MIS 331
Database Management Systems

Lecture Notes
(Student Version)

Instructor

Jinsoo Park

Department of Management Information Systems
College of Business and Public Administration
The University of Arizona
Tucson, AZ 85721
E-mail: jpark@bpaosf.bpa.arizona.edu
Web Page: http://jpark.bpa.arizona.edu

---

1 9 9 8

Table of Contents

SQL (Structured Query Language)

- SQL — A Relational Database Language
- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Derived Relations
- Views
- Modification of the Database
- Joined Relations
- Data Definition Language

SQL — A Relational Database Language

- Relational algebra (and calculus) provides a concise notation for representing queries. However, commercial database systems require as more "user-friendly" and high-level declarative query language.

- There are several high-level query languages that have been partially or fully implemented and are available on commercial DBMSs. They include SQL (Structured Query Language), QUEL (Query Language), and QBE (Query By Example).

- Originally, SQL was called SEQUEL (for Structured English QUEry Language) and was designed and implemented at IBM Research as the interface for System R.

- In 1986, a joint effort under way by ANSI and ISO has led to a standard version of SQL, called SQL1. A revised and more expanded standard called SQL2 (also referred to as SQL-92) has been developed. Plans are already underway for SQL3 which will further extended SQL with object-oriented and other recent database concepts.

- Numerous products now support the SQL language. Although the product versions of SQL differ in several language details, the differences are, for the most part, minor.

- SQL has clearly established itself as the standard relational database language.

SQL (Cont.)

Although SQL is referred to as a "query language," it contains many other capabilities besides querying a database.

- Data definition language (DDL) — The SQL DDL provides commands for defining relation schemas, deleting relations, modifying relation schemas, and creating indices.

- Data manipulation language (DML) — The SQL DML includes a query language based on both the relational algebra and the tuple relational calculus. It includes also commands to insert, delete, and modify tuples in the database.

- View definition — The SQL DDL includes commands for defining views.

- Authorization — The SQL data control language includes commands for specifying access rights to relations and views.

- Integrity — The SQL DDL provides the mechanism for specifying integrity constraint.

- Transaction control — The includes commands for specifying beginning and ending of transactions and may include commands for explicitly locking data for concurrency control purpose.

Basic Structure

- SQL is based on set and relational operations with certain modifications and enhancements

- A typical SQL query has the form:

```
SELECT A1, A2, …, An
  FROM r1, r2, …, rm
 WHERE P
```

  - A_i s represent attributes
  - r_i s represent relations
  - P is a predicate

- This query is equivalent to the relational algebra expression:

$$\Pi_{A_1, A_2, \ldots, A_n} (\sigma_p (r_1 \times r_2 \times \ldots \times r_m))$$

- A more complete form of an SQL SELECT expression is:

```
SELECT A1, A2, …, An
  FROM r1, r2, …, rm
 WHERE P
 GROUP BY A1, A2, …, An
HAVING P
 ORDER BY A1, A2, …, An
```

- The result of an SQL query is a relation.

The `SELECT` Clause

- The **`SELECT`** clause corresponds to the *projection* operation of the relational algebra. It is used to list the attributes desired in the result of a query.

  Syntax:
  – Retrieve all attributes and tuples from a relation.

  ```
  SELECT * FROM R
  ```

  – Retrieve certain attributes and all tuples from a relation.

  ```
  SELECT A1, A2, …, An FROM R
  ```

- Find the names of all branches in the *loan* relation.

  ```
  SELECT Branch_name
    FROM LOAN
  ```

  In the "pure" relational algebra syntax, this query would be:

  $$\Pi_{\text{Branch-name}}(\text{LOAN})$$

- An asterisk in the **`SELECT`** clause denotes "all attributes"

  ```
  SELECT *
    FROM LOAN
  ```

- SQL allows *duplicates* in relations as well as in query results.

The `SELECT` Clause (Cont.)

- To force the elimination of duplicates, insert the keyword **`DISTINCT`** after **`SELECT`**.

  Find the names of all branches in the *loan* relation, and remove duplicates.

- The keyword **`ALL`** specifies that duplicates not be removed.

- The **`SELECT`** clause can contain arithmetic expressions involving the operators, +, –, *, and /, and operating on constants or attributes of tuples.

