SQL

Structured Query Language (SQL)

- Standard language for commercial relational DBMSs
- originally called **SEQUEL** - (Structured English QUEry Language)
- designed in IBM Research for an experimental DBMS called SYSTEM R

SQL standards

- SQL-86 (SQL1)
- SQL-92 (SQL2)
- SQL:1999 (SQL3)
- SQL:2003
- SQL:2006
- SQL:2008

New standards are composed of core and extensions

SQL structure

Sql commands can be categorized in two groups

1. DDL - Data Definition Language
   – commands to create schemas, relations
   – commands to create domains and other constructs (trigger, views, assertion, constraints)
2. DML - Data Manipulation Language
   – data retrieval and modifications insert, delete , update

SQL uses terms

– relation - > **table**

– tuple - > **row** , (**record**)

– attribute - > **column**, (**field**)

SQL as a declarative language

SQL is a declarative language

– In contrast to relational algebra, SQL specifies only what it wants and not how it should be done

Declarative approach:

```
SELECT * FROM STUDENT
WHERE Class_num = 2;
```

Imperative approach (python like syntax):

```
students = read_students();
for student in students  :
    if  student.Class_num == 2:
        print(student)
```

SQL statements can be implemented in different ways

Database and database schema

A **database** is a collection of tables and other constructs and is the main container on an SQL environment.

```
CREATE DATABASE  University ;
```

In PostgreSQL and many other RDBMS, a **database schema** is a namespace that contains named database objects, views, indexes, etc.

- a database may contain one or multiple schemas.
  - some DBMSs such as MySQL equates the notion of schema and database

```
CREATE SCHEMA administration;
```

CREATE TABLE statement

The **CREATE TABLE** is used to specify a new relation (table)

```
CREATE TABLE  table_name(
 attr1    type_1,
 attr2    type_2,
 ...
 attrn    type_n
);
```

Attributes have specific names and data types from the set of data types supported by the DBMS (int, char, varchar, etc.)

```
CREATE TABLE  Student(
 Ssn    int,
 Fname  varchar(50),
 Lname  varchar(50),
);
```

Attributes are ordered in the sequence in which they are specified in the CREATE TABLE command

Attribute Data Types and Domains

Basic data types available for attributes

Numeric

- INT or INTEGER, SMALLINT
- FLOAT or REAL, DOUBLE PRECISION
- DECIMAL(i,j), DEC(i,j), NUMERIC(i,j) formatted numbers

Character-strings

- CHAR(n) or CHARACTER(n) - fixed length
- VARCHAR(n) or CHARACTER VARYING(n) - varying length
  - characters enclosed within single quotation marks (case sensitive)
- CLOB (CHARACTER LARGE OBJECT) (Postgres uses TEXT instead of CLOB)

Attribute Data Types and Domains

Bit-strings

- BIT(n) , BIT VARYING(n)
- BLOB or BINARY LARGE OBJECT or (BYTEA in PostgreSQL) used for storing images and other multimedia files

Boolean

- BOOLEAN - with two possible values TRUE and FALSE

Date

- DATE (in the form YYYY-MM-DD)
- TIME (with 10 positions of in the form HH:MM:SS ex. ‘09:12:47’)
- DATETIME (TIMESTAMP postgresql) (year, month, day, hour, minute, second and second parts) ‘2019-12-12 09:12:47.242’

Declaring a new data type (domain)

```
CREATE DOMAIN SSN_TYPE as CHAR(9)
```

New data type (domain) can be easily changed

– for instance more characters in our example

The CHECK clause can be used to further specify domain

```
CREATE DOMAIN EMPLOYEE_AGE AS INTEGER
CHECK (EMPLOYEE_AGE > 15 AND EMPLOYEE_AGE < 90);
```

Specifying constraints

Basic constraints are specified as part of table creation after domain declaration

- PRIMARY KEY - specifies that an attribute is the primary key (for composite primary keys a list of attributes is enclosed in parentheses)
- UNIQUE - attribute must be unique, but not part of the primary key - specifies secondary keys
- NOT NULL - nulls are not permitted for a particular attribute (NOT NULL is implicitly specified for attributes of the primary key)
- DEFAULT - specifies a default value for an attribute (if DEFAULT is not specified, the default value is NULL for those attributes not having NOT NULL clause)
- CHECK - restricts attribute or domain values
- REFERENCES - specifies a set of attributes to which a foreign key refers

