Chapter 5 Relational Database Standard

5.1 Structured Query Language (SQL)

An Introduction to SQL

In relational database systems data are represented using tables (relations). A query issued against the database system also results in a database.

A table is uniquely identified by its name and consists of rows that contain the stored information, each row containing exactly one tuple (or record). A table can have one or more columns.
A column is made up of a column name and a data type, and it describes an attribute of the tuples. The structure of a table, also called relation schema, thus is defined by its attributes.
The type of information to be stored in a table is defined by the data types of the attributes at table creation time.
SQL uses the terms table, row, and column for relation, tuple, and attribute, respectively.

A table can have up to 254 columns which may have different or same data types and sets of values (domains), respectively. Possible domains are alphanumeric data (strings), numbers and date formats.
Oracle offers the following basic data types:

- **char(n):** Fixed-length character data (string), n characters long. The maximum size for n is 255 bytes (2000 in Oracle8). Note that a string of type char is always padded on right with blanks to full length of n. (can be memory consuming).
  Example: **char**(40)
- **varchar2**(n): Variable-length character string. The maximum size for n is 2000 (4000 in Oracle8). Only the bytes used for a string require storage. Example: **varchar2**(80)
- **number**(o, d): Numeric data type for integers and reals.
  o= overall number of digits, d = number of digits to the right of the decimal point. Maximum values: o=38, d= -84 to +127.
  Examples: **number**(8), **number**(5,2)
  Note that, e.g., **number**(5,2) cannot contain anything larger than 999.99 without resulting in an error. Data types derived from **number** are **int**[**eger**], **dec**[**imal**], **smallint** and **real**.
- **date**: Date data type for storing date and time. The default format for a date is: DD-MMM-YY.
  Examples: '13-OCT-94', '07-JAN-98'
- **long**: Character data up to a length of 2GB. Only one **long** column is allowed per table.

Note: In Oracle-SQL there is no data type **boolean**. It can, however, be simulated by using either **char**(1) or **number**(1).
As long as no constraint restricts the possible values of an attribute, it may have the special value null (for unknown). This value is different from the number 0, and it is also different from the empty string ''.

Example Database

We use an example database to manage information about employees, departments and salary scales.
The table EMP is used to store information about employees:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
……	……	……	……	……	……	……	……..
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20

For the attributes, the following data types are defined:
EMPNO: number(4), ENAME:varchar2(30), JOB: char(10), MGR: Number(4),
HIREDATE: date, SAL: number(7,2), DEPTNO: number(2)

Each row (tuple) from the table is interpreted as follows: an employee has a number, a name, a job title and a salary. Furthermore, for each employee the number of his/her manager, the date he/she was hired, and the number of the department where he/she is working are stored.
The table DEPT stores information about departments (number, name, and location):

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

Finally, the table SALGRADE contains all information about the salary scales, more precisely, the maximum and minimum salary of each scale.

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

SQL Queries

In SQL a query has the following form (components in brackets [ ] are optional):

```
SELECT [ALL | DISTINCT] <list of columns>
FROM < list of tables or views>
[WHERE < condition(s)>]
[GROUP BY <column(s)>]
[HAVING < condition>]
ORDER BY < column(s) [ASC|DESC] >];
```

where ASC|DESC allows the ordering to be done in ascending or descending order.

Selecting Columns

The SELECT statement can be used to retrieve specific column(s) from the table by specifying the column names from the table.

Syntax:

```
SELECT column_name1 [, column_name2]
FROM table _name;
```

Example:

```
SELECT ename, sal
FROM emp;
```

The above table retrieves the ename and sal from emp table.

Selecting All Columns

The SELECT statement along with the asterisk (*) symbol produces the result in the form of detailed data.

Example:

```
SELECT *
FROM emp;
```

The above statement lists all the rows along with all the columns specified at the time of the emp table.

Changing Column Sequence

The order of columns can be changed in the result set of the SELECT statement.

Example:

```
SELECT ename, empno
FROM emp;
```

Manipulating Column Names

SQL hides the column heading when any kind of expression or manipulation on multiple columns take place. A user-defined column heading can replace the default column heading.

Syntax:

```
SELECT column-name column-alias [, column-name column-alias]
FROM table-name
```

where column-alias is the user-defined column heading that is to be specified in place of the default heading.

Example:
```
SELECT ename "Name", sal "Salary"
FROM emp;
```

Using Literals

The result set of the data query statement can be made more readable by including a string called a literal in the SELECT list. A string literal should typically be used before the column name for which a string is to be displayed.

Syntax:

SELECT *column-name* |'*String-literal*' [, *column-name* | '*String-literal* ']
FROM *table-name*

where *string-literal* is the string enclosed in quotes to be displayed before the column data.

Example:

```
SELECT empno, 'Employee Name is:', ename
FROM emp;
```

Arithmetic Operators

SQL supports operators that perform arithmetic operations like addition, subtraction, division and multiplication on numeric columns.

The arithmetic operators supported by SQL are:

+ for Addition, - for subtraction, / for division, * for multiplication, % for modulo.

Example: Write a SQL statement to display the name and annual salary of each employee. Display the annual salary column as "Annual Salary".

```
SELECT ename "NAME", sal*12 "Annual Salary"
FROM emp;
```

Selecting Rows

There are situations in which only a few rows need to be retrieved from the table based on the condition. We use WHERE clause for this.

Syntax:

```
SELECT column-list
FROM table-name
WHERE search-condition;
```

where *search-condition* is the condition(s) on which data is to be retrieved.

Example: Write a SQL statement to display the name and salary of employees who are working at department 10.

```
SELECT ename Name, sal Salary
FROM emp
WHERE deptno = 10;
```

Search based on conditions

SQL provides a few methods for searching the row in a table.

Comparison Operator

Comparison operator allows row retrieval from a table based on the condition specified in WHERE clause.

Syntax:

```
SELECT column-list
FROM table-name
WHERE expression1 comparison-operator expression2;
```

where *expression1* and *expression2* is any valid combination of constant, variable, function or column-based expression.

comparison-operator is any valid operator from the operator list <, <=, =, !=, >, >=

( ) controls precedence.