- The query:

```
SELECT Branch_name, Loan_number, Amount *100
  FROM LOAN
```

  would return a relation which is the same as the LOAN relation, except that the attribute Amount is multiplied by 100.

The **WHERE** Clause

- The **WHERE** clause corresponds to the *selection* predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the **FROM** clause.

- Find all loan numbers for loans made at the Tucson branch with loan amounts greater than $1200.

- SQL uses the logical connectives **AND**, **OR**, and **NOT**. It allows the use of arithmetic expressions as operands to the comparison operators.

- SQL includes a **BETWEEN** comparison operator in order to simplify **WHERE** clauses that specify that a value be less than or equal to some value and greater than or equal to some other value.

- Find the loan number of those loans amounts between $90,000 and $100,000 (that is, ≥ $90,000 and ≤ $100,000)

The `FROM` Clause

- The **`FROM`** clause corresponds to the *Cartesian product* operation of the relational algebra. It lists the relations to be scanned in the evaluation of the expression.

- Find the Cartesian product BORROWER $\times$ LOAN

Find the name and loan number of all customers having a loan at the Tucson branch.

The Rename Operation

- The SQL mechanism for renaming relations and attributes is accomplished through the **`AS`** clause:

  old-name **`AS`** *new-name*

- Find the name and loan number of all customers having a loan at the Tucson branch; replace the column name Loan-number with the name Loan-id.

Tuple Variables

- Tuple variables are defined in the **`FROM`** clause via the use of the **`AS`** clause.
- Find the customer names and their loan numbers for all customers having a loan at some branch.

- Find the names of all branches that have greater assets than some branch located in Brooklyn.

String Operations

- SQL includes a string-matching operator for comparisons on character strings. Patterns are described using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.

- Find the names of all customers whose street includes the substring ‘Main’.

- Match the name “Main%”

Ordering the Display of Tuples

- List in alphabetic order the names of all customers having a loan at the Tucson branch.

- We may specify **`DESC`** for descending order or **`ASC`** for ascending order, for each attribute; ascending order is the default.

- SQL must perform a sort to fulfill an **`ORDER BY`** request. Since sorting a large number of tuples may be costly, it is desirable to sort only when necessary.

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.

- *Multiset* versions of some of the relational algebra operators – given multiset relations r_1 and r_2:
  - If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selection σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta (r_1)$.
  - For each copy of tuple t_1 in r_1, there is a copy of tuple $\prod_A (t_1)$ in $\prod_A (r_1)$, where $\prod_A (t_1)$ denotes the projection of single tuple t_1.
  - If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are $c_1 \times c_2$ copies of the tuple $t_1 . t_2$ in $r_1 \times r_2$.

- Suppose relations r_1 with schema (A, B) and r_2 with schema (C) are the following multisets:

$$r_1 = \{(1, a), (2, a)\} \qquad r_2 = \{(2), (3), (3)\}$$

- Then $\prod_B (r_1)$ would be {(a), (a)}, while $\prod_B (r_1) \times r_2$ would be {(a, 2), (a, 2), (a, 3), (a, 3), (a, 3)}

- SQL duplicate semantics:

```
SELECT A1, A2, …, An
  FROM r1, r2, …, rm
 WHERE P
```

  is equivalent to the *multiset* version of the expression:

$$\prod_{A_1, A_2, \ldots, A_n} (\sigma p (r_1 \times r_2 \times \ldots \times r_m))$$

Set Operations

- The set operations **`UNION`**, **`INTERSECT`**, and **`EXCEPT`** operate on relations and correspond to relational algebra operations $\cup$, $\cap$ and $-$.

- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **`UNION ALL`**, **`INTERSECT ALL`** and **`EXCEPT ALL`**. Suppose a tuple occurs m times in r and n time in s, then it occurs:
  - $m + n$ times in r **`UNION ALL`** s
  - $min(m, n)$ times in r **`INTERSECT ALL`** s
  - $max(0, m - n)$ times in r **`EXCEPT ALL`** s

- Find all customers who have a loan, an account, or both:

- Find all customers who have both a loan and an account:

- Find all customers who have an account but no loan:

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value.

