

Information Systems

SQL

Temur Kutsia

Research Institute for Symbolic Computation
Johannes Kepler University of Linz, Austria
`kutsia@risc.uni-linz.ac.at`

Outline

SQL

Table Creation

Populating and Modifying Tables

Querying

SQL

- ▶ SQL: a standard interactive and programming language for getting information from and updating a database.
- ▶ The first version of SQL (initially called SEQUEL) was developed in 1970s at IBM, based on Codd's original ideas.
- ▶ Standardized since 1986.
- ▶ Latest release: SQL 2003.
- ▶ As it is widely accepted, SQL stands for Structured Query Language.
- ▶ SQL specifies:
 - ▶ a data definition language (DDL),
 - ▶ a data manipulation language (DML),
 - ▶ embedded SQL (to make relational databases accessible in other programming languages, like C, Pascal, PL/I).

SQL

- ▶ The SQL language is broken into several distinct parts:
 - ▶ **SQL schema statements**, used to define the data structures stored in the database;
 - ▶ **SQL data statements**, used to manipulate the data structures previously defined using SQL schema statements;
 - ▶ **SQL transaction statements**, used to begin, end, and rollback transactions.
- ▶ In SQL terminology a relation is a table, an attribute is a column and a tuple is a row.

Standard Types

- ▶ **char(n)** a character string of fixed length n,
- ▶ **int** an integer (length can be implementation/hardware dependent),
- ▶ **numeric(i, d)** a numerical value with i digits in the integer part (and a sign) and d digits in the decimal part,
- ▶ **real** a single precision floating point number,
- ▶ **date** storing the years in four digits and the months and the days in two,
- ▶ **time** in hh:mm:ss format.

Standard Types

- ▶ Coercion between compatible types, and the usual operations (e.g. arithmetic for numerical types, or string concatenation) are supported.
- ▶ Many DBMSs also support the BLOB type (Binary Large Object).
- ▶ Simple domain definitions can be made, for example:
`CREATE DOMAIN name type AS CHAR(20)`

Table Creation. Step 1: Design

- ▶ Let's define a table to hold information about a person.
- ▶ First, decide what kind of information should be included in the database. Assume:
 - ▶ Name
 - ▶ Gender
 - ▶ Birth date
 - ▶ Address
 - ▶ Favorite foods
- ▶ Next, assign column names and data types:

Column	Type	Allowable values
Name	Varchar(40)	
Gender	Char(1)	M, F
Birth_date	Date	
Address	Varchar(100)	
Favorite_foods	Varchar(200)	

Table Creation. Step 2: Refinement

Problems with the definition of the person table:

- ▶ The name and address columns are compound (first name, last name for name, and street, city, postal code, country for address).
- ▶ Multiple people can have the same name, gender, birth date, etc., there are no columns in the person table that guarantee uniqueness.
- ▶ The favorite_foods column is a list containing 0, 1, or more independent items. It would be best to create a separate table for this data that includes a foreign key to the person table.

Table Creation. Step 2: Refinement

- ▶ New version of the person table:

Column	Type	Allowable values
Person_id	Smallint	
First_name	Varchar(20)	
Last_name	Varchar(20)	
Gender	Char(1)	M,F
Birth_date	Date	
Street	Varchar(30)	
City	Varchar(20)	
State	Varchar(20)	
Country	Varchar(20)	
Postal_code	Varchar(20)	

- ▶ Person_id will serve as the primary key.
- ▶ The favorite_food table includes a foreign key to the person table:

Column	Type
Person_id	Smallint
Food	Varchar(20)

Table Creation. Step 3: Building SQL Schema Statements

- ▶ After the design is complete, the next step is generate SQL statements to create tables in the database:

```
CREATE TABLE person
(
    person_id SMALLINT,
    fname VARCHAR(20),
    lname VARCHAR(20),
    gender CHAR(1) CHECK (gender in ('M', 'F')),
    birth_date DATE,
    address VARCHAR(30),
    city VARCHAR(20),
    state VARCHAR(20),
    country VARCHAR(20),
    postal_code VARCHAR(20),
    CONSTRAINT pk_person PRIMARY KEY (person_id)
);
```

Table Creation. Step 3: Building SQL Schema Statements

- ▶ If we want to make sure that the table exists, we can use the PostgreSQL `\d` command:
 `person_db=> \d person`

Table Creation. Step 3: Building SQL Schema Statements

- ▶ Creating the favorite_food table:

```
CREATE TABLE favorite_food
(
    person_id SMALLINT,
    food VARCHAR(20),
    CONSTRAINT pk_favorite_food
        PRIMARY KEY (person_id, food),
    CONSTRAINT fk_person_id
        FOREIGN KEY (person_id)
        REFERENCES person (person_id)
);
```