Examples: 1. Write a SQL statement to display the name and salary of employee who earn less than Rs. 3000.

```
SELECT ename "Name", sal "Salary"
FROM emp
WHERE sal < 3000;
```

2. Write SQL statement to display name and salary of employees who are managers.

```
SELECT ename Name, sal salary
FROM emp
WHERE job = 'MANAGER';
```

Range Operator

The *range* operator is used to retrieve data that can be extracted in ranges. The *range* operators are BETWEEN and NOT BETWEEN.
BETWEEN keyword specifies an inclusive range to search.
NOT BETWEEN keyword is used to exclude rows from specified range in the result set.

Syntax:

SELECT *column-list*
FROM *table-name*
WHERE *expression1 range-operator expression2 AND expression3*;

where
Expression1 and *expression2* is any valid combination of constant, variable, function or column-based expression.
Range-operator is any valid range operator.

Example: Write a SQL statement to display the name and salary of employees who earn between 3000 and 5000.

```
SELECT ename Name, sal Salary
FROM emp
WHERE sal BETWEEN 3000 AND 5000;
```

List Operator

SQL provides the IN operator to allow the selection of values that match any one of the values in the list. NOT IN restricts the selection of values that match any one of the values in the list.

Syntax:

SELECT *column-list*
FROM *table-name*
WHERE *expression list-operator (value-list)*;

where
expression is any valid constant, variable, function or column-based expression.
list-operator is any valid list operator.
value-list is the list of values to be included or excluded in the condition.

Examples:

1. Write SQL statement to display the name and salary of employee who earn more than 1500 and are in the department 10 or 30.

```
SELECT ename Name, sal Salary, deptno "Dept No."
FROM emp
WHERE sal > 1500 AND deptno IN(10, 30);
```

2. Write SQL statement to display the name and salary of employee who earn more than 1500 and are not in the department 10 or 30.

```
SELECT ename Name, sal Salary, deptno "Dept No."
FROM emp
WHERE sal > 1500 AND deptno NOT IN(10, 30);
```

String Operator

SQL provides a pattern-matching method for string expressions using the LIKE keyword with the wildcard mechanism.

Wildcard	Description
%	Represents any string of zero or more character(s)
_	(underscore) Represents a single character
[ ]	Represents any single character within specified range
[^]	Represents any single character not within specified range

Examples:

1. Write SQL statement to display the name and salary of employees where the third letter of their name is A.

```
SELECT ename Name, sal Salary
FROM emp
WHERE ename LIKE '__A%';
```

2. Write SQL statement to display the name of employee who have two ‘L’s in their name and their manager is 7782.

```
SELECT ename Name
FROM emp
WHERE ename LIKE '%L%L%' AND mgr = 7782;
```

Unknown Values

In SQL terms, NULL is an unknown value or a value for which data is not available.

Syntax:

```
SELECT column-list
FROM table-name
WHERE column-name unknown-value-operator;
```

where

unknown-value-operator is either the keywords IS NULL or IS NOT NULL.

Example: Write SQL statement to display the name of employee who earn commission.

```
SELECT ename Name, comm Commission
FROM emp
WHERE comm IS NOT NULL;
```

Logical Operator

Multiple search conditions can be combined by using logical operators provided by ORACLE-SQL.
OR – Returns the result when any of the specified search conditions is true.
AND – Returns the result when all of the specified search conditions are true.
NOT – Neutralize the expression that follows it.

Syntax:

```
SELECT column-list
FROM table-name
WHERE conditional-expression {AND/OR}[NOT ] conditional-expression
```

where

conditional-expression is any conditional expression that is combined with a logical operator..

Example:

1. Write SQL statement to display the name of employee and department number for all employees in the department number for all employees in the department 10 or 30.

```
SELECT ename Name, deptno DeptNumber
FROM emp
WHERE deptno=10 OR deptno = 30;
```

2. Write SQL statement to display the name salary of employees who earn more than 1500 and are in the department 10.

```
SELECT ename Name, sal Salary, deptno DeptNumber
FROM emp
WHERE deptno=10 AND sal > 1500;
```

Limiting Result Sets

The DISTINCT keyword

The DISTINCT keyword removes duplicate rows from the result set. By default, a query returns all the records including duplicates.

Syntax:

```
SELECT [ ALL | DISTINCT ] column-list
FROM table-name
WHERE search-condition;
```

where

search-condition is any condition on which a row is to be retrieved.

Example:

Write SQL statement to display distinct jobs that ends with ‘T’ from emp table.

```
SELECT DISTINCT job
FROM emp
WHERE job LIKE '%T‘;
```

Aggregate Functions

AVG([ALL |DISTINCT] *expression*)	Return the average of the values specified in the expression, either for all records or distinct records.
SUM([ALL |DISTINCT] *expression*)	Return sum of the values specified in the expression, either for all records or distinct records.
MIN(*expression*)	Return the minimum of the values specified in the expression.
MAX(*expression*)	Return the maximum of the values specified in the expression.
COUNT([ALL |DISTINCT] *expression*)	Returns the unique number of records or all records specified in the expression.
COUNT(*)	Returns the total number of records specified in the expression.

The AVG, SUM, MIN, MAX, and COUNT functions ignore NULL values, whereas the COUNT (*) function counts the NULL values.

Examples:

1. Write SQL statement to display the total number of employees in the company.
```
SELECT COUNT(empno) "Total Employee“
FROM emp;
```
2. Write SQL statement to display the average salary of employee.
```
SELECT ROUND(AVG(sal)) "Average sal emp“
FROM emp;
```
3. Write SQL statement to display the total number of analyst in the company.
```
SELECT COUNT(empno) "Employee in Analyst Dept“
FROM emp
WHERE job = 'ANALYST';
```
4. Write SQL statement to display the highest, lowest, sum and average salary of all employees. Label the columns MAX_SAL, MIN_SAL, SUM_SAL, and AVG_SAL respectively.
```
SELECT MAX(sal) Max_Sal, MIN(sal) Min_Sal, SUM(sal) Sum_Sal,
ROUND(AVG(sal)) Avg_Sal
FROM emp;
```

Grouping Result Sets

The GROUP BY clause summarises the result set into the groups defined in the query using aggregate functions. The HAVING clause further restricts result set, produce the data based on a condition.

Syntax:

SELECT *column-list*
FROM *table-name*
WHERE *condition*
GROUP BY [ALL] *expression* [, *expression*]
[HAVING *search-condition*]*;*

where

ALL – is the keyword used to include the groups that do not meet the search condition.
expression – specifies the column name(s) or expression(s) on which the result set of SELECT statement is to be grouped.
search-condition – is the conditional expression on which the result is to be produced.