  AVG: average value

  MIN: minimum value

  MAX: maximum value

  SUM: sum of values

  COUNT: number of values

- The **COUNT** function returns the number of tuples or values specified in a query. The functions **SUM**, **MAX**, **MIN**, and **AVG** are applied to a set of numeric values and return respectively, the sum, maximum value, minimum value, and average of those values.

- If the **GROUP BY** clause is not used in a **SELECT** statement, the aggregate functions treat the resulting relation from the query as a single group. However, if **GROUP BY** is used, the aggregate functions are applied to subgroups of tuples in the resulting relation (see slides for Grouping).

- The aggregate functions can be applied to display the aggregate information when used in the **SELECT** clause or to select some subgroups based on some aggregate information when used in a **HAVING** clause (see slides for Grouping).

Aggregate Functions — Example

- Find the average account balance at the Tucson branch.

- Find the number of tuples in the CUSTOMER relation.

- Find the number of depositors in the bank.

Aggregate Functions – Examples (`GROUP BY`)

- The **`GROUP BY`** clause logically rearranges the resulting relation into groups. The attribute or attributes given in the **`GROUP BY`** clause are used to form groups. Tuples with the same value on all attributes in the **`GROUP BY`** clause are placed in one group.

- If the aggregate functions appear in the **`SELECT`** clause and if **`GROUP BY`** is used, the aggregate functions are applied to each such group independently.

- *Note*: Attributes in **`SELECT`** clause outside of aggregate functions must appear in **`GROUP BY`** list.

- Find the number of depositors for each branch.

Aggregate Functions – Example (`HAVING` Clause)

- SQL provides a **`HAVING`** clause, which can appear only in conjunction with a **`GROUP BY`** clause, for retrieving groups that satisfy certain conditions.

- *Note*: Predicates in the **`HAVING`** clause are applied after the formation of groups.

- Find the names of all branches where the average account balance is more than $1,200

- The execution order of **`SELECT`**, **`WHERE`**, **`GROUP BY`**, and **`HAVING`** is as follows.
  - The predicate in the **`WHERE`** clause is applied first.
  - Tuples satisfying the **`WHERE`** clause are then placed into groups by the **`GROUP BY`** clause.
  - The **`HAVING`** clause is then applied to each group.
  - The groups that satisfy the **`HAVING`** clause predicate are used by the **`SELECT`** clause to generate the result of the query.

Null Values

- It is possible for tuples to have a null value, denoted by **`null`**, for some of their attributes. **`null`** signifies unknown value or value does not exist.

- The result of any arithmetic expression involving **`null`** is **`null`**.

- Roughly speaking, all comparisons involving **`null`** return *false*. More precisely,

- Any comparison with **`null`** returns *unknown*
  - (*true* **or** *unknown*) = *true*,
    (*false* **or** *unknown*) = *unknown*,
    (*unknown* **or** *unknown*) = *unknown*,
    (*true* **and** *unknown*) = *unknown*,
    (*false* **and** *unknown*) = *false*,
    (*unknown* **and** *unknown*) = *unknown*
  - Result of **`WHERE`** clause predicate is treated as *false* if it evaluates to *unknown.*

Null Values (Cont.)

- Find all loan numbers which appear in the LOAN relation with null values for Amount.

- Total all loan amounts

  Above statement ignores null amounts; result is null if there is no non-null amount.

- All aggregate operations except **`COUNT(*)`** ignore tuples with null values on the aggregated attributes.

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.

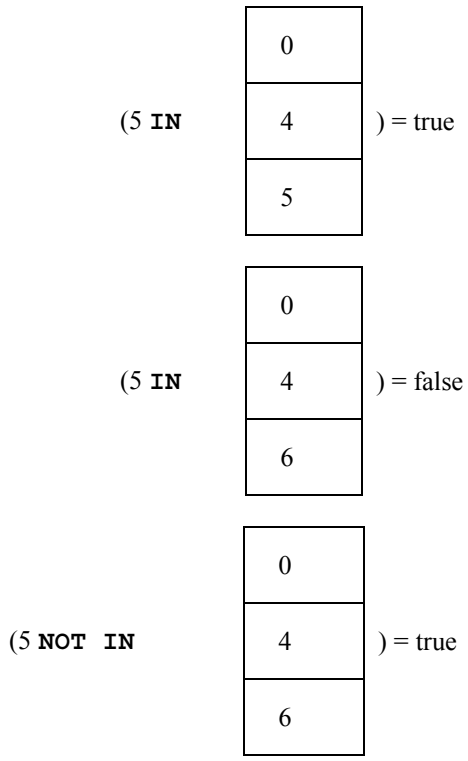
- A subquery is a **`SELECT-FROM-WHERE`** expression that is nested within another query.

- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

Set Membership

- SQL allows testing tuples of membership in a set or a relation. The **`IN`** connective tests for set membership, while the **`NOT IN`** connective tests for the absence of set membership. The **`IN`** or **`NOT IN`** is used in the **`WHERE`** clause.

- F **`in`** $r \Leftrightarrow \exists\, t \in r\, (t = \mathrm{F})$



Example Query

- Find all customers who have both an account and a loan at bank.

- Find all customers who have a loan at the bank but do not have an account at the bank.

- Find all customers who have both an account and a loan at the Tucson branch.

Set Comparison

- A number of set comparison operators can be used to compare a single value to a set or a set to another set.

- Find all branches that have greater assets than some branch located in Brooklyn.

```
SELECT DISTINCT T.Branch_name
  FROM BRANCH AS T, BRANCH AS S
 WHERE T.Assets > S.Assets
   AND  S.Branch_city = "Brooklyn"
```

- = **SOME** operator returns true if the value is equal to some value in the set and is hence equivalent to **IN**. other operators that can be combined with **SOME** include >, >=, <, <=, and < >.

- The keyword **ALL** can also be combined with one of these operators(>, >=, <, <=, and < >). For example, > **ALL** operators returns true if the value is greater than all the values in the set.

- **CONTAINS** (or **NOT CONTAINS**) operator compares two sets of values and returns true if the first set contains (or does not contain) all values in the second set. The **CONTAINS** operator is similar to the **DIVISION** operation of the relational algebra.

- The **CONTAINS** was introduced in the original SEQUEL language. It was dropped from later SQL standards. A likely reason is that processing the **CONTAINS** operator is extremely expensive computationally.

The SOME Clause

- F <comp> **SOME** $r \Leftrightarrow \exists t\ (t \in r \wedge [F <com> t])$
  where <comp> can be: $<, \leq, > \geq, =, \neq$

(5 < **SOME** { 0, 5, 6 }) = true (read: 5 < some tuple in the relation)

(5 < **SOME** { 0, 5 }) = false

(5 = **SOME** { 0, 5 }) = true

(5 ≠ **SOME** { 0, 5 }) = true (since 0 ≠ 5)

- (= **SOME**) ≡ **IN**
- However, (≠ **SOME**) ≢ **NOT IN**
- Example Query:
  Find all branches that have greater assets than some branch located in Brooklyn.

The `ALL` Clause

- F <comp> **`ALL`** $r \Leftrightarrow \forall\ t\ (t \in r \wedge$ [F <com> t])

(5 < **`ALL`** {0, 5, 6}) = false

(5 < **`ALL`** {6, 10}) = true

(5 = **`ALL`** {4, 5}) = false

(5 ≠ **`ALL`** {4, 6}) = true (since 5 ≠ 4 and 5 ≠ 6)

- (≠ ALL) ≡ NOT IN
- However, (= **`ALL`**) ≢ **`IN`**
- Example Query:
  Find the names of all branches that have greater assets than all branches located in Brooklyn.

Test for Empty Relations

- The **`EXISTS`** construct returns the value *true* if the argument subquery is nonempty.
- **`EXISTS`** $r \Leftrightarrow r \neq \emptyset$
- **`NOT EXISTS`** $r \Leftrightarrow r = \emptyset$
- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Example Query:
  Find all customers who have an account at all branches located in Brooklyn.

Test for Absence of Duplicate Tuples

- The **`UNIQUE`** construct tests whether a subquery has any duplicate tuples in its result.

- Find all customers who have one account at the Tucson branch.

- Find all customers who have at least two accounts at the Tucson branch.

Views

- Provide a mechanism to hide certain data from the view of certain users. To create a view we use the command:

  `CREATE VIEW` *v* **`AS`** <query expression>

  where:

  - <query expression> is any legal expression
  - the view name is represented by *v*

- Example Queries:
  - A view consisting of branches and their customers

  - Find all customers of the Tucson branch.

