBA can create a public synonym for the Student table owned by scott as follows:

```
create public synonym Student for scott.Student;
```

Now all users can access the table using the synonym Student.

The DBA can also create a public synonym for the Address table through the database link as follows:

```
create public synonym Address for scott.Student;
```

To drop a synonym, use

```
drop [public] synonym SynonymName;
```

5.15 Sequences

A sequence is an Oracle object that can be used to generate sequence of integers. These integers are often used as values for surrogate integer key attribute such as userId and deptId, and productId. The syntax for creating a sequence is

```
create sequence sequenceName
  [increment by n]
  [start with s]
  [maxvalue max | nomaxvalue]
  [minvalue minr | nominvalue]
  [cycle | nocyle]
```

- **increment by n** increments the next generated value by n. If n is negative, the next generated value decreases. By default, the increment is 1.
- **start with s** specifies that the sequence starts with integer s. By default, the start value is 1.
- **maxvalue max** specifies that the maximum value generated. **nomaxvalue** indicates that the sequence continues to generate until it reaches 10^{27}. If the

increment is negative, the nomaxvalue is –1. **nomaxvalue** is the default.

- **minvalue min** specifies that the minimum value generated. If **nominvalue** is specified, the minimum value is 1 for ascending sequence, and -10^{26} for descending sequence.

- **cycle** indicates that the sequence continues to generate from the start after reaching the maximum or minimum value. **nocycle** indicates that no more generation after reaching the maximum or minimum value. The default is nocycle.

The following statement creates a sequence starting with 100, with an increment 10, maximum value 9000, and no cycle.

```
create sequence MySequence
  increment by 10
  start with 100
  maxvalue 9000
  nocycle;
```

The sequences are accessed by two functions nextval and curval. nextval returns the next generated value and curval returns the current generated value. The nextval function must be used before the first invocation of the curval function. For example, the following statement inserts a new course with a generated course ID.

```
insert into Course values (
  MySequence.nextval, 'CSCI', MySequence.nextval,
  'Advanced DB', 3);
```

You can use the following statement to view the current generated value:

```
select MySequence.currval from Dual;
```

> NOTE: The nextval generates the next value in the sequence. Once it is generated, it cannot be rollback even if the statement that invokes the function is rollbacked.

Chapter Summary

This chapter introduced advanced SQL. You learned to write queries using nested SQL, using various types of joins, and using the operators in, all, any, exists, and unique. You learned to use the select statements with other SQL

statements such as the create table, insert, update, and delete statements. You learned to create views, indexes, and clusters. You also learned to create database links, synonyms, and sequences.

Review Questions

5.1 Can you join two same tables? What is a table alias?

5.2 What is a natural join, inner join, left outer join, right outer join, and full outer join?

5.3 Can the equality (=) operator be used to compare an element with a set?

5.4 Describe the in, all, any, exists, and unique operators.

5.5 Can you create a view using the syntax create or replace view? Can you create a table using the syntax create or replace table?

5.6 In the create view statement, what is the force option? what is the with check option? What is the with read only option?

5.7 Can a view be updated in the following cases:

- A view contains derived columns.
- A view is defined as read only.

5.8 Can you insert into a view in the following cases:

- Its base table contain a column with the not null constraint, which is not selected in the view.
- The select-statement in the view definition has an order by clause.
- A view is defined on multiple tables.

5.9 What are the rowId and rowNum columns? Can you modify the values on these columns?

5.10 If you use the following query to find the top three faculty with the highest salary, what is wrong?

```
select firstName || ' ' || lastName "Name", salary
from Faculty
where salary is not null and rowNum <= 3
order by salary desc;
```

5.11 What type of indexes can you create in Oracle?

5.12 What is a cluster? How do you create a cluster in

Oracle?

5.13 What is a synonym? How do you create a synonym in Oracle?

5.14 What is a database link? How do you create a database link in Oracle?

5.15 What is a sequence object for? How do you create a sequence? How do you get the next generated sequence value and how do you obtain the current sequence value?

Exercises

5.1 Find the department with the most faculty.

5.2 List the total number of faculty in each college in order.

5.3 Find the college with the most faculty.

5.4 Find the student who takes the most courses.

5.5 List the course taken by student Jacob Smith and the instructor who teach the course.

5.6 Find the pairs of faculty who share the same office.

5.7 Find all Computer Science students who take Math courses.

5.8 Find all Computer Science students who *only* take Computer Science courses.

5.9 Find all Computer Science students who take at least one Computer Science course and at least one Math course.

5.10 Find all faculty with the same last name.

5.11 Find all Computer Science students who take at least two Computer Science courses and at least one Math course.

5.12 List the Computer Science students whose last name begins with S and who take a course from Professor Alex Bedat in alphabetical order of their last names.

5.13 List the number of students in each college.

5.14 List the total credit hours in each department.

5.15 Find all faculty who teach courses in both Computer Science and Math departments.

5.16 Create a table that consists of students in the computer science department who receive grade ‘A’ or ‘B’ on all courses. The table contains the student name, course and the grade.

5.17 Delete all students who are not taking any courses.

5.18 Increase salary of Math professor by 5%.

5.19 Create a view named CSStudent that consists of all Computer Science students.

5.20 List three departments with most of students.

5.21 Rewrite Example 5.8 using the in operator.

5.22 Find all faculty who currently teach at least one course.

5.23 Find all faculty who teach exactly one course.

5.24 Find top two departments with the most faculty.

5.25 Find the department name for the department with the largest total salary.

******Au NOTE:***

??function-based index, bitmap, (hash??) internal use

Clusters (Find the CD-ROM)

Internationalization?

More examples and more exercises