Example: Find department number and number of employees working in that department.

```
SELECT deptno, count (empno)
FROM emp
GROUP BY deptno;
```

2. Find department number and maximum salary of those department where maximum salary is more than Rs. 20000.
```
SELECT DEPTNO, MAX(SAL)
FROM EMP
GROUP BY DEPTNO
HAVING MAX(SAL) > 20000;
```

Joins

SQL provides a method of retrieving data from more than one table using joins. A join can be defined as an operation that includes retrieval of data from more than one table at a time.

Syntax:

SELECT *column-name*, *column-name* [, *column-name*]
FROM *table-name* JOIN *table-name*
ON *table-name.ref-column-name join-operator table-name.ref-column-name*

where

column-name – specifies the name of the column(s) from one or more than one table that has to be displayed.
table-name - specifies the name of the table(s) from which data is to be retrieved.
ref-column-name - specifies the name of the column(s) that is used to combine two tables using the common keys from the respective tables.
join-operator – specifies the operator used to join the table(s).

When two tables are joined, they must share a common key that defines how the rows in the tables correspond to each other.

When using joins to retrieve data from multiple tables, a non-selected column can be referred to elsewhere in the SELECT statement. Generally, a primary key is validated against the foreign key when a join is used. Whenever a column is referred to in a join condition, it should be referred to either by prefixing it with the table name to which it belongs or by a table alias.

A table name or a table alias is required whenever an ambiguity is possible due to duplicate column names in multiple tables. A table alias is a keyword defined in the FROM clause of the SELECT statement to uniquely identify the table.

The command syntax for a table alias is:

FROM *table-name table-alias*

table-name – specifies the name of the table(s) that has to be combined in the query.
table-alias – is the keyword used to refer to a table. It must follow the rules of identifiers.

Cross-Join

A join that includes more than one table without any condition in the ON clause is called a *cross join*. The output of such join is called a Cartesian product.

Example: If a join is performed on a table named “emp” that has 14 rows and another named “dept” with 4 rows, the result will be 56 rows, the Cartesian product of the two tables.

```
SELECT empno, ename, deptno
FROM emp CROSS JOIN dept;
```

Natural Join

A join that restricts the redundant column data from the result set is known as a natural join. It is implemented by specifying the various column names of the tables in the SELECT list.

Example:

```
SELECT empno, ename, dname, depno
FROM emp NATURAL JOIN dept;
```

Equi-Join

A join that uses an asterisk (*) sign in the SELECT list and displays redundant column data in the result set is termed as an equi-join. An equi-join displays redundant column data in the result set, where two or more tables are compared for equality.

Examples:

1. Example: In the table EMP only the numbers of the departments are stored, not their name. For each salesman, we now want to retrieve the name as well as the number and the name of the department where he is working:

```
SELECT ename, e.deptno, dname
FROM emp e, dept d
WHERE e.deptno = d.deptno
AND job = ’salesman’;
```

2. Any number of tables can be combined in a select statement.
   For each project, retrieve its name, the name of its manager, and the name of the department where the manager is working:

```
SELECT ename, dname, pname
FROM emp e, dept d, project p
WHERE e.empno = p.mgr AND  d.deptno=e. deptno;
```

3. Write a SQL statement to display the total salary of each department and also the department name.

```
SELECT SUM(e.sal) "Total salary", d.dname DEPARTMENT
FROM emp e, dept d WHERE e.deptno=d.deptno GROUP BY dname;
```

4. Write a SQL statement to display the employee name, department name and location of all employees who earn commission.

```
SELECT ename NAME, dname DEPT_NAME, loc LOCATION FROM emp, dept
WHERE emp.deptno=dept.deptno AND comm IS NOT NULL;
```

Self-Join

A join is said to be a self join when one row in a table correlates with other rows in the same table. Since the same table is used twice for comparison, an alias name differentiates the two copies of the table.

1. List the names of all employees together with the name of their manager.

```
SELECT e1.ename, e2.ename
FROM emp e1, emp e2
WHERE e1.mgr = e2.empno;
```

2. Write a SQL statement to display employee name and employee number along with their manager's name and manager number. Label the columns as EMP#, NAME, MGR#, MANAGER respectively.

```
SELECT e.empno EMP#, e.ename NAME, e.mgr MGR#, m.ename MANAGER
FROM emp e, emp m
WHERE e.mgr=m.empno
ORDER BY e.mgr;
```

Subqueries

A subquery can be defined as a group of nested SELECT statements inside a SELECT, INSERT, UPDATE, or DELETE statement. It can be used to retrieve data from multiple tables and as an alternative to a join. It can be used inside the WHERE or HAVING clause of the outer SELECT, INSERT, UPDATE, and DELETE statements. The SELECT statements that contain one or more subqueries are called nested queries.

Syntax:

```
(SELECT [ALL | DISTINCT] subquery-select-list
[FROM {table-name | view-name}
      [[, {table-name2 | view-name2}] […,{table-name16 | view-name16}]]
[WHERE clause]
[GROUP BY clause]
[HAVING clause])
```

A subquery must be enclosed within parentheses and can not use the ORDER BY, COMPUTE BY or FOR BROWSE clause.

Subqueries can be divided into three categories depending upon the values the return:

- Subqueries that operate on lists: This type of query returns single column-multiple value results and are implemented using the IN clause. The syntax is as follows:

  WHERE *expression* [NOT] IN (*subquery*)

- Subqueries that are introduced with an unmodified comparison operator: This type of query returns single column-single value results for outer query evaluation and is implemented using unmodified comparison operators(operators without the ANY or ALL keyword). The syntax is as follows:

  WHERE *expression comparison_operator*[ANY|ALL] (*subquery*)

- Subqueries that check for the existence of data: This type of query checks for the existence of records in a table that is used in the inner query, and returns either a TRUE or FALSE value based on the existence of data. It is implemented using the EXISTS keyword. The syntax is as follows:

  WHERE [NOT] EXISTS (*subquery*)

Subqueries with IN

A subquery introduced with IN returns zero or more values.