CREATE TABLE - example

```
CREATE TABLE Student (
    Ssn CHAR(9) PRIMARY KEY,
    Matr_num CHAR(7) UNIQUE,
    Fname VARCHAR(45) NOT NULL,
    Lname VARCHAR (45) NOT NULL,
    Class_num INT CHECK (Class_num > 0 and Class_num < 5)
          DEFAULT 1,
    Dept_code VARCHAR(4) REFERENCES Department(Dept_code)
);
```

In this CREATE TABLE statement

- Ssn - specified as a primary key
- Matr_num - matriculation number is a secondary key (has unique values)
- Fname and Lname - cannot be set to NULL
- Class_num - the class of the student can be (1, 2, 3, or 4) and the default value is 1 (freshman)
- Dept_code - represents a foreign key to the table Department

Constraints on Tuples using CHECK

Tuple based constraints

– are checked whenever a tuple is inserted or modified

```
CHECK (Class_num > 0 and Class_num < 5)
```

Operators used with CHECK

– comparison operators =, <, <=, >, >=, <>

– [NOT] IN (<list of values>)

– [NOT] BETWEEN (<range of values>) - *(inclusive)*

– [NOT] LIKE (<text pattern>)

Example:

```
CREATE TABLE  EMPLOYEE(
...
age int CHECK (age BETWEEN 16 AND 80)
);
```

Key, entity and referential integrity constraints

The following example shows:

– how to specify primary keys with more attributes

– how to specify foreign key using the foreign key clause

```
CREATE TABLE DEPT_LOCATION (
 Dept_number int not null,
 Dept_location varchar(20) not null,
 PRIMARY KEY (Dept_number, Dept_location),
 FOREIGN KEY (Dept_number) REFERENCES Department(DNumber)
);
```

UNIQUE constraint on more attributes is specified in the same way as PRIMARY KEY

Primary and foreign key constraints specified explicitly

```
CREATE TABLE DEPARTMENT (
 DNumber int ,
 Mgr_ssn char(9) not null,
 ...
 CONSTRAINT DEPTPK
  PRIMARY KEY(DNumber),
 CONSTRAINT DEPTUNIQUE
  UNIQUE(DName),
 CONSTRAINT DEPT_MGR_FK
  FOREIGN KEY (Mgr_ssn) REFERENCES Employee(Ssn)
      ON DELETE SET DEFAULT
      ON UPDATE CASCADE
);
```

In this example we specified

- constraint names explicitly
  - PostgreSQL, in contrast to some other systems, does not require constraint names to be unique within a schema (but only per-table)
- *referential triggered action* to the foreign key

Referential triggered actions

The **referential triggered actions** can be attached to any foreign key

1. **ON DELETE** - specifies how to respond to deleting of the referenced row
   - CASCADE - deletes all rows(records) related to the deleted row in the referenced table
   - SET NULL / SET DEFAULT - set values in the current table to be null or a default value, respectively
   - NO ACTION - referential integrity constraint will be violated and the delete operation will be rejected
2. **ON UPDATE** - specifies how to respond to the update operation of the primary key in the referenced row
   - CASCADE - update values in rows to be the same as changed primary key values in the referenced table
   - SET NULL / SET DEFAULT - set values in the current table to be null or a default value, respectively
   - NO ACTION - referential integrity constraint will be violated and the update operation will be rejected

DROP commands

DROP commands are used to delete a database or database elements such an index, table or view

- DROP DATABASE database_name
  - this action deletes database and catalog entries
  - cannot be undone and should be used with caution
  - we have to be connected to another database to execute DROP DATABASE statement
- DROP TABLE table_name
  - removes an existing table from the database
  - all content of a table will be deleted and it cannot be undone

The ALTER table commands

The ALTER table commands are used to change already created tables and it can be used to:

– add new columns

```
ALTER TABLE table_name
ADD new_column_name column_definition ;
```

– rename a table, change a column or remove columns

```
 ALTER TABLE table_name
  RENAME TO table_name1 ;
```

```
ALTER TABLE table_name
  ALTER COLUMN column_name TYPE column_definition ;
```

```
ALTER TABLE Employee
  RENAME COLUMN old_name TO new_name,
  DROP COLUMN column_name;
```

Adding constraints using ALTER table

ALTER table statement can be used to add a new constraint such as a foreign key

– specially useful for self referencing and creation of foreign keys which couldn't be created in CREATE TABLE statements

```
ALTER TABLE  table_name
  ADD CONSTRAINT fk_constraint_name
    FOREIGN KEY (column_1) REFERENCES parent_table(column_2);
```

– adding the primary key as constraint

```
ALTER TABLE  table_name
  ADD PRIMARY KEY (column1_name, column2_name, ...);
```

