Database Concept and Systems

CIT 314 (2 Units)

Bilkisu Muhammad-Bello (Mrs.)

Relational Model cont

Basic Concepts

Introduction

- The relational algebra described in the previous unit provides a formal notation for representing queries. The **tuple relational calculus** and the **domain relational calculus** are nonprocedural languages that represent the basic power required in a relational query language.
- The basic relational algebra is a procedural language that is equivalent in power to both forms of the relational calculus when they are restricted to safe expressions.
- We are now going to introduce a query language that is more user friendly, and used to communicate with databases.

Objectives

- At the end of this unit, you should be able to:
  – Query a database using a query language
  – Create database tables
  – Modify database information
  – Insert and delete data from a database

SQL

- SQL uses a combination of relational-algebra and relational-calculus constructs.
- Although we refer to the SQL language as a “query language,” it can do much more than just query a database. It can define the structure of the data, modify data in the database, and specify security constraints.
- SQL’s fundamental constructs and concepts are presented in this unit.
- Note that Individual implementations of SQL may differ in details, or may support only a subset of the full language.

SQL Background

- IBM developed the original version of SQL at its San Jose Research Laboratory (now the Almaden Research Center).
- IBM implemented the language, originally called Sequel, as part of the System R project in the early 1970s. The Sequel language has evolved since then, and its name has changed to SQL (Structured Query Language).
- Many products now support the SQL language. SQL has clearly established itself as the standard relational-database language.
- In 1986, the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) published an SQL standard, called SQL-86.
- IBM published its own corporate SQL standard, the Systems Application Architecture Database Interface (SAA-SQL) in 1987.
- ANSI published an extended standard for SQL, SQL-89, in 1989. The next version of the standard was SQL-92 standard, and the most recent version is SQL:1999.

The SQL language Parts

1. **Data-definition language (DDL).** The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
2. **Interactive 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 tuples into, delete tuples from, and modify tuples in the database.
3. **View definition.** The SQL DDL includes commands for defining views*.*
4. **Transaction control.** SQL includes commands for specifying the beginning and ending of transactions.
5. **Embedded SQL and dynamic SQL**. Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, Java, PL/I, Cobol, Pascal, and Fortran.
6. **Integrity.** The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
7. **Authorization.** The SQL DDL includes commands for specifying access rights to relations and views.

Basic Structure

- The basic structure of an SQL expression consists of three clauses: `select`, `from`, and `where`.
- 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.
- 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.
- 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.

SQL versus Relational Algebra

- A typical SQL query has the form:

  select ***A1, A2, . . .,An***

  from ***r1, r2, . . . , rm***

  where ***P***

- The query is equivalent to the relational-algebra expression:
  - $\prod_{A1, A2,...,An} (\sigma_P (r1 \times r2 \times \cdots \times rm))$

Example schema

- *Branch-schema* = (*branch-name, branch-city, assets*)
- *Customer-schema* = (*customer-name, customer-street, customer-city*)
- *Loan-schema* = (*loan-number, branch-name, amount*)
- *Borrower-schema* = (*customer-name, loan-number*)
- *Account-schema* = (*account-number, branch-name, balance*)
- *Depositor-schema* = (*customer-name, account-number*)

The select Clause

- The result of an SQL query is, of course, a relation. For example:
  - select ***branch-name***
  - from ***loan***
- distinct|all **keywords**
  - select distinct ***branch-name***
  - from ***loan***

- select *
  - select *****
  - from ***loan***

The where Clause

- SQL uses the logical connectives **and, or, and not**. E.g.
  - **select** loan-number
  - **from** loan
  - **where** branch-name = 'Kaduna' **and** amount > 1200
- SQL includes a **between** comparison operator 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. E.g.
  - **select** ***loan-number***
  - **from** ***loan***
  - **where** ***amount*** **between** ***90000*** **and** ***100000***
- instead of
  - **select** ***loan-number***
  - **from** ***loan***
  - **where** ***amount <= 100000*** **and** ***amount >= 90000***

Examples

Examples:

1. $\sigma_{Course\ =\ \text{“CIT314”}}$ (*Student*)
2. $\sigma_{Score\ >\ 50}$ (*Student*)
3. $\sigma_{Course\ =\ \text{“CIT314”}\ \wedge\ Score\ >\ 50}$ (*Student*)
4. $\Pi_{Mat\_No,\ Course,\ Score}$ (*Student*)
5. $\Pi_{Name,Course}$ (*Student*)
6. Π_{Name} ($\sigma_{course\ =\ \text{“CIT314”}}$ (*Student*))

Student Table

Mat_No	Name	Course	Score
17056	Kenny	CIT314	90
18805	Mary	MAT325	45
45813	Bala	CPT313	70
62586	Henry	CPT317	23
75184	Richard	CIT314	66
98235	Ibrahim	CSS314	30
17056	Kenny	CSS314	40

The Rename Operation

- SQL provides a mechanism for renaming both relations and attributes. It uses the **as** clause, taking the form:
  - *old-name* **`as`** *new-name*
- The **as** clause can appear in both the **`select`** and **`from`** clauses. E.g.
  - **`select`** ***customer-name, borrower.loan-number*** **`as`** ***loan-id, amount***
  - **`from`** ***borrower, loan***
  - **`where`** ***borrower.loan-number = loan.loan-number***

String Operations

- SQL specifies strings by enclosing them in single quotes, for example, ’Kaduna’, as we saw earlier.
- A single quote character that is part of a string can be specified by using two single quote characters; for example the string “It’s right” can be specified by ’It”s right’.
- The most commonly used operation on strings is pattern matching using the operator **`like`**. We describe patterns by using two special characters:
  - *Percent (%): The % character matches any substring.*
  - *Underscore (_): The character matches any character.*
- Note that patterns are case sensitive; that is, uppercase characters do not match lowercase characters, or vice versa.