Examples:

1. List the name and salary of employees of the department 20 who are leading a project that started before December 31, 1990:

```
SELECT ename, sal FROM emp
WHERE empno IN
        (SELECT pmgr FROM project
        WHERE pstart < '31-dec-90') AND deptno =20;
```

2. List all employees who are working in a department located in BOSTON:

```
SELECT * FROM emp
WHERE deptno IN
        (SELECT deptno FROM dept  WHERE loc = 'BOSTON');
```

3. List all those employees who are working in the same department as their manager (note that components in [ ] are optional:

```
SELECT * FROM emp e1
WHERE deptno IN
        (SELECT deptno FROM emp [e]  WHERE [e.]empno = e1.mgr);
```

4. Write a SQL statement to display the name of employee who is drawing highest salary.

```
SELECT ename NAME, sal SALARY FROM emp
WHERE sal IN
        (SELECT MAX(sal) FROM emp)
```

Subqueries with EXISTS

List all departments that have no employees:

```
SELECT * FROM dept
WHERE NOT EXISTS
(SELECT * FROM emp WHERE deptno = dept.deptno);
```

Subqueries with ANY

1. Retrieve all employees who are working in department 10 and who earn at least as much as any (i.e., at least one) employee working in department 30:

```
SELECT * FROM emp
WHERE sal >= ANY
        (SELECT sal FROM emp
        WHERE deptno = 30)
        AND deptno = 10;
```

2. Write a SQL statement to display the employee number and name for all employees who earn more than the average salary. Sort the results in descending order.

```
SELECT ename NAME, job DESIGN, sal SALARY
FROM emp
WHERE sal > ANY
        (SELECT MAX(AVG(sal)) FROM emp GROUP BY deptno)
        ORDER BY sal DESC;
```

Operations on Result Sets

Sometimes it is useful to combine query results from two or more queries into a single result.
SQL supports three set operators which have the pattern:
<query 1> <set operator> <query 2>

The set operators are:

- **union** [**all**] returns a table consisting of all rows either appearing in the result of <query 1> or in the result of <query 2>. Duplicates are automatically eliminated unless the clause **all** is used.
- **intersect** returns all rows that appear in both results <query 1> and <query 2>.
- **minus** returns those rows that appear in the result of <query 1> but not in the result of <query 2>.

Examples:
Assume that we have a table EMP2 that has the same structure and columns as the table EMP:

- All employee numbers and names from both tables:

```
(SELECT empno, ename FROM emp)
UNION
(SELECT empno, ename FROM emp2);
```

- Employees who are listed in both EMP and EMP2:

```
(SELECT * FROM emp)
INTERSECT
(SELECT * FROM emp2);
```

- Employees who are only listed in EMP:
  (SELECT *FROM emp)
  MINUS
  (SELECT * FROM emp2);

Each operator requires that both tables have the same data types for the columns to which the operator is applied.

DDL - Data Definition Language
Statements used to define the database structure or schema.
Some examples:
CREATE - to create objects in the database
ALTER - alters the structure of the database
DROP - delete objects from the database
TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
COMMENT - add comments to the data dictionary
RENAME - rename an object
DCL - Data Control Language.
Some examples:
GRANT - gives user's access privileges to database
REVOKE - withdraw access privileges given with the GRANT command
DML - Data Manipulation Language
Statements used for managing data within schema objects.
Some examples:
SELECT - retrieve data from the a database
INSERT - insert data into a table
UPDATE - updates existing data within a table
DELETE - deletes all records from a table, the space for the records remain
MERGE - UPSERT operation (insert or update)
CALL - call a PL/SQL or Java subprogram
EXPLAIN PLAN - explain access path to data
LOCK TABLE - control concurrency
TCL - Transaction Control
Statements used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.
COMMIT - save work done
SAVEPOINT - identify a point in a transaction to which you can later roll back
ROLLBACK - restore database to original since the last COMMIT
SET TRANSACTION - Change transaction options like isolation level and what rollback segment to use
DML are not auto-commit. i.e. you can roll-back the operations, but DDL are auto-commit

Creating Tables

The SQL command for creating an empty table has the following form:

```
CREATE TABLE <table> (
<column 1> <data type> [NOT NULL] [UNIQUE] [<column constraint>],
. . . . . .
<column n> <data type> [NOT NULL] [UNIQUE] [<column constraint>],
[<table constraint(s)>]
);
```

For each column, a name and a data type must be specified and the column name must be unique within the table definition. Column definitions are separated by comma. There is no difference between names in lower case letters and names in upper case letters. In fact, the only place where upper and lower case letters matter are strings comparisons. A not null constraint is directly specified after the data type of the column and the constraint requires defined attribute values for that column, different from null.

The keyword **unique** specifies that no two tuples can have the same attribute value for this column. Unless the condition **not null** is also specified for this column, the attribute value null is allowed and two tuples having the attribute value null for this column do not violate the constraint.

Example: The create table statement for our EMP table has the form

```
CREATE TABLE emp1 (
empno NUMBER(4) NOT NULL,
ename VARCHAR2(30) NOT NULL,
job VARCHAR2(10),
mgr NUMBER(4),
hiredate DATE,
sal NUMBER(7,2),
deptno NUMBER(2)
);
```

Constraints

The definition of a table may include the specification of integrity constraints. Basically two types of constraints are provided: column constraints are associated with a single column whereas table constraints are typically associated with more than one column. However, any column constraint can also be formulated as a table constraint.

The specification of a constraint has the following form:

column_name CONSTRAINT *constraint_name constraint_type*
[, *column_name* CONSTRAINT *constraint_name constraint_type*]

- *column_name* – name of column on which the constraint is to be defined.
- *constraint_name* – name of constraint to be created and must follow rules of the identifier.
- *constraint_type* – type of constraint to be added.

PRIMARY KEY Constraint

A primary key constraint enables a unique identification of each tuple in a table. Based on a primary key, the database system ensures that no duplicates appear in a table.

For example, for our emp1 table, the specification

```
CREATE TABLE emp1 (
empno NUMBER(4) CONSTRAINT pk_emp PRIMARY KEY,
. . . );
```

defines the attribute empno as the primary key for the table. Each value for the attribute empno thus must appear only once in the table emp1. A table, of course, may only have one primary key. Note that in contrast to a **unique** constraint, null values are not allowed.