Modification of the Database – Deletion

- The **DELETE** command removes tuples from an existing relation. It may include a **WHERE** clause, similar to the used in an SQL **SELECT** statement, to select the tuples to deleted. A missing **WHERE** clause specifies that all tuples in the relation are to be deleted; however, the relation remains in the database as an empty table.
- The deletion is expressed by

```
DELETE FROM r
 WHERE P
```

  where P represents a predicate and r represents a relation.
- We can delete only whole tuple from a relation, but cannot delete values on only particular attributes.
- Delete all account records at the Tucson branch.

- Delete the records of all accounts with balances below the average at the bank

```
DELETE FROM ACCOUNT
 WHERE Balance < (SELECT AVG(Balance)
                  FROM ACCOUNT)
```

  - Problem: as we delete tuples from DEPOSIT, the average balance changes
  - Solution used in SQL:
    1. First, compute **AVG** balance and find all tuples to delete
    2. Next, delete all tuples found above (without recomputing **AVG** or retesting the tuples)

Modification of the Database – Insertion

- All attributes values for inserted tuples must be members of the corresponding attribute's domain

- To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted.

  Syntax (for inserting a tuple):

  INSERT INTO *r* (A_1, A_2, …, A_n)
  VALUES (V_1, V_2,…, V_n)

  or

  INSERT INTO *r* **VALUES** (V_1, V_2,…, V_n)

  V_1, V_2,…, V_n are listed in the same order in which the corresponding attributes were specified in the **CREATE TABLE** command for R.

  Syntax (for inserting data from a query):

  INSERT INTO *r* (A_1, A_2, …, A_n)
  SELECT – Statement

  The result of the select statement and R must be union compatible.

Modification of the Database – Insertion (Cont.)

- Add a new tuple to ACCOUNT

```
INSERT INTO ACCOUNT
VALUES ("Tucson", A-9732, 1200)
```

  Or equivalently

```
INSERT INTO ACCOUNT (Branch_name, Balance,
                     Account_number)
VALUES ("Tucson", 1200, A-9732)
```

- Add a new tuple to account with balance set to *null*

```
INSERT INTO ACCOUNT
VALUES ("Tucson", A-777, null)
```

- We can insert tuples based on the result of a query.
  - Provide as a gift for all loan customers of the Tucson branch, a $200 savings account. Let the loan number serve as the account number for the new savings account

Modification of the Database – Update

- The **`UPDATE`** command is used to modify attribute values of one or more selected tuples. A **`WHERE`** clause in the **`UPDATE`** command selects the tuples to be modified. An additional **`SET`** clause specifies the attributes to be modified and their new values.

  Syntax:

  ```
  UPDATE r
  SET expression
  <WHERE SELECT – statement>
  ```

  < > represents an optional clause.

- The order is important

- Increase all accounts with balances over $10,000 by 6%, all other account receive 5%.

Update of a View

- Create a view of all loan data in the *loan* relation, hiding the *amount* attribute

```
CREATE VIEW BRANCH_LOAN AS
       SELECT Branch_name, Loan_number
       FROM LOAN
```

- Add a new tuple to BRANCH_LOAN

```
INSERT INTO BRANCH_LOAN
VALUES ("Tucson", "L-307")
```

  This insertion must be represented by the insertion of the tuple

  ("Tucson", "L-307", *null*)

  into the LOAN relation

- There are no restrictions on querying a view. However, a modification is permitted through a view (called updatable view) only if the view is defined in terms of one table and if the view attributes contain the primary key or some other candidate key of the base table.

- Updates on more complex view (consists of more than two relations) are difficult or impossible to translate, and hence are disallowed.

Joined Relations

- Join operations take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **`FROM`** clause.
- *Join type* – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.
- *Join condition* – defines which tuples in the two relations match, and what attributes are present in the result of the join.