Populating and Modifying Tables. Insert

- ▶ Four SQL data statements: insert, update, delete, and select.
- ▶ Three main components to an insert statement:
 - ▶ The name of the table into which to add the data.
 - ▶ The names of the columns in the table to be populated.
 - ▶ The values with which to populate the columns.
- ▶ `INSERT INTO person`
`(person_id, fname, lname, gender, birth_date)`
`VALUES (1, 'William','Turner', 'M', '1972-05-27');`

Populating and Modifying Tables. Insert

- ▶ More insert statements:

```
INSERT INTO favorite_food (person_id, food)
VALUES (1, 'pizza');
```

```
INSERT INTO favorite_food (person_id, food)
VALUES (1, 'cookies');
```

```
INSERT INTO favorite_food (person_id, food)
VALUES (1, 'nachos');
```

Select

- ▶ We can look at the data just added to the table person by issuing a select statement:
- ▶

```
SELECT person_id, fname, lname, gender, birth_date  
FROM person;
```
- ▶ If there were more than one row in the table, we could add a 'where' clause to specify that we only want to retrieve data for the row having a value of 1 for the person_id column:

```
SELECT person_id, fname, lname, birth_date  
FROM person  
WHERE person_id = 1;
```

Select

- ▶ The following query retrieves William's favorite foods in alphabetic order using an 'order by' statement:
- ▶

```
SELECT food
  FROM favorite_food
 WHERE person_id = 1
 ORDER BY food;
```


Insert and Select

- ▶ Another insert statement adds Susan Smith to the person table:
- ▶

```
INSERT INTO person  
    (person_id, fname, lname, gender, birth_date,  
     address, city, state, country, postal_code)  
VALUES (2, 'Susan','Smith', 'F', '1975-11-02',  
        '23 Maple St.', 'Arlington', 'VA', 'USA', '20220');
```
- ▶ We can query the person table again:
- ▶

```
SELECT person_id, fname, lname, gender, birth_date  
FROM person;
```

Updating

- ▶ When the data about William Turner was added to the table, data for the various address columns was omitted in the insert statement.
- ▶ These columns can be populated via an update statement:
- ▶ UPDATE person
 SET address = '1225 Tremont St.',
 city = 'Boston',
 state = 'MA',
 country = 'USA',
 postal_code = '02138'
 WHERE person_id = 1;
- ▶ update can modify more than one rows at once.
- ▶ If the WHERE clause is omitted than all rows will be updated.

Deleting

- ▶ Delete Susan Smith from the person table:
- ▶ Delete FROM person
WHERE person_id = 2;
- ▶ delete can delete more than one rows at once.
- ▶ If the WHERE clause is omitted than all rows will be deleted.

When Good Statements Go Bad

- ▶ Nonunique primary key:
- ▶ INSERT INTO person
 (person_id, fname, lname, gender, birth_date)
 VALUES (1, 'Charles','Fulton', 'M', '1968-01-15');
- ▶ Error message will be given.

When Good Statements Go Bad

- ▶ Nonexistent foreign key:
- ▶ `INSERT INTO favorite_food (person_id, food)
VALUES (999, 'lasagna');`
- ▶ There is no person in the person table with the id 999. An error message will be issued.

When Good Statements Go Bad

- ▶ Column value violation:
- ▶ UPDATE person
 SET gender = 'Z'
 WHERE person_id = 1;
- ▶ Error message. The gender value 'Z' violates CHECK constraint.

Dropping Tables

- ▶ `DROP TABLE favorite_food;`
Drops the table `favorite_food`;
- ▶ `DROP TABLE person;`
Drops the table `person`;

Querying

- ▶ select statement.
- ▶ Before executing queries, the server checks the following things:
 - ▶ Do you have permission to execute the statement?
 - ▶ Do you have permission to access the desired data?
 - ▶ Is your statement syntax correct?
- ▶ If the query passes these three tests, then it is handed to the query optimizer.
- ▶ The query optimizer determines the most efficient way to execute the query and created the execution plan used by the server.
- ▶ Once the server has finished executing the query, the result set is returned to the calling application.

Querying

- ▶ Query example (suppliers-parts-projects database):
- ▶ `SELECT sname, city
FROM S;`
- ▶ The result table will be returned, that contains two columns and five rows.