Example:

We want to create a table called PROJECT to store information about projects. For each project, we want to store the number and the name of the project, the employee number of the project's manager, the budget and the number of persons working on the project, and the start date and end date of the project. Furthermore, we have the following conditions:

- a project is identified by its project number,
- the name of a project must be unique,
- the manager and the budget must be defined.

Table definition:

```
CREATE TABLE project1 (
pno NUMBER(3) CONSTRAINT prj_pk PRIMARY KEY,
pname VARCHAR2(60) UNIQUE,
pmgr NUMBER(4) NOT NULL,
persons NUMBER(5),
budget NUMBER(8,2) NOT NULL,
pstart DATE,
pend DATE);
```

A **unique** constraint can include more than one attribute. In this case the pattern **unique**(<column i>, . . . , <column j>) is used. If it is required, for example, that no two projects have the same start and end date, we have to add the table constraint

```
CONSTRAINT no_same_dates UNIQUE(pend, pstart)
```

This constraint has to be defined in the create table command after both columns *pend* and *pstart* have been defined. A primary key constraint that includes more than only one column
can be specified in an analogous way.
Instead of a **not null** constraint it is sometimes useful to specify a default value for an attribute if no value is given, e.g., when a tuple is inserted. For this, we use the **default** clause.
Example: If no start date is given when inserting a tuple into the table *project1*, the project start date should be set to January 1st, 1995:
pstart DATE DEFAULT('01-JAN-95')
Note: Unlike integrity constraints, it is not possible to specify a name for a default.

Check Constraints

Often columns in a table must have values that are within a certain range or that satisfy certain conditions. Check constraints allow users to restrict possible attribute values for a column to admissible ones. They can be specified as column constraints or table constraints.
The syntax for a check constraint is
[CONSTRAINT <name>] CHECK(<condition>)

If a CHECK constraint is specified as a column constraint, the condition can only refer that column.
Example
The name of an employee must consist of upper case letters only; the minimum salary of an employee is 500; department numbers must range between 10 and 100:

```
CREATE TABLE emp1
( . . . ,
ename VARCHAR2(30) CONSTRAINT check_name
        CHECK(ename = UPPER(ename) ),
sal NUMBER(5,2) CONSTRAINT check_sal CHECK(sal >= 500),
deptno NUMBER(3) CONSTRAINT check_deptno
        CHECK(deptno BETWEEN 10 AND 100) );
```

If a CHECK constraint is specified as a table constraint, <condition> can refer to all columns of the table. Note that only simple conditions are allowed. For example, it is not allowed to refer to columns of other tables or to formulate queries as check conditions. Furthermore, the functions SYSDATE and USER cannot be used in a condition. In principle, thus only simple attribute comparisons and logical connectives such as AND, OR, and NOT are allowed. A check condition, however, can include a not null constraint:

```
sal NUMBER(5,2) CONSTRAINT check_sal
        CHECK(sal IS NOT NULL AND sal >= 500),
```

Without the NOT NULL condition, the value *null* for the attribute sal would not cause a violation of the constraint.
Example: At least two persons must participate in a project, and the project's start date must be before the project's end date:

```
CREATE TABLE project1
( . . . ,
persons NUMBER(5) CONSTRAINT check_pers CHECK (persons > 2),
. . . ,
CONSTRAINT dates_ok CHECK(pend > pstart) );
```

In this table definition, check_pers is a column constraint and dates_ok is a table constraint.

Foreign Key Constraints

A foreign key constraint (or referential integrity constraint) can be specified as a column constraint or as a table constraint:

```
[CONSTRAINT <name>] [FOREIGN KEY (<column(s)>)]
REFERENCES <table>[(<column(s)>)]
```

This constraint specifies a column or a list of columns as a foreign key of the referencing table. The referencing table is called the *child-table,* and the referenced table is called the *parent-table.* In other words, one cannot de.ne a referential integrity constraint that refers to a table R before that table R has been created.

The clause FOREIGN KEY has to be used in addition to the clause REFERENCES if the foreign key includes more than one column. In this case, the constraint has to be specified as a table constraint. The clause REFERENCES defines which columns of the parent-table are referenced. If only the name of the parent-table is given, the list of attributes that build the primary key of that table is assumed.

Example: Each employee in the table EMP must work in a department that is contained in the table DEPT:

```
CREATE TABLE emp1
( empno NUMBER(4) CONSTRAINT pk_emp PRIMARY KEY,
. . . ,
deptno NUMBER(3) CONSTRAINT fk_deptno
        REFERENCES dept1(deptno) );
```

The column deptno of the table emp1 (child-table) builds the foreign key and references the primary key of the table dept1 (parent-table). The relationship between these two tables is illustrated in Figure. Since in this table definition the referential integrity constraint includes only one column, the clause FOREIGN KEY is not used. It is very important that a foreign key must refer to the complete primary key of a parent-table, not only a subset of the attributes that build the primary key.

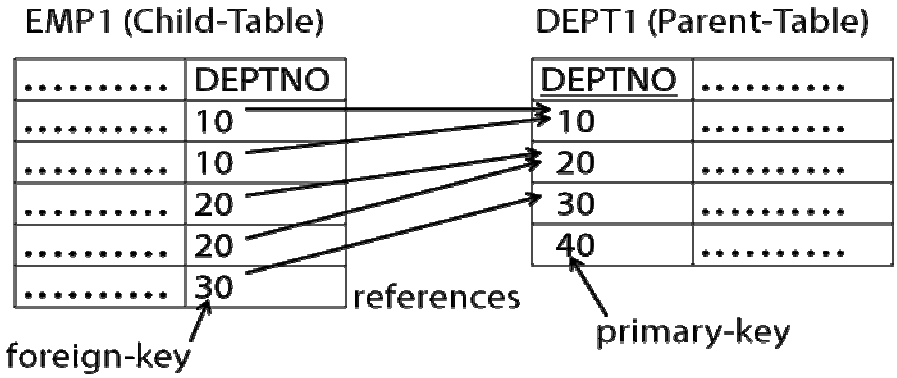

Fig. Foreign Key Constraint between the Tables EMP1 and DEPT1

In order to satisfy a foreign key constraint, each row in the child-table has to satisfy one of following two conditions:

- the attribute value (list of attribute values) of the foreign key must appear as a primary key value in the parent-table, or
- the attribute value of the foreign key is *null* (in case of a composite foreign key, at least one attribute value of the foreign key is *null*)

According to the above definition for the table *emp1*, an employee must not necessarily work in a department, i.e., for the attribute *deptno* the value *null* is admissible.