Join Types
`INNER JOIN`
`LEFT OUTER JOIN`
`RIGHT OUTER JOIN`
`FULL OUTER JOIN`

Join Conditions
`NATURAL`
`ON` < predicate>
`USING` (A_1, A_2, …, A_n)

Joined Relations – Datasets for Examples

- Relation LOAN

Branch_name	Loan_number	Amount
Downtown	L-170	3000
Redwood	L-230	4000
Tucson	L-260	1700

- Relation BORROWER

Customer_name	Loan_number
Jones	L-170
Smith	L-230
Lee	L-155

■ LOAN **INNER JOIN** BORROWER **ON**
LOAN.Loan_number = BORROWER.Loan_number

Branch_name	Loan_number	Amount	Customer_name	Loan_number
Downtown	L-170	3000	Jones	L-170
Redwood	L-230	4000	Smith	L-230

■ LOAN **NATURAL INNER JOIN** BORROWER

Branch_name	Loan_number	Amount	Customer_name
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith

■ LOAN **LEFT OUTER JOIN** BORROWER **ON**
LOAN.Loan_number = BORROWER.Loan_number

Branch_name	Loan_number	Amount	Customer_name	Loan_number
Downtown	L-170	3000	Jones	L-170
Redwood	L-230	4000	Smith	L-230
Tucson	L-260	1700	*null*	*null*

- LOAN **NATURAL RIGHT OUTER JOIN** BORROWER

Branch_name	Loan_number	Amount	Customer_name
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith
null	L-155	*null*	Lee

- LOAN **FULL OUTER JOIN** BORROWER **USING** (Loan_number)

Branch-name	Loan_number	Amount	Customer_name
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith
Tucson	L-260	1700	*null*
Null	L-155	*null*	Lee

- Find all customers who have either an account or a loan (but not both) at the bank.

```
SELECT Customer_name
  FROM (DEPOSITOR NATURAL FULL OUTER JOIN BORROWER)
 WHERE Account_number IS null
    OR Loan_number IS null
```

Data Definition Language (DDL)

Allows the specification of a set of relations and information about each relation, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints.
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Domain Types in SQL

- **char(n)**. Fixed length character string, with user-specified length *n*.
- **varchar(n)**. Variable length character strings, with user-specified maximum length *n*.
- **int**. Integer (a finite subset of the integers that is machine-dependent).
- **smallint**. Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p, d)**. Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- **real, double precision**. Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n)**. Floating point number, with user-specified precision of at least *n* digits.
- **date**. Dates, containing a (4 digit) year, month and date.
- **time**. Time of day, in hours, minutes and seconds.

– Null values are allowed in all the domain types. Declaring an attribute to **`NOT NULL`** prohibits null values for that attribute.

– **`CREATE DOMAIN`** construct in SQL-92 creates user-defined domain types

```
CREATE DOMAIN Person_name CHAR(20) NOT NULL
```

CREATE TABLE Construct

- An SQL relation is defined using the **CREATE TABLE** command by giving it a name and specify its attributes and constraints.

Syntax:

```
CREATE TABLE R1
(att-name data-type[NULL | NOT NULL],
…
Att-name data-type[NULL | NOT NULL],
PRIMARY KEY (att-name <{, att-name}>),
FOREIGN KEY (att-name) REFERENCES R2 <(att-name)>,
…
FOREIGN KEY (att-name) REFERENCES Rm <(att-name)>,
CHECK ( condition )
```

R_1 is the name of the new relation, and $R_2 \ldots R_m$ are the names of the existing relations. att-name is the name of an attribute. [A | B] denotes either A ro B. < > represents an optional clause, while { } represents a possibly repeated clause.

CREATE TABLE — Example:

```
CREATE TABLE BRANCH
(
   Branch_name    char(15) NOT NULL,
   Branch_city    char(30),
   Assets         integer
)
```

```
CREATE TABLE  BRANCH
(
   Branch_name    char(15) NOT NULL,
   Branch_city    char(30),
   Assets         integer
   PRIMARY KEY (Branch-name),
   CHECK (Asset >= 0)
)
```

```
CREATE TABLE PROJECT
(
    PName       char(20)    NOT NULL,
    PNumber     integer     NOT NULL,
    DNumber     integer     NULL,
    PRIMARY KEY(PNumber),
    FOREIGN KEY(DNumber) REFERENCES
            DEPARTMENT (DNumber)
)
```

- **PRIMARY KEY** declaration on attribute automatically ensures **NOT NULL** in SQL-92

Drop and Alter Table Constructs

- The **`DROP TABLE`** command deletes all information about the dropped relation from the database.
- The **`ALTER TABLE`** command is used to add attributes to an existing relation. All tuples in the relation are assigned *null* as the value for the new attribute. The form of the **`ALTER TABLE`** command is

  `ALTER TABLE` *r* **`ADD`** *A D*

  where *A* is the name of the attribute be added to relation *r* and *D* is the domain of *A*.