Query Clauses

- ▶ The select statement is made up from several components, not all of them are mandatory:
- ▶ **SELECT:** Determines which columns to include in the query's result set.
- ▶ **FROM:** Identifies the tables from which to draw data and how the tables should be joined.
- ▶ **WHERE:** Restricts the number of rows in the final result set.
- ▶ **GROUP BY:** Used to group rows together by common column values.
- ▶ **HAVING:** Restricts the number of rows in the final result set using grouped data.
- ▶ **ORDER BY:** Sorts the rows of the final result set by one or more columns.

The SELECT Clause

- ▶ Show all the columns in the suppliers table:

```
SELECT *  
FROM S;
```

- ▶ In addition to specifying all of the columns via the asterisk character, you can explicitly name the columns we are interested in, such as:

```
SELECT sno, sname, status, city  
FROM S;
```

- ▶ We can choose to include only a subset of the columns in the suppliers table as well:

```
SELECT sno, sname  
FROM S;
```

The SELECT Clause

- ▶ We can include in the select clause such things as:
 - ▶ Literals, such as numbers or strings
 - ▶ Expressions, such as `transaction.amount * -1`
 - ▶ Built-in function calls, such as `ROUND(transaction.amount, 2)`

```
SELECT sno,  
       'Supplier',  
       status * 10,  
       UPPER(sname)  
FROM S;
```

- ▶ When the query simply calls built-in functions and doesn't retrieve data from any tables, there is no need for a FROM clause:

```
SELECT version ( ), current_date, 2+2;
```

The SELECT Clause

- ▶ Adding a column alias after each element of the SELECT clause will display the aliases as the column name:

```
SELECT sno,  
       'Supplier',  
       status * 10 AS status_x_10,  
       UPPER(sname) AS last_name_upper  
FROM S;
```

The SELECT Clause

- ▶ In some cases, a query might return duplicate rows of data:

```
SELECT city  
FROM S;
```

- ▶ To get distinct rows, we can add the keyword DISTINCT:

```
SELECT DISTINCT city  
FROM S;
```

- ▶ Generating a distinct set of results requires the data to be sorted, which can be time consuming for large result sets.

The FROM Clause

- ▶ The FROM clause defines the tables used by a query, along with the means of linking the tables together.
- ▶ Three types of tables can be used in the FROM clause
 - ▶ Permanent tables (i.e., created using the create table statement)
 - ▶ Temporary tables (i.e., rows returned by a subquery)
 - ▶ Virtual tables (i.e., created using the create view statement)
- ▶ Queries on the previous slides used permanent tables.

The FROM Clause. Subqueries

- ▶ The FROM clause using a temporary table:

```
SELECT e.sname, e.city  
FROM (SELECT sname, status, city FROM S) e;
```

- ▶ A subquery against the S table returns three columns, and the containing query references two of the three available columns.
- ▶ The subquery is referenced by the containing query via its alias, which, in this case, is e.

The FROM Clause. Views

- ▶ A view is a query that is stored in the data dictionary.
- ▶ It looks and acts like a table, but there is no data associated with a view.
- ▶ When we issue a query against a view, the query is merged with the view definition to create a final query to be executed.

```
CREATE VIEW supplier_view AS
SELECT sno, sname, status
FROM S;
```

- ▶ After the view has been created, no additional data is created: the select statement is simply stored by the server for future use.
- ▶ Now that the view exists, we can issue queries against it:

```
SELECT sno, sname
FROM supplier_view;
```

The FROM Clause. Joins

- ▶ Joins link information from several tables together.

```
SELECT sno, sname, s.city, pno, pname  
FROM s INNER JOIN p  
ON s.city = p.city;
```

The WHERE Clause

- ▶ The where clause is the mechanism for filtering out unwanted rows from your result set.
- ▶ Selecting only those rows from S that have London in the city column:

```
SELECT *  
FROM s  
WHERE city = 'London';
```

- ▶ Selecting only those rows from S that have Paris in the city column and whose status is greater than 10:

```
SELECT *  
FROM s  
WHERE city = 'Paris' AND status > 10;
```

The WHERE Clause

- ▶ Selecting those rows from P that have in the pname column parts whose names start either with 'S' or with 'C':

```
SELECT *  
FROM p  
WHERE pname LIKE 'S%' OR pname like 'C%';
```

- ▶ Get all triples of supplier numbers and the city names such that the suppliers concerned are colocated in the city:

```
SELECT A.sno AS SA, B.sno as SB, A.city  
From S as A, S as B  
WHERE A.city = B.city AND A.sno < B.sno;
```

The GROUP BY and HAVING Clauses

- ▶ Sometimes we want the server to change the data before giving back the result
- ▶ GROUP BY groups data by column values.
- ▶ HAVING filters group data in the same way the WHERE clause helps us to filter raw data.

```
SELECT city  
FROM S  
GROUP BY city;
```

- ▶ The result set contains one row for each distinct value in the city column.