String Operations

- To illustrate pattern matching, we consider the following examples:
- *'Niger%' matches any string beginning with "Niger".*
- *'%idge%' matches any string containing "idge" as a substring, e.g.,* 'Perryridge', 'Rock Ridge', 'Mianus Bridge', and 'Ridgeway'.
- *'_ _ _'matches any string of exactly three characters.*
- *'_ _ _ %' matches any string of at least three characters.*
- SQL expresses patterns by using the **`like`** comparison operator. Consider the query: "Find the names of all customers whose street address includes the substring 'Main'.
- This query can be written as
  - **`select`** ***customer-name***
  - **`from`** ***customer***
  - **`where`** ***customer-name*** **`like`** ***'%Main%'***

Ordering the Display of Tuples

- The **order by** clause causes the tuples in the result of a query to appear in sorted order. E.g.
  - **select distinct** *customer-name*
  - **from** *borrower, loan*
  - **where** *borrower.loan-number = loan.loan-number* **and** *branch-name = 'Kaduna'*
  - **order by** customer-name
- By default, the **order by** clause lists items in ascending order. To specify the sort order, we may specify **desc** for descending order or **asc** for ascending order. Furthermore, ordering can be performed on multiple attributes.
  - **select ***
  - **from** ***loan***
  - **order by** ***amount*** **desc*****, loan-number*** **asc**

Set Operations

- The SQL operations **`union`**, **`intersect`**, and **`except`** operate on relations and correspond to the relational-algebra operations ∪, ∩, and −.
- Like union, intersection, and set difference in relational algebra, the relations participating in the operations must be *compatible;* that is, they must have the same set of attributes.

Examples

- To find all customers having a loan, an account, or both at the bank, we write
  - (**select** ***customer-name***
  - **from** ***depositor)***
  - **union**
  - (**select** ***customer-name***
  - **from** ***borrower)***

- To find all customers who have both a loan and an account at the bank, we write
  - (select distinct customer-name
  - from depositor)
  - **intersect**
  - (select distinct customer-name
  - from borrower)
- To find all customers who have an account but no loan at the bank, we write
  - (select distinct customer-name
  - from depositor)
  - **except**
  - (select customer-name
  - from borrower)

Aggregate Functions

- Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value.
- SQL offers five built-in aggregate functions:
  - Average: `avg`
  - Minimum: `min`
  - Maximum: `max`
  - Total: `sum`
  - Count: `count`
- The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.
- We use the aggregate function `count` frequently to count the number of tuples in a relation. The notation for this function in SQL is `count(*)`. Thus, to find the number of tuples in the *customer* relation, we write
  - **select** `count (*)`
  - **from** ***customer***

Examples

select avg (*balance*)
from *account*
where *branch-name* = 'Minna'

select *branch-name*, **avg** (*balance*)
from *account*
group by *branch-name*

select *branch-name*, **count** (**distinct** *customer-name*)
from *depositor, account*
where *depositor.account-number* = *account.account-number*
group by *branch-name*

select *branch-name*, **avg** (*balance*)
from *account*
group by *branch-name*
having avg (*balance*) > 1200

Null Values

- SQL allows the use of null values to indicate absence of information about the value of an attribute.
- We can use the special keyword **`null`** in a predicate to test for a null value.
- Thus, to find all loan numbers that appear in the *loan relation* with null values for amount, we write
  - **`select`** ***loan-number***
  - **`from`** ***loan***
  - **`where`** ***amount is*** **`null`**
- The predicate **`is not null`** tests for the absence of a null value.

Nested Subqueries

- SQL provides a mechanism for nesting 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, make set comparisons, and determine set cardinality.
- Example: Find all customers who have both a loan and an account at the bank.
  - **`select`** **distinct** ***customer-name***
  - **`from`** ***borrower***
  - **`where`** ***customer-name*** **in (`select`** **customer-name**
  - **`from`** ***depositor)***

Views

- We define a view in SQL by using the **`create view`** command.
- To define a view, we must give the view a name and must state the query that computes the view. The form of the create view command is
  - **`create view`** *v* as <query expression>
- where <query expression> is any legal query expression. The view name is represented by v.

MODIFICATION OF THE DATABASE

We have restricted our attention until now to the extraction of information from the database. Now, we show how to add, remove, or change information with SQL.