- The **`ALTER TABLE`** command can also be used to drop attributes of a relation

  `ALTER TABLE` *r* **`DROP`** *A*

  where *A* is the name of an attribute of relation *r*.

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.

- Domain Constraints

- Referential Integrity

- Assertions

- Triggers

Domain Constraints

- The most elementary form of integrity constraint.

- They test values inserted in the database, and test queries to ensure that the comparisons make sense.

- The **CHECK** clause in SQL-92 permits domains to be restricted:
  - Use **CHECK** clause to ensure that an Hourly-wage domain allows only values greater than a specified value.

```
CREATE DOMAIN Hourly_wage NUMERIC(5,2)
       CONSTRAINT Value_test
             CHECK (value >= 4.00)
```

  - The domain Hourly-wage is declared to be a decimal number with 5 digits, 2 of which are after the decimal point.
  - The domain has a constraint that ensures that the Hourly-wage is greater than 4.00.
  - The clause **CONSTRAINT** Value-test is optional; useful to indicate which constraint an update violated.

Referential Integrity

- Ensures that a value that appears in one relation for given set of attributes also appears for a certain set of attributes in another relation.
  - Example: if "Tucson" is a branch name appearing in one of the tuples in the ACCOUNT relation, then there exists a tuple in the BRANCH relation for branch "Tucson".

- Formal Definition
  - Let r_1 (R_1) and r_2 (R_2) be relations with primary keys K_1 and K_2 respectively.
  - The subset α of R_2 is a *foreign key* referencing K_1 in relation r_1, if for every t_2 in r_2 there must be a tuple t_1 in r_1 such that $t_1[K_1] = t_2[\alpha]$
  - Referential integrity constraint: $\prod_{\alpha} (r_2) \subseteq \prod_{K_1} (r_1)$

- Referential Integrity in the E-R Model
  - Consider relationship set R between entity sets E_1 and E_2. The relational schema for R includes the primary keys K_1 of E_1 and K_2 of E_2. Then K_1 and K_2 form foreign keys on the relational schemas for E_1 and E_2 respectively.
  - Weak entity sets are also a source of referential integrity constraints. For, the relation schema for weak entity set must include the primary key of the entity set on which it depends.

Database Modification

- The following tests must be made in order to preserve the following referential integrity constraint:

$$\Pi_{\alpha}(r_2) \subseteq \Pi_{K}(r_1)$$

- **`INSERT`**. If a tuple t_2 is inserted into r_2, the system must ensure that there is a tuple t_1 in r_1 such that $t_1[K] = t_2[\alpha]$. That is

$$t_2[\alpha] \in \Pi_{K}(r_1)$$

- **`DELETE`**. If a tuple t_1 is deleted from r_1, the system must compute the set of tuples in r_2 that reference t_1:

$$\sigma_{\alpha = t_1[K]}(r_2)$$

  If this set is not empty, either the delete command is rejected as an error (RESTRICTED), or the tuples that reference t_1 must themselves be deleted (CASCADING deletions are possible).

Database Modification (Cont.)

- **`UPDATE`**. There are two cases:
  - If a tuple t_2 is updated in r_2 and the update modifies values for the foreign key α, then a test similar to the insert case is made. Let t_2' denote the new value of tuple t_2. The system must ensure that

    $$t_2'[\alpha] \in \Pi_K (r_1)$$

  - If a tuple t_1 is updated in r_1 , and the update modifies values for the primary key (K), then a test similar to the delete case is made. The system must compute

    $$\sigma_{\alpha = t_1[K]} (r_2)$$

    using the old value of t_1 (the value before the update is applied). If this set is not empty, the update may be rejected as an error (RESTRICTED), or the update may CASCADED to the tuples in the set, or the tuples in the set may be deleted.