– To drop a constraint we use:

```
ALTER TABLE  table_name
  DROP CONSTRAINT constraint_name;
```

The INSERT command

The INSERT INTO command is used to add tuples to a relation (rows to a table)

```
INSERT INTO STUDENT(S_id, Matr_num, Student_name, Class)
VALUES (1, 9240006, 'John Brown', 1) ;
```

In the previous example explicit attribute names are specified

- values are listed in the same order as the attributes
- all not specified attributes are set to their DEFAULT or to NULL
- if the attribute list is not specified then values should be listed in the same order in which attributes were specified in the CREATE TABLE command

```
INSERT INTO STUDENT
VALUES (1, 9240006, 'John Brown', 1) ;
```

The INSERT command variations

The INSERT statement can insert multiple tuples in the single command

```
INSERT INTO STUDENT(S_id, Matr_num, Student_name, Class)
VALUES
(1, 9240006, 'John Brown', 1),
(2, 5763576, 'Christine Smith', 2) ,
(3, 1069362, 'Leslie Connor', 1) ;
```

Yet another variation of the INSERT statement can use a query to populate a relation with tuples

```
INSERT INTO FRESHMAN(S_id, Matr_num, Student_name)
SELECT S_id, Matr_num
FROM STUDENT
WHERE Class = 1;
```

The UPDATE command

The Update command is used to modify attribute values of one or more selected tuples

– SET clause specifies attributes whose values should be modified

```
UPDATE STUDENT
SET Class = 3
WHERE S_id = 2;
```

If we want to increment the class of all students then we change SET statement in the following way

```
UPDATE STUDENT
SET Class = Class + 1;
```

– Class attribute on the left side represents the new value

– Class attribute on the right represents the old value

– WHERE clause is missing here

The DELETE command

The DELETE command is used to remove one or more selected tuples from a relation

- includes WHERE clause similar to that used in an SQL query
- deletes from only one table at a time (can propagate to other tables if referential triggered actions are specified)

```
DELETE FROM STUDENT
WHERE S_id = 2;
```

If WHERE clause is missing then all rows in the table are to be deleted

```
DELETE FROM STUDENT;
```

Basic SQL queries

A basic SQL statement or a **select-from-where** block has the following form:

```
SELECT [DISTINCT] <attribute list>
FROM <table list>
WHERE <condition>;
```

- *<attribute list>* list of attributes that should be retrieved by the query
- *<table list>* list of tables required to process the query
- *<condition>* conditional (Boolean) expression that identifies tuples to be retrieved by the query

Basic logical operators used for comparing attributes and literal constants in the where clause are:

=, <, <=, >, >=, <>

Basic SQL query - example

```
SELECT DISTINCT Student_name
FROM Student
WHERE Class=1;
```

- SELECT clause corresponds not to the selection **but** to the projection operator in the relational algebra
  - attribute list contains attributes of the projection
- WHERE clause contains conditions which are specified within the selection operator in the relational algebra
- DISTINCT clause eliminates duplicates

SQL queries and Relational algebra

The following example represents correspondence between relational algebra and SQL queries

```
SELECT DISTINCT FName, LName
FROM Employee
WHERE DNo=4 AND Salary > 30000;
```

$$\pi_{FName,LName}(\sigma_{Dno=\text{"4"} \wedge Salary>30000}(Employee))$$

Simple SQL queries

If an SQL query doesn't have WHERE clause, it returns all tuples (because WHERE condition is always true)

– since DISTINCT is omitted it returns all duplicates of FName and LName values

```
SELECT FName, LName
FROM Employee
```

If the SELECT clause contains * then the query returns all attributes

```
SELECT *
FROM Employee
WHERE DNo=4 AND Salary > 30000;
```

Substring pattern matching in SQL queries

Partial character pattern matching can be done in SQL using LIKE comparison operator. The following two special characters are important to string pattern matching:

- [%] - used to match an arbitrary number of characters (zero or more characters)
- [_] - used to match any single character

Example: Retrieve names of all students whose address is in Mannheim.

```
SELECT Student_name
FROM Student
WHERE Address LIKE '%Mannheim'
```

ORDER BY clause

Tuples in the relational model are not ordered because they belong to a set of tuples

In contrast, SQL queries give us the possibility to order the result of a query by the values of one or more of the attributes using the ORDER BY clause

```
SELECT DISTINCT FName, LName
FROM Employee
ORDER BY FName ASC, LName DESC
```