Example: Each project manager must be an employee:

```
CREATE TABLE project1
 ( pno NUMBER(3) CONSTRAINT prj_pk PRIMARY KEY,
   pmgr NUMBER(4) NOT NULL
        CONSTRAINT fk_pmgr REFERENCES emp1,
  . . . );
```

Because only the name of the parent-table is given (DEPT), the primary key of this relation is assumed. A foreign key constraint may also refer to the same table, i.e., parent-table and child- table are identical.

Example: Each manager must be an employee:

```
CREATE TABLE emp1
( empno NUMBER(4) CONSTRAINT emp_pk PRIMARY KEY,
. . .
mgr NUMBER(4) NOT NULL
        CONSTRAINT fk_mgr REFERENCES emp1,
. . .
);
```

Data Manipulation Language

After a table has been created using the create table command, tuples can be inserted into the table, or tuples can be deleted or modified.

Insertions

The most simple way to insert a tuple into a table is to use the INSERT statement.

```
INSERT INTO <table> [(<column i, . . . , column j>)]
VALUES (<value i, . . . , value j>);
```

For each of the listed columns, a corresponding (matching) value must be specified. Thus an insertion does not necessarily have to follow the order of the attributes as specified in the CREATE TABLE statement. If a column is omitted, the value NULL is inserted instead. If no column list is given, however, for each column as defined in the CREATE TABLE statement a value must be given.

Examples

```
INSERT INTO project1(pno, pname, persons, budget, pstart)
VALUES(313, 'DBS', 4, 150000.42, '10-OCT-94');
```

or

```
INSERT INTO project1
VALUES(313, 'DBS', 7411, NULL, 150000.42, '10-OCT-94', NULL);
```

If there are already some data in other tables, these data can be used for insertions into a new table. For this, we write a query whose result is a set of tuples to be inserted. Such an insert statement has the form

insert into <table> [(<column i, . . . , column j>)] <query>

Example: Suppose we have defined the following table:

```
CREATE TABLE oldemp (
eno      NUMBER(4) NOT NULL,
hdate    DATE);
```

We now can use the table EMP to insert tuples into this new relation:

```
INSERT INTO oldemp  (eno,  hdate)
SELECT empno,  hiredate FROM emp1
WHERE hiredate < '31-DEC-60';
```

Updates

For modifying attribute values of (some) tuples in a table, we use the UPDATE statement:

```
UPDATE <table> SET
<column i> = <expression i>, . . . , <column j> = <expression j>
[where <condition>];
```

An expression consists of either a constant (new value), an arithmetic or string operation, or an SQL query. Note that the new value to assign to <column i> must a the matching data type.

An update statement without a where clause results in changing respective attributes of all tuples in the specified table. Typically, however, only a (small) portion of the table requires an update.

Examples

1. The employee JONES is transferred to the department 20 as a manager and his salary is increased by 1000:

```
UPDATE emp1 SET
job = 'MANAGER', deptno = 20, sal = sal +1000
WHERE ename = 'JONES';
```

2. All employees working in the departments 10 and 30 get a 15% salary increase.

```
UPDATE emp1 SET
sal = sal  1.15 WHERE deptno IN (10,30);
```

Analogous to the insert statement, other tables can be used to retrieve data that are used as new values. In such a case we have a <query> instead of an <expression>.

Example

All salesmen working in the department 20 get the same salary as the manager who has the lowest salary among all managers.

```
UPDATE emp1 SET
sal = (SELECT MIN(sal) FROM emp1
        WHERE job = 'MANAGER')
WHERE job = 'SALESMAN' AND deptno = 20;
```

Deletions

All or selected tuples can be deleted from a table using the DELETE command:

```
DELETE FROM <table> [WHERE <condition>];
```

If the WHERE clause is omitted, all tuples are deleted from the table. An alternative command for deleting all tuples from a table is the TRUNCATE TABLE <table> command.

Example: Delete all projects (tuples) that have been finished before the actual date (system date).

```
DELETE FROM project1 WHERE pend < SYSDATE;
```

SYSDATE is a function in SQL that returns the system date.

Commit and Rollback

A sequence of database modifications, i.e., a sequence of INSERT, UPDATE, and DELETE statements, is called a transaction. Modifications of tuples are temporarily stored in the database system. They become permanent only after the statement COMMIT; has been issued.

As long as the user has not issued the COMMIT statement, it is possible to undo all modifications since the last COMMIT. To undo modifications, one has to issue the statement ROLLBACK;.

It is advisable to complete each modification of the database with a COMMIT (as long as the modification has the expected effect). Note that any data definition command such as CREATE TABLE results in an internal COMMIT. A COMMIT is also implicitly executed when the user terminates an Oracle session.

Views

The SQL command to create a view (virtual table) has the form

```
CREATE [OR REPLACE] VIEW <view-name> [(<column(s)>)]
 AS <select-statement> [WITH CHECK OPTION];
```

where

<view-name> specifies the name of the view and follow the rules for identifiers.

<column(s)> specifies the name of the columns to be used in a view.

If <column(s)> option is not specified, then the view is created with the same column(s) as those specified in the <select-statement>.

AS specifies the actions that will be performed by the view.

<select-statement> specifies the SELECT statement that defines a view. The view may refer to the data contained in other views and tables.

WITH CHECK OPTION forces the data modification statements to fulfill the criteria given in the SELECT statement defining the view. It also ensures that the data is visible after the modifications are made permanent.

The optional clause or replace re-creates the view if it already exists.

Restrictions imposed on views are as follows:

- A view can be created only in the current database.
- A view can be created only if there is SELECT permission on its base table.
- A SELECT INTO statement cannot be used in a view declaration statement.
- A view cannot derive its data from temporary tables.

Example

The following view contains the name, job title and the annual salary of employees working in the department 20:

```
CREATE VIEW dept20 AS
SELECT ename, job, sal□12 annual_salary
FROM emp1
WHERE deptno = 20;
```

In the select statement the column alias ANNUAL SALARY is specified for the expression SAL□12 and this alias is taken by the view. An alternative formulation of the above view definition is

```
CREATE VIEW dept20 (ename, job, annual_salary) AS
SELECT ename, job, sal □ 12 FROM emp1
WHERE deptno = 20;
```

Dropping Views

You can drop a view from a database by using the DROP VIEW statement. When a view is dropped, it has no effect on the underlying table(s). Dropping a view removes its definition and all the permissions assigned to it.