The GROUP BY and HAVING Clauses

- ▶ To see how many suppliers in each city are, we can use an aggregate function in the SELECT clause to count the number of rows in each group:

```
SELECT city, COUNT(*) AS how_many
FROM S
GROUP BY city;
```

- ▶ To filter out the cities with one supplier only, we can use the HAVING clause:

```
SELECT city, COUNT(*) AS how_many
FROM S
GROUP BY city
HAVING COUNT(*)>1;
```

The ORDER BY Clause

- ▶ The order by clause is the mechanism for sorting our result set using either raw column data or expressions based on column data.

- ▶ Sorted in ascending order:

```
SELECT *  
FROM spj  
ORDER BY qty;
```

- ▶ Sorted in descending order:

```
SELECT *  
FROM spj  
ORDER BY qty DESC;
```

Querying Multiple Tables

- ▶ Join: The mechanism for bringing multiple tables together in the same query.
- ▶ Various kinds of joins: Cross join (cartesian product), inner join, outer joins
- ▶ Cross join is the simplest kind of join:

```
SELECT *  
FROM S CROSS JOIN P;
```


Querying Multiple Tables

- ▶ Inner join: Joining tables on their common columns
SELECT sno, sname, status,
 s.city, pno, pname, weight
FROM s INNER JOIN p ON s.city = p.city
- ▶ If the names of the columns used to join the two tables are identical, which is true in the previous query, we can use the USING subclause instead of the ON subclause:
SELECT sno, sname, status,
 s.city, pno, pname, weight
FROM s INNER JOIN p USING (city);

Querying Multiple Tables

- ▶ Three or more tables can be queried in a similar way:

```
SELECT sno, sname, status,  
       s.city, pno, pname, weight, jno  
FROM s INNER JOIN p USING (city)  
     INNER JOIN j USING (city);
```

- ▶ We can have subqueries as tables:

```
SELECT sno, sname, status,  
       s.city, pno, pname, weight, jno  
FROM s INNER JOIN  
     (SELECT pno,city,pname,weight  
      FROM P  
      WHERE weight > 14)  
     AS e USING (city)  
     INNER JOIN j USING (city);
```

Set Operations

- ▶ Union, intersection, set minus, complement.
- ▶ UNION combines all rows from two tables and requires the number of columns in the tables to be the same:

```
SELECT sno, sname
FROM S
UNION
SELECT pno, pname
FROM P;
```

- ▶ Union removes duplicate rows from the result.
- ▶ If you want all the rows, including duplicates, use UNION ALL.

Set Operations

- ▶ INTERSECT takes the intersection of the rows from two tables and requires the number of columns in the tables to be the same:

```
(SELECT sno, sname
FROM S
UNION
SELECT pno, pname
FROM P) INTERSECT
(SELECT sno, sname
FROM S
WHERE status > 10);
```

- ▶ INTERSECT removes duplicate rows from the result.
- ▶ If you want all the rows, including duplicates, use INTERSECT ALL.

Set Operations

- ▶ The EXCEPT operation returns the first table minus any overlap with the second table and requires the number of columns in the tables to be the same:

```
(SELECT sno, sname
FROM S
UNION
SELECT pno, pname
FROM P) EXCEPT
(SELECT sno, sname
FROM S
WHERE status > 10);
```

- ▶ EXCEPT removes duplicate rows from the result.
- ▶ If you want all the rows, including duplicates, use EXCEPT ALL.

Embedded SQL

- ▶ Embedded SQL is defined to allow access to databases from general purpose programming languages (Perl, C, Ada, etc.) which are called host languages.
- ▶ The SQL statements in the host language are enclosed in an EXEC SQL, END-EXEC (or simply semicolon) pair.
- ▶ Inside an SQL statement variable names of the host language can be referred by attaching a colon as a prefix.
- ▶ The program in the host language that uses embedded SQL must also contain an SQLSTATE variable, in which the status code is returned after every SQL statement.

Embedded SQL

Example

- ▶ Assume the 'number' host variable contains a supplier number whose name and status we want to fetch into 'n' and 'st'.

- ▶ EXEC SQL

```
    SELECT sname, status
        INTO :n, :st
        FROM S
        WHERE sno = :number ;
```

References



Alan Beaulieu.
Learning SQL.
O'Reilly, 2005.