– ascending and descending order can be explicitly specified using keywords **ASC** and **DESC**

– default order is the ascending order of values (applied when the order is not specified)

Rename operation using AS (aliasing)

SQL allows the user to rename a relation, attributes or both.

```
SELECT S.ssn AS s_ssn,
       S.first_name ||' '|| S.last_name as s_name,
       S.class AS s_class
FROM Student AS S
```

– Aliasing corresponds to the rename operation in the relational algebra

$$\rho_{S(B_1,B_2,\ldots B_n)}(R)$$

– FROM clause can contain rename of the table with all attributes

```
FROM Student AS S( s_ssn, s_fname, s_lname)
```

Operations can be applied in the SELECT clause (in our example operator ll presents concatenation of strings)

Set operations

Set operation as part of relational algebra are also incorporated in SQL

- UNION - union operation
- EXCEPT - set difference operation
- INTERSECT - intersection operation

The relations resulting from this operations are sets of values and don't have duplicates. Set operations apply only to union-compatible relations. Attributes must have the same domains and must appear in the same order

Set operation example

```
SELECT * FROM R
UNION
SELECT * FROM S;
```

The union operation with keyword UNION in this example can be replaced with INTERSECT (EXCEPT) in the case of intersection (set difference) operation.

ALL keyword is used to retrieve duplicates in queries with set operations and we have:

- UNION ALL
- INTERSECT ALL
- EXCEPT ALL

Cartesian product operation

Queries are mostly executed over more relations. Cartesian product of two relations is a list of all pairs of elements from those two relations

In SQL, Cartesian product is expressed by putting two (or more table names) in the FROM clause separated by comma sign

```
SELECT *
FROM Employee, Dependent;
```

Cartesian product can also be expressed by using *cross join* clause.

```
SELECT *
FROM Employee CROSS JOIN Dependent;
```

Cross product example

R

A	B
a1	b1
a1	b2
a2	b3

S

A	C
a1	c1
a3	c1

CROSS PRODUCT

A	B	A	C
a1	b1	a1	c1
a1	b1	a3	c1
a1	b2	a1	c1
a1	b2	a3	c1
a2	b3	a1	c1
a2	b3	a3	c1

To extract related tuples Cartesian product (Cross product) is combined with the WHERE conditions (selection conditions in relational algebra) and then it's called **JOIN**

JOIN OPERATION

Inner joins can be expressed using JOIN keyword in the FROM clause

```
SELECT *
FROM Employee JOIN Dependent
   ON Employee.ssn = Dependent.essn;
```

The same result gives the following command using Cartesian product.

```
SELECT *
FROM Employee, Dependent
WHERE Employee.ssn = Dependent.essn;
```

DIFFERENT TYPES OF JOINS

There are different types of joins and different ways to write the same query

– JOIN from the previous slide can also be written in PostgreSQL using INNER JOIN which gives the same result

```
SELECT *
FROM Employee INNER JOIN Dependent
   ON Employee.ssn = Dependent.essn;
```

– in the ON clause can be used other comparison operators from the set $\theta \in \{=, <, \leq, >, \geq, \neq\}$ (in the relational algebra called theta join) and operators are written as: =, <, <=, >, >=, <>

```
SELECT *
FROM Employee JOIN Dependent
   ON Employee.ssn <> Dependent.essn;
```

OUTER JOINS

Outer joins can be used for the following queries:

– write a query to list all employees who have no dependents

```
SELECT *
FROM Employee LEFT JOIN Dependent
   ON Employee.ssn = Dependent.essn
WHERE Dependent.essn is null;
```

– write a query to list all dependents who don't have an employ they depend on

```
SELECT *
FROM Employee RIGHT JOIN Dependent
   ON Employee.ssn = Dependent.essn
WHERE Employee.ssn is null;
```

SELF JOIN

A JOIN of a table with itself is called self join. Consider a query to retrieve employees with their supervisors

– self join cannot be implemented writing the same table name as in the following example because the problem with the same names occurs

```
SELECT *                                    (WRONG)
FROM Employee JOIN Employee
   ON Employee.super_ssn = Employee.ssn;
```

– correct way to write it is by using rename of tables.

```
SELECT E.name AS Employee_name, S.name AS Supervisor_name
FROM Employee E JOIN Employee S
   ON E.super_ssn = S.ssn;
```

Nested queries

Nested queries are used when queries require that existing values be fetched and then be used in comparisons.