DROP VIEW view_name

where

view_name is the name of the view to be dropped.

You can drop multiple views with a single DROP VIEW statement. The names of views that need to be dropped are separated by commas in the DROP VIEW statement.

5.2 The Tuple Relational Calculus

The tuple relational calculus, is a nonprocedural query language. It describes the desired information without giving a specific procedure for obtaining that information.

A query in the tuple relational calculus is expressed as

$$\{t \mid P(t)\}$$

that is, it is the set of all tuples t such that predicate P is true for t. Following our earlier notation, we use t[A] to denote the value of tuple t on attribute A, and we use $t \in r$ to denote that tuple t is in relation r.

5.2.1 Example Queries

Say that we want to find the branch-name, loan-number, and amount for loans of over $1200:

$$\{t \mid t \in loan \wedge t[amount] > 1200\}$$

Suppose that we want only the loan-number attribute, rather than all attributes of the loan relation. To write this query in the tuple relational calculus, we need to write an expression for a relation on the schema (loan-number). We need those tuples on (loan-number) such that there is a tuple in loan with the amount attribute > 1200. To express this request, we need the construct "there exists" from mathematical logic. The notation

$$\exists t \in r\ (Q(t))$$

means "there exists a tuple t in relation r such that predicate Q(t) is true."

Using this notation, we can write the query "Find the loan number for each loan of an amount greater than $1200" as

$$\{t \mid \exists s \in loan\ (t[loan\text{-}number] = s[loan\text{-}number] \wedge s[amount] > 1200)\}$$

In English, we read the preceding expression as "The set of all tuples t such that there exists a tuple s in relation loan for which the values of t and s for the loan-number attribute are equal, and the value of s for the amount attribute is greater than $1200."

Tuple variable t is defined on only the loan-number attribute, since that is the only attribute having a condition specified for t. Thus, the result is a relation on (loan-number).

Consider the query "Find the names of all customers who have a loan from the Perryridge branch." This query is slightly more complex than the previous queries, since it involves two relations: borrower and loan. As we shall see, however, all it requires is that we have two "there exists" clauses in our tuple-relational-calculus expression, connected by and (∧). We write the query as follows:

$$\{t \mid \exists s \in borrower\ (t[customer\text{-}name] = s[customer\text{-}name] \wedge \exists u \in loan\ (u[loan\text{-}number] = s[loan\text{-}number] \wedge u[branch\text{-}name] = \text{“}Perryridge\text{”}))\}$$

In English, this expression is "The set of all (customer-name) tuples for which the cus- tomer has a loan that is at the Perryridge branch." Tuple variable u ensures that the customer is a borrower at the Perryridge branch. Tuple variable s is restricted to per- tain to the same loan number as s. Figure 5.1 shows the result of this query.

To find all customers who have a loan, an account, or both at the bank, we used the union operation in the relational algebra. In tuple relational calculus, we shall need two "there exists" clauses, connected by or (∨):

$$\{t \mid \exists s \in borrower\ (t[customer\text{-}name] = s[customer\text{-}name]) \vee \exists u \in depositor\ (t[customer\text{-}name] = u[customer\text{-}name])\}$$

This expression gives us the set of all customer-name tuples for which at least one of the following holds:

- The customer-name appears in some tuple of the borrower relation as a borrower from the bank.
- The customer-name appears in some tuple of the depositor relation as a depositor of the bank.

If some customer has both a loan and an account at the bank, that customer appears only once in the result, because the mathematical definition of a set does not allow duplicate members.

If we now want only those customers who have both an account and a loan at the bank, all we need to do is to change the or (∨) to and (∧) in the preceding expression.

$$\{t \mid \exists s \in borrower\ (t[customer\text{-}name] = s[customer\text{-}name]) \wedge \exists u \in depositor\ (t[customer\text{-}name] = u[customer\text{-}name])\}$$

Now consider the query "Find all customers who have an account at the bank but do not have a loan from the bank." The tuple-relational-calculus expression for this query is similar to the expressions that we have just seen, except for the use of the not (¬) symbol:

$$\{t \mid \exists u \in depositor\ (t[customer\text{-}name] = u[customer\text{-}name]) \wedge \neg \exists s \in borrower\ (t[customer\text{-}name] = s[customer\text{-}name])\}$$

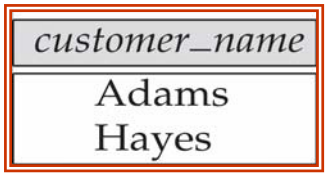

Figure 5.1 Names of all customers who have a loan at the Perryridge branch.

This tuple-relational-calculus expression uses the $\exists$ u $\in$ *depositor* (...) clause to require that the customer have an account at the bank, and it uses the $\neg \exists$ s $\in$ *borrower* (...) clause to eliminate those customers who appear in some tuple of the borrower relation as having a loan from the bank.

The query that we shall consider next uses implication denoted by $\Rightarrow$. The formula P $\Rightarrow$ Q means "P implies Q"; that is, "if P is true, then Q must be true." Note that P $\Rightarrow$ Q is logically equivalent to $\neg$P $\vee$ Q. The use of implication rather than not and or often suggests a more intuitive interpretation of a query in English.

Consider the query "Find all customers who have an account at all branches located in Brooklyn." To write this query in the tuple relational calculus, we introduce the "for all" construct, denoted by $\forall$. The notation

$$\forall t \in r\ (Q(t))$$

means "Q is true for all tuples t in relation r."

We write the expression for our query as follows:

$$\{t \mid \exists r \in customer\ (r[customer\text{-}name] = t[customer\text{-}name]) \wedge (\forall u \in branch\ (u[branch\text{-}city] = \text{“}Brooklyn\text{”} \Rightarrow \exists s \in depositor\ (t[customer\text{-}name] = s[customer\text{-}name] \wedge \exists w \in account\ (w[account\text{-}number] = s[account\text{-}number] \wedge w[branch\text{-}name] = u[branch\text{-}name])))\}$$

In English, we interpret this expression as "The set of all customers (that is, (customer- name) tuples t) such that, for all tuples u in the branch relation, if the value of u on at- tribute branch-city is Brooklyn, then the customer has an account at the branch whose name appears in the branch-name attribute of u."