Deletion

- A delete request is expressed in much the same way as a query. We can delete only whole tuples; we cannot delete values on only particular attributes.
- SQL expresses a deletion by:
  - **`delete`** from r
  - **`where`** P
- where P represents a predicate and r represents a relation.
- The delete statement first finds all tuples (t) in r for which P(t) is true, and then deletes them from r.
- The where clause can be omitted, in which case all tuples in r are deleted.
- Note that a delete command operates on only one relation. If we want to delete tuples from several relations, we must use one delete command for each relation.

Insertion

- 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.
- Obviously, the attribute values for inserted tuples must be members of the attribute's domain. Similarly, tuples inserted must be of the correct arity.
  - **`insert into`** ***account***
  - **`values`** **('A-9732', 'Perryridge', 1200)**

  - **`insert into`** account (account-number, branch-name, balance)
  - **`values`** ('A-9732', 'Perryridge', 1200)

  - **`insert into`** account (branch-name, account-number, balance)
  - **`values`** ('Perryridge', 'A-9732', 1200)

Updates

- In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple.
- For this purpose, the **`update`** statement can be used. As we could for **`insert`** and **`delete`**, we can choose the tuples to be updated by using a query. Examples
  - **`update`** ***account***
  - **`set`** ***balance = balance * 1.05***

  - **`update`** ***account***
  - **`set`** ***balance = balance * 1.05***
  - **`where`** ***balance >= 1000***

  - **`update`** ***account***
  - **`set`** ***balance = balance * 1.05***
  - **`where`** ***balance >*** **`select`** **`avg`** ***(balance)***
  - **`from`** ***account***

Data-Definition Language

- The SQL DDL allows specification of not only a set of relations, but also information about each relation, including:
  - The schema for each relation
  - The domain of values associated with each attribute
  - The integrity constraints
  - The set of indices to be maintained for each relation
  - The security and authorization information for each relation
  - The physical storage structure of each relation on disk

Domain Types in SQL

- The SQL standard supports a variety of built-in domain types, including:
  - **char(*n*):** A fixed-length character string with user-specified length n. The full form, character, can be used instead.
  - ***varchar(n):*** A variable-length character string with user-specified maximum length n. The full form, character varying, is equivalent.
  - ***int:*** An integer (a finite subset of the integers that is machine dependent). The full form, integer, is equivalent.
  - ***numeric(p, d):*** A fixed-point number with user-specified precision. The number consists of p digits (plus a sign), and d of the p digits are to the right of the decimal point. Thus, numeric(3,1) allows 44.5 to be stored exactly, but neither 444.5 or 0.32 can be stored exactly in a field of this type.
  - ***date:*** A calendar date containing a (four-digit) year, month, and day of the month.

Schema Definition in SQL

- We define an SQL relation by using the **create table command:**
  - `create table` *r (A1D1,A2D2, . . . , AnDn,*
  - (integrity-constraint 1),
  - *. . . ,*
  - (integrity-constraint *k))*
  - where *r is the name of the relation, each Ai is the name of an attribute in the schema* of relation *r, and Di is the domain type of values in the domain of attribute Ai.*
- The allowed integrity constraints include:
  - **Primary key**: The primary key attributes are required to be non-null and unique; that is, no tuple can have a null value for a primary key attribute, and no two tuples in the relation can be equal on all the primary-key
  - **Check(P)**: The check clause specifies a predicate P that must be satisfied by every tuple in the relation.

Schema Definition in SQL cont

- By default **`null`** is a legal value for every attribute in SQL, unless the attribute is specifically stated to be **`not null`**. An attribute can be declared to be **`not null`** in the following way:
  - *account-number* ***char(10)*** **`not null`**
- A newly created relation is empty initially. We can use the **`insert`** command to load data into the relation.
- Many relational-database products have special bulk loader utilities to load an initial set of tuples into a relation.

Drop Table command

- To remove a relation from an SQL database, we use the **drop table** command.
- The drop table command deletes all information about the dropped relation from the database.
- The command `drop table r` is a more drastic action than `delete from r`
- The latter retains relation *r,* but deletes all tuples in r. The former deletes not only all tuples of *r,* but also the schema for r. After r is dropped, no tuples can be inserted into *r* unless it is re-created with the `create table` command.

Alter Table Command

- We use the **`alter table`** command 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 *r* is the name of an existing relation, A is the name of the attribute to be added, and D is the domain of the added attribute.We can drop attributes from a relation by the command
  - **`alter table`** ***r drop A***
- where *r* is the name of an existing relation, and A is the name of an attribute of the relation. Many database systems do not support dropping of attributes, although they will allow an entire table to be dropped.

Examples of SQL data definition

- **create table** ***customer***
- (*customer-name* ***char(20),***
- *customer-street* ***char(30),***
- *customer-city* ***char(30),***
- **primary key (*customer-name))***

- **create table** ***account***
- (*account-number* ***char(10),***
- *branch-name* ***char(15),***
- *balance* ***integer,***
- **primary key (*account-number),***
- **check (*balance >= 0))***

- **create table** ***depositor***
- (*customer-name* ***char(20),***
- *account-number* ***char(10),***
- **primary key (*customer-name, account-number))***