- query is composed of the *nested* and the *outer* query.
- tuples of values can be used in comparisons if we place them in parentheses as illustrated in the following query

```
SELECT DISTINCT Essn
FROM WORKS_ON
WHERE (Pno, Hours) IN ( SELECT Pno, Hours
                        FROM WORKS_ON
                        WHERE Essn='123456789');
```

The query retrieves ssn of all employees which work the same (Project,Hours) combination as the employee with the given essn '123456789' in the nested query

Operators for nested queries

In addition to the IN operator, comparison can be done using other operators:

- ANY (some SOME) - =, >, >=, <, <=, <> can be used with this operators
  - (= ANY) - returns true if the value is equal to some value in the nested set
- > ALL - returns true if the value is greater than all values from the nested query
- [NOT] EXISTS - used for correlated queries

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > ALL ( SELECT Salary
                     FROM EMPLOYEE
                     WHERE Dno=5 );
```

Previous query selects the names of all employees having salary greater than salary of all employees in the specified department

Grouping and aggregation

Aggregate functions are used to summarize data from multiple tuples into a single tuple. Mostly used with the grouping which makes distinct groups to be summarized.

Important built in aggregate functions are:

- COUNT - returns number of tuples or values
- SUM - returns the sum of all values
- MAX - returns the maximum value in the set (multiset)
- MIN - returns minimum value in the set of values
- AVG - gives the average(mean) of the set of attribute values

GROUP BY clause is used to group tuples according to different attribute values

- a new group is created for each distinct value
- output attributes can be only those in the *group by* clause and in the aggregate functions

Grouping and aggregation examples

Simple grouping query to count a number of freshman students.

```
SELECT Class_num, COUNT(*) Num_of_students
FROM STUDENT
GROUP BY Class_num
HAVING Class_num=1;
```

Aggregate functions can be used in correlated nested queries as in the following example

```
SELECT name
FROM EMPLOYEE E
WHERE ( SELECT COUNT (*)
        FROM DEPENDENT D
        WHERE E.Ssn=D.Essn ) >= 2;
```

Views (Virtual tables)

A view is a **virtual table** derived from one or more other tables or views.

- a view isn't necessarily physically stored in the database as base tables
- update operations on views are limited

```
CREATE VIEW Department_info
AS
SELECT Dname, COUNT(*), SUM(Salary)
FROM Department d, Employee e
WHERE d.Dnumber = e.Dno
GROUP BY Dname
```

Queries are specified on views in the same way as on base tables.

```
SELECT * FROM Department_info
WHERE Total_salary>130000;
```

Indexes

Index are part of the physical organization of database

- it is a data structure that improves the speed of data retrieval operations
- primarily used to enhance database performance by improving query execution
- when used inappropriate can result in slower performance

```
CREATE INDEX matr_number_idx
ON Student (matr_number);
```

- unique constraint (index) is automatically created when a primary key or an unique constraint is defined
- partial index is an index that contains only part of a table, (partial index is created using WHERE clause)

Example: Create a unique index on the column matr_number in the table Student.

Stored procedures

Stored procedures and functions are code elements which extend plain SQL with procedural elements e.g. control structures, loops and complex calculations

– PostgreSQL comes with PL/pgSQL procedure language

```
CREATE FUNCTION function_name(<argument_list>)
 RETURNS type AS
BEGIN
  staments;
END;
LANGUAGE 'language_name';
```

Triggers

Triggers are stored procedures which are automatically invoked when a special event in the database occurs

- can be invoked when a row of table is inserted (updated or deleted)
- can be used to track changes made to tables or to enforce business rules

```
CREATE TRIGGER [trigger_name]
[BEFORE | AFTER]
{INSERT | UPDATE | DELETE}
ON [table_name]
[FOR EACH ROW]
[TRIGGER_BODY]
```

Trigger example

Trigger function can be defined in two steps:

```
CREATE OR REPLACE FUNCTION student_last_name_change()
RETURNS trigger AS
$BODY$
BEGIN
   IF NEW.student_name <> OLD.student_name THEN
     INSERT INTO Student_audit(id, student_name, changed_on)
     values(OLD.id, OLD.student_name, now());
   END IF;
RETURN NEW;
END;
$BODY$
```

```
CREATE TRIGGER student_lname_trigger
BEFORE UPDATE ON Student
FOR EACH ROW EXECUTE PROCEDURE student_last_name_change();
```

Review questions

– Describe the following six clauses used in the syntax of an SQL retrieval query.

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

– Explain using NULL values with comparison operators in SQL?

– How NULLs are treated when aggregate functions are applied?

– Explain how aggregate functions and grouping are used. Which aggregate functions we mentioned?