Note that there is a subtlety in the above query: If there is no branch in Brooklyn, all customer names satisfy the condition. The first line of the query expression is crit- ical in this case — without the condition

$$\exists r \in customer\ (r[customer\text{-}name] = t[customer\text{-}name])$$

if there is no branch in Brooklyn, any value of t (including values that are not customer names in the depositor relation) would qualify.

5.2.2 Formal Definition

We are now ready for a formal definition. A tuple-relational-calculus expression is of the form

$$\{t \mid P(t)\}$$

where P is a formula. Several tuple variables may appear in a formula. A tuple variable is said to be a free variable unless it is quantified by a $\exists$ or $\forall$ .Thus, in

$$t \in loan \wedge \exists s \in customer\ (t[branch\text{-}name] = s[branch\text{-}name])$$

t is a free variable. Tuple variable s is said to be a bound variable.

A tuple-relational-calculus formula is built up out of *atoms*. An atom has one of the following forms:

- s $\in$ r, where s is a tuple variable and r is a relation (we do not allow use of the $\notin$ operator)
- s[x] Θ u[y], where s and u are tuple variables, x is an attribute on which s is defined, y is an attribute on which u is defined, and Θ is a comparison operator ($<, \leq, =, =, >, \geq$); we require that attributes x and y have domains whose members can be compared by Θ
- s[x] Θ c, where s is a tuple variable, x is an attribute on which s is defined, Θ is a comparison operator, and c is a constant in the domain of attribute x

We build up formulae from atoms by using the following rules:

- An atom is a formula.
- If P1 is a formula, then so are $\neg$P1 and (P1).
- If P1 and P2 are formulae, then so are P1 $\vee$ P2, P1 $\wedge$ P2, and P1 $\Rightarrow$ P2.
- If P1 (s) is a formula containing a free tuple variable s, and r is a relation, then $\exists$ s $\in$ r (P1 (s)) and $\forall$ s $\in$ r (P1 (s)) are also formulae.

As we could for the relational algebra, we can write equivalent expressions that are not identical in appearance. In the tuple relational calculus, these equivalences include the following three rules:

1. P1 $\wedge$ P2 is equivalent to $\neg (\neg (P1) \vee \neg (P2))$.
2. $\forall$ t $\in$ r (P1 (t)) is equivalent to $\neg \exists$ t $\in$ r ($\neg$P1(t)).
3. P1 $\Rightarrow$ P2 is equivalent to $\neg$ (P1) $\vee$ P2.

5.3 The Domain Relational Calculus**

A second form of relational calculus, called domain relational calculus, uses domain variables that take on values from an attributes domain, rather than values for an entire tuple. The domain relational calculus, however, is closely related to the tuple relational calculus.

Domain relational calculus serves as the theoretical basis of the widely used QBE language, just as relational algebra serves as the basis for the SQL language.

5.3.1 Formal Definition

An expression in the domain relational calculus is of the form

$$\{< x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

where $x_1, x_2, \ldots, x_n$ represent domain variables. P represents a formula composed of atoms, as was the case in the tuple relational calculus. An atom in the domain relational calculus has one of the following forms:

- $< x_1, x_2, \ldots, x_n > \in$ r, where r is a relation on n attributes and $x_1, x_2, \ldots, x_n$ are domain variables or domain constants.
- x Θ y, where x and y are domain variables and Θ is a comparison operator ($<, \leq, =, =, >, \geq$). We require that attributes x and y have domains that can be compared by Θ.
- x Θ c, where x is a domain variable, Θ is a comparison operator, and c is a constant in the domain of the attribute for which x is a domain variable.

We build up formulae from atoms by using the following rules:

- An atom is a formula.
- If P_1 is a formula, then so are $\neg P_1$ and (P_1).
- If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$, and $P_1 \Rightarrow P_2$.
- If P_1 (x) is a formula in x, where x is a domain variable,
  then $\exists$ x (P_1 (x) and $\forall$ x (P1 (x) are also formulae.

As a notational shorthand, we write

$\exists$ a, b, c (P (a, b, c)) for $\exists$ a ($\exists$ b ($\exists$ c (P (a, b, c))))

5.3.2 Example Queries

We now give domain-relational-calculus queries for the examples that we considered earlier. Note the similarity of these expressions and the corresponding tuple- relational-calculus expressions.

- Find the loan number, branch name, and amount for loans of over $1200:

  $$\{< l, b, a > \mid < l, b, a > \in loan \wedge a > 1200\}$$

- Find all loan numbers for loans with an amount greater than $1200:

  $$\{< l > \mid \exists\, b, a\, (< l, b, a > \in loan \wedge a > 1200)\}$$

Although the second query appears similar to the one that we wrote for the tuple relational calculus, there is an important difference. In the tuple calculus, when we write $\exists$ s for some tuple variable s, we bind it immediately to a relation by writing $\exists$ s $\in$ r. However, when we write $\exists$ b in the domain calculus, b refers not to a tuple, but rather to a domain value. Thus, the domain of variable b is unconstrained until the subformula $< l, b, a > \in$ *loan* constrains b to branch names that appear in the loan relation. For example,

- Find the names of all customers who have a loan from the Perryridge branch and find the loan amount:

  $$\{< c, a > \mid \exists\, l\, (< c, l > \in borrower \wedge \exists\, b\, (< l, b, a > \in loan \wedge b = \text{“Perryridge”}))\}$$

- Find the names of all customers who have a loan, an account, or both at the Perryridge branch:

  $$\{< c > \mid \exists\, l\, (< c, l > \in borrower \wedge \exists\, b, a\, (< l, b, a > \in loan \wedge b = \text{“Perryridge”})) \vee \exists\, a\, (< c, a > \in depositor \wedge \exists\, b, n\, (< a, b, n > \in account \wedge b = \text{“Perryridge”}))\}$$

- Find the names of all customers who have an account at all the branches located in Brooklyn:

  $$\{< c > \mid \exists\, n\, (< c, n > \in \text{customer}) \wedge \forall\, x, y, z\, (< x,y,z > \in \text{branch} \wedge y = \text{“Brooklyn”} \Rightarrow \exists\, a, b\, (< a,x,b > \in \text{account} \wedge < c, a > \in \text{depositor}))\}$$