Referential Integrity in SQL

- Primary and candidate keys and foreign keys can be specified as part of the SQL **`CREATE TABLE`** statement:
  - The **`PRIMARY KEY`** clause of the **`CREATE TABLE`** statement includes a list of the attributes that comprise the *primary key*.
  - The **`UNIQUE KEY`** clause of the **`CREATE TABLE`** statement includes a list of the attributes that comprise a *candidate key*.
  - The **`FOREIGN KEY`** clause of the **`CREATE TABLE`** statement includes both a list of the attributes that comprise the foreign key and the name of the relation referenced by the *foreign key*.

Referential Integrity in SQL — Examples:

```
CREATE TABLE CUSTOMER
(
    Customer_name    char(20) NOT NULL,
    Customer_street  char(30),
    Customer_city    char(30),
    PRIMARY KEY (Customer_name)
)

CREATE TABLE BRANCH
(
    Branch_name  char(15) NOT NULL,
    Branch_city  char(30),
    Assets       integer,
    PRIMARY KEY (Branch_name)
)

CREATE TABLE ACCOUNT
(
    Branch_name      char(15),
    Account_number   char(10) NOT NULL,
    Balance          integer,
    PRIMARY KEY (Account_number),
    FOREIGN KEY (Branch_name) REFERENCES BRANCH
)

CREATE TABLE DEPOSITOR
(
    Customer_name    char(20) NOT NULL,
    Account_number   char(10) NOT NULL,
    PRIMARY KEY (Customer_name, Account_number),
    FOREIGN KEY (Account_number)
     REFERENCES ACCOUNT,
    FOREIGN KEY (Customer_name)
     REFERENCES CUSTOMER
)
```

Assertions

- An **ASSERTION** is a predicate expressing a condition that we wish the database always to satisfy.

- An **ASSERTION** is SQL-92 takes the form

```
CREATE ASSERTION <Assertion-name>
           CHECK <Predicate>
```

- When an **ASSERTION** is made, the system tests it for validity. This testing may introduce a significant amount of overhead; hence assertions should be used with great care.

Assertions Example:

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

```
CREATE ASSERTION Sum_constraint CHECK
  (NOT EXISTS (SELECT * FROM BRANCH
        WHERE (SELECT SUM (Amount) FROM LOAN
               WHERE LOAN.Branch_name = BRANCH.Branch_name)
                     >= (SELECT SUM (Amount)
                           FROM ACCOUNT
                          WHERE LOAN.Branch_name =
                                BRANCH.Branch_name)))
```

- Every loan has at least one borrower who maintains an account with a minimum balance of $1000.00.

```
CREATE ASSERTION Balance_constraint CHECK
  (NOT EXISTS (SELECT * FROM LOAN
        WHERE NOT EXISTS (SELECT *
               FROM BRROWER, DEPOSITOR, ACCOUNT
               WHERE LOAN.Loan_number =
                     BORROWER.Loan_number
                 AND BORROWER.Customer_name =
                     DEPOSITOR.Customer_name
                 AND DEPOSITOR.Account-number =
                     ACCOUNT.Account_number
                 AND ACCOUNT.Balance >= 1000)))
```

Triggers

- A **`TRIGGER`** is a statement that is executed automatically by the system as a side effect of a modification to the database.

- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.

- The SQL-92 standard does not include triggers, but many implementations support triggers.

Trigger Example

- Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
  - setting the account balance to zero
  - creating a loan in the amount of the overdraft
  - giving this loan a loan number identical to the account number of the overdrawn account

- The condition for executing the trigger is an update to the ACCOUNT relation that results in a negative Balance value.

```
DEFINE TRIGGER Overdraft ON UPDATE OF ACCOUNT T
 (IF NEW T.Balance < 0
 THEN (INSERT INTO LOAN VALUES
           (T.Branch_name, T.Account_number, – NEW T.Balance)
       INSERT INTO BORROWER
        (SELECT Customer_name, Account_number
           FROM DEPOSITOR
          WHERE T.Account_number = DEPOSITOR.Account_number)
       UPDATE ACCOUNT S
          SET S.Balance = 0
        WHERE S.Account_number = T.Account_number))
```

The keyword **NEW** used before *T*.Balance indicates that the value of *T*.Balance after the update should be used; if it is omitted, the value before